

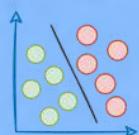
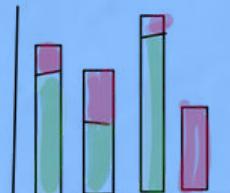
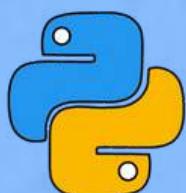
**FREE**

# DATA SCIENCE

**FULL ARCHIVE**

320+ Data Science posts

580+ pages



Daily Dose of  
Data Science



[blog.DailyDoseofDS.com](http://blog.DailyDoseofDS.com)



# Table of Contents

<i>The Must-Know Categorisation of Discriminative Models</i> .....	12
<i>Where Did The Regularization Term Originate From?</i> .....	18
<i>How to Create The Elegant Moving Bubbles Chart in Python?</i> .....	22
<i>Gradient Checkpointing: Save 50-60% Memory When Training a Neural Network</i> .....	24
<i>Gaussian Mixture Models: The Flexible Twin of KMeans</i> .....	28
<i>Why Correlation (and Other Summary Statistics) Can Be Misleading</i> .....	33
<i>MissForest: A Better Alternative To Zero (or Mean) Imputation</i> .....	35
<i>A Visual and Intuitive Guide to The Bias-Variance Problem</i> .....	39
<i>The Most Under-appreciated Technique To Speed-up Python</i> .....	41
<i>The Overlooked Limitations of Grid Search and Random Search</i> .....	44
<i>An Intuitive Guide to Generative and Discriminative Models in Machine Learning</i> .....	48
<i>Feature Scaling is NOT Always Necessary</i> .....	55
<i>Why Sigmoid in Logistic Regression?</i> .....	58
<i>Build Elegant Data Apps With The Coolest Mito-Streamlit Integration</i> .....	62
<i>A Simple and Intuitive Guide to Understanding Precision and Recall</i> .....	64
<i>Skimpy: A Richer Alternative to Pandas' Describe Method</i> .....	69
<i>A Common Misconception About Model Reproducibility</i> .....	71
<i>The Biggest Limitation Of Pearson Correlation Which Many Overlook</i> .....	76
<i>Gigasheet: Effortlessly Analyse Upto 1 Billion Rows Without Any Code</i> .....	78
<i>Why Mean Squared Error (MSE)?</i> .....	82
<i>A More Robust and Underrated Alternative To Random Forests</i> .....	90
<i>The Most Overlooked Problem With Imputing Missing Values Using Zero (or Mean)</i> .....	93
<i>A Visual Guide to Joint, Marginal and Conditional Probabilities</i> .....	95
<i>Jupyter Notebook 7: Possibly One Of The Best Updates To Jupyter Ever</i> .....	96
<i>How to Find Optimal Epsilon Value For DBSCAN Clustering?</i> .....	97
<i>Why R-squared is a Flawed Regression Metric</i> .....	99
<i>75 Key Terms That All Data Scientists Remember By Heart</i> .....	102



<i>The Limitation of Static Embeddings Which Made Them Obsolete .....</i>	109
<i>An Overlooked Technique To Improve KMeans Run-time .....</i>	116
<i>The Most Underrated Skill in Training Linear Models.....</i>	119
<i>Poisson Regression: The Robust Extension of Linear Regression.....</i>	125
<i>The Biggest Mistake ML Folks Make When Using Multiple Embedding Models .....</i>	126
<i>Probability and Likelihood Are Not Meant To Be Used Interchangeably.....</i>	129
<i>SummaryTools: A Richer Alternative To Pandas' Describe Method. ....</i>	135
<i>40 NumPy Methods That Data Scientists Use 95% of the Time.....</i>	136
<i>An Overly Simplified Guide To Understanding How Neural Networks Handle Linearly Inseparable Data.....</i>	138
<i>2 Mathematical Proofs of Ordinary Least Squares .....</i>	145
<i>A Common Misconception About Log Transformation .....</i>	146
<i>Raincloud Plots: The Hidden Gem of Data Visualisation.....</i>	149
<i>7 Must-know Techniques For Encoding Categorical Feature .....</i>	153
<i>Automated EDA Tools That Let You Avoid Manual EDA Tasks .....</i>	154
<i>The Limitation Of Silhouette Score Which Is Often Ignored By Many .....</i>	156
<i>9 Must-Know Methods To Test Data Normality.....</i>	159
<i>A Visual Guide to Popular Cross Validation Techniques .....</i>	163
<i>Decision Trees ALWAYS Overfit. Here's A Lesser-Known Technique To Prevent It. ....</i>	167
<i>Evaluate Clustering Performance Without Ground Truth Labels.....</i>	169
<i>One-Minute Guide To Becoming a Polars-savvy Data Scientist.....</i>	172
<i>The Most Common Misconception About Continuous Probability Distributions .....</i>	174
<i>Don't Overuse Scatter, Line and Bar Plots. Try These Four Elegant Alternatives. ....</i>	175
<i>CNN Explainer: Interactively Visualize a Convolutional Neural Network.....</i>	178
<i>Sankey Diagrams: An Underrated Gem of Data Visualization .....</i>	180
<i>A Common Misconception About Feature Scaling and Standardization.....</i>	181
<i>7 Elegant Usages of Underscore in Python.....</i>	184
<i>Random Forest May Not Need An Explicit Validation Set For Evaluation.....</i>	185
<i>Declutter Your Jupyter Notebook Using Interactive Controls .....</i>	188
<i>Avoid Using Pandas' Apply() Method At All Times .....</i>	190



<i>A Visual and Overly Simplified Guide To Bagging and Boosting .....</i>	<b>192</b>
<i>10 Most Common (and Must-Know) Loss Functions in ML.....</i>	<b>195</b>
<i>How To Enforce Type Hints in Python?.....</i>	<b>196</b>
<i>A Common Misconception About Deleting Objects in Python.....</i>	<b>197</b>
<i>Theil-Sen Regression: The Robust Twin of Linear Regression .....</i>	<b>200</b>
<i>What Makes The Join() Method Blazingly Faster Than Iteration?.....</i>	<b>202</b>
<i>A Major Limitation of NumPy Which Most Users Aren't Aware Of .....</i>	<b>205</b>
<i>The Limitations Of Elbow Curve And What You Should Replace It With .....</i>	<b>206</b>
<i>21 Most Important (and Must-know) Mathematical Equations in Data Science .....</i>	<b>210</b>
<i>Beware of This Unexpected Behaviour of NumPy Methods.....</i>	<b>213</b>
<i>Try This If Your Linear Regression Model is Underperforming.....</i>	<b>214</b>
<i>Pandas vs Polars — Run-time and Memory Comparison .....</i>	<b>216</b>
<i>A Hidden Feature of a Popular String Method in Python.....</i>	<b>218</b>
<i>The Limitation of KMeans Which Is Often Overlooked by Many .....</i>	<b>219</b>
<i>🚀 Jupyter Notebook + Spreadsheet + AI — All in One Place With Mito.....</i>	<b>221</b>
<i>Nine Most Important Distributions in Data Science.....</i>	<b>223</b>
<i>The Limitation of Linear Regression Which is Often Overlooked By Many .....</i>	<b>229</b>
<i>A Reliable and Efficient Technique To Measure Feature Importance .....</i>	<b>231</b>
<i>Does Every ML Algorithm Rely on Gradient Descent?.....</i>	<b>233</b>
<i>Why Sklearn's Linear Regression Has No Hyperparameters?.....</i>	<b>235</b>
<i>Enrich The Default Preview of Pandas DataFrame with Jupyter DataTables ..</i>	<b>237</b>
<i>Visualize The Performance Of Linear Regression With This Simple Plot .....</i>	<b>238</b>
<i>Enrich Your Heatmaps With This Simple Trick .....</i>	<b>240</b>
<i>Confidence Interval and Prediction Interval Are Not The Same .....</i>	<b>241</b>
<i>The Ultimate Categorization of Performance Metrics in ML .....</i>	<b>243</b>
<i>The Coolest Matplotlib Hack to Create Subplots Intuitively .....</i>	<b>247</b>
<i>Execute Python Project Directory as a Script.....</i>	<b>249</b>
<i>The Most Overlooked Problem With One-Hot Encoding .....</i>	<b>250</b>
<i>9 Most Important Plots in Data Science .....</i>	<b>252</b>
<i>Is Categorical Feature Encoding Always Necessary Before Training ML Models? .....</i>	<b>254</b>
<i>Scikit-LLM: Integrate Sklearn API with Large Language Models .....</i>	<b>257</b>



<i>The Counterintuitive Behaviour of Training Accuracy and Training Loss .....</i>	258
<i>A Highly Overlooked Point In The Implementation of Sigmoid Function .....</i>	262
<i>The Ultimate Categorization of Clustering Algorithms .....</i>	265
<i>Improve Python Run-time Without Changing A Single Line of Code .....</i>	267
<i>A Lesser-Known Feature of the Merge Method in Pandas .....</i>	269
<i>The Coolest GitHub-Colab Integration You Would Ever See .....</i>	271
<i>Most Sklearn Users Don't Know This About Its LinearRegression Implementation .....</i>	272
<i>Break the Linear Presentation of Notebooks With Stickyland.....</i>	274
<i>Visualize The Performance Of Any Linear Regression Model With This Simple Plot .....</i>	275
<i>Waterfall Charts: A Better Alternative to Line/Bar Plot.....</i>	277
<i>What Does The Google Styling Guide Say About Imports.....</i>	278
<i>How To Truly Use The Train, Validation and Test Set .....</i>	280
<i>Restart Jupyter Kernel Without Losing Variables.....</i>	283
<i>The Advantages and Disadvantages of PCA To Consider Before Using It .....</i>	284
<i>Loss Functions: An Algorithm-wise Comprehensive Summary .....</i>	286
<i>Is Data Normalization Always Necessary Before Training ML Models? .....</i>	288
<i>Annotate Data With The Click Of A Button Using Pigeon .....</i>	291
<i>Enrich Your Confusion Matrix With A Sankey Diagram.....</i>	292
<i>A Visual Guide to Stochastic, Mini-batch, and Batch Gradient Descent .....</i>	294
<i>A Lesser-Known Difference Between For-Loops and List Comprehensions .....</i>	297
<i>The Limitation of PCA Which Many Folks Often Ignore .....</i>	299
<i>Magic Methods: An Underrated Gem of Python OOP .....</i>	302
<i>The Taxonomy Of Regression Algorithms That Many Don't Bother To Remember .....</i>	305
<i>A Highly Overlooked Approach To Analysing Pandas DataFrames .....</i>	307
<i>Visualise The Change In Rank Over Time With Bump Charts .....</i>	308
<i>Use This Simple Technique To Never Struggle With TP, TN, FP and FN Again ..</i>	309
<i>The Most Common Misconception About Inplace Operations in Pandas .....</i>	311
<i>Build Elegant Web Apps Right From Jupyter Notebook with Mercury .....</i>	313
<i>Become A Bilingual Data Scientist With These Pandas to SQL Translations ....</i>	315
<i>A Lesser-Known Feature of Sklearn To Train Models on Large Datasets .....</i>	317
<i>A Simple One-Liner to Create Professional Looking Matplotlib Plots .....</i>	319



<i>Avoid This Costly Mistake When Indexing A DataFrame.....</i>	321
<i>9 Command Line Flags To Run Python Scripts More Flexibly.....</i>	324
<i>Breathing KMeans: A Better and Faster Alternative to KMeans .....</i>	326
<i>How Many Dimensions Should You Reduce Your Data To When Using PCA?... </i>	329
<i>🚀 Mito Just Got Supercharged With AI!.....</i>	332
<i>Be Cautious Before Drawing Any Conclusions Using Summary Statistics .....</i>	334
<i>Use Custom Python Objects In A Boolean Context.....</i>	336
<i>A Visual Guide To Sampling Techniques in Machine Learning.....</i>	338
<i>You Were Probably Given Incomplete Info About A Tuple's Immutability .....</i>	342
<i>A Simple Trick That Significantly Improves The Quality of Matplotlib Plots ....</i>	344
<i>A Visual and Overly Simplified Guide to PCA.....</i>	346
<i>Supercharge Your Jupyter Kernel With ipyflow .....</i>	349
<i>A Lesser-known Feature of Creating Plots with Plotly .....</i>	351
<i>The Limitation Of Euclidean Distance Which Many Often Ignore.....</i>	353
<i>Visualising The Impact Of Regularisation Parameter .....</i>	356
<i>AutoProfiler: Automatically Profile Your DataFrame As You Work.....</i>	358
<i>A Little Bit Of Extra Effort Can Hugely Transform Your Storytelling Skills .....</i>	360
<i>    A Nasty Hidden Feature of Python That Many Programmers Aren't Aware Of .....</i>	362
<i>Interactively Visualise A Decision Tree With A Sankey Diagram .....</i>	365
<i>Use Histograms With Caution. They Are Highly Misleading! .....</i>	367
<i>Three Simple Ways To (Instantly) Make Your Scatter Plots Clutter Free .....</i>	369
<i>A (Highly) Important Point to Consider Before You Use KMeans Next Time .....</i>	372
<i>Why You Should Avoid Appending Rows To A DataFrame .....</i>	375
<i>Matplotlib Has Numerous Hidden Gems. Here's One of Them.....</i>	377
<i>A Counterintuitive Thing About Python Dictionaries .....</i>	379
<i>Probably The Fastest Way To Execute Your Python Code .....</i>	382
<i>Are You Sure You Are Using The Correct Pandas Terminologies? .....</i>	384
<i>Is Class Imbalance Always A Big Problem To Deal With?.....</i>	387
<i>A Simple Trick That Will Make Heatmaps More Elegant .....</i>	389
<i>A Visual Comparison Between Locality and Density-based Clustering .....</i>	391
<i>Why Don't We Call It Logistic Classification Instead? .....</i>	392
<i>A Typical Thing About Decision Trees Which Many Often Ignore.....</i>	394



<i>Always Validate Your Output Variable Before Using Linear Regression .....</i>	395
<i>A Counterintuitive Fact About Python Functions.....</i>	396
<i>Why Is It Important To Shuffle Your Dataset Before Training An ML Model ....</i>	397
<i>The Limitations Of Heatmap That Are Slowing Down Your Data Analysis.....</i>	398
<i>The Limitation Of Pearson Correlation Which Many Often Ignore .....</i>	399
<i>Why Are We Typically Advised To Set Seeds for Random Generators?.....</i>	400
<i>An Underrated Technique To Improve Your Data Visualizations .....</i>	401
<i>A No-Code Tool to Create Charts and Pivot Tables in Jupyter.....</i>	402
<i>If You Are Not Able To Code A Vectorized Approach, Try This. .....</i>	403
<i>Why Are We Typically Advised To Never Iterate Over A DataFrame?.....</i>	405
<i>Manipulating Mutable Objects In Python Can Get Confusing At Times .....</i>	406
<i>This Small Tweak Can Significantly Boost The Run-time of KMeans .....</i>	408
<i>Most Python Programmers Don't Know This About Python OOP .....</i>	410
<i>Who Said Matplotlib Cannot Create Interactive Plots? .....</i>	412
<i>Don't Create Messy Bar Plots. Instead, Try Bubble Charts!.....</i>	413
<i>You Can Add a List As a Dictionary's Key (Technically)!.....</i>	414
<i>Most ML Folks Often Neglect This While Using Linear Regression .....</i>	415
<i>35 Hidden Python Libraries That Are Absolute Gems .....</i>	416
<i>Use Box Plots With Caution! They May Be Misleading. .....</i>	417
<i>An Underrated Technique To Create Better Data Plots .....</i>	418
<i>The Pandas DataFrame Extension Every Data Scientist Has Been Waiting For</i>	419
<i>Supercharge Shell With Python Using Xonsh .....</i>	420
<i>Most Command-line Users Don't Know This Cool Trick About Using Terminals .....</i>	421
<i>A Simple Trick to Make The Most Out of Pivot Tables in Pandas.....</i>	422
<i>Why Python Does Not Offer True OOP Encapsulation.....</i>	423
<i>Never Worry About Parsing Errors Again While Reading CSV with Pandas.....</i>	424
<i>An Interesting and Lesser-Known Way To Create Plots Using Pandas.....</i>	425
<i>Most Python Programmers Don't Know This About Python For-loops .....</i>	426
<i>How To Enable Function Overloading In Python.....</i>	427
<i>Generate Helpful Hints As You Write Your Pandas Code .....</i>	428
<i>Speedup NumPy Methods 25x With Bottleneck .....</i>	429
<i>Visualizing The Data Transformation of a Neural Network .....</i>	430



<i>Never Refactor Your Code Manually Again. Instead, Use Sourcery!</i> .....	431
<i>Draw The Data You Are Looking For In Seconds</i> .....	432
<i>Style Matplotlib Plots To Make Them More Attractive</i> .....	433
<i>Speed-up Parquet I/O of Pandas by 5x</i> .....	434
<i>40 Open-Source Tools to Supercharge Your Pandas Workflow</i> .....	435
<i>Stop Using The Describe Method in Pandas. Instead, use Skimpy.</i> .....	436
<i>The Right Way to Roll Out Library Updates in Python</i> .....	437
<i>Simple One-Liners to Preview a Decision Tree Using Sklearn</i> .....	438
<i>Stop Using The Describe Method in Pandas. Instead, use Summarytools.</i> .....	439
<i>Never Search Jupyter Notebooks Manually Again To Find Your Code</i> .....	440
<i>F-strings Are Much More Versatile Than You Think</i> .....	441
<i>Is This The Best Animated Guide To KMeans Ever?</i> .....	442
<i>An Effective Yet Underrated Technique To Improve Model Performance</i> .....	443
<i>Create Data Plots Right From The Terminal</i> .....	444
<i>Make Your Matplotlib Plots More Professional</i> .....	445
<i>37 Hidden Python Libraries That Are Absolute Gems</i> .....	446
<i>Preview Your README File Locally In GitHub Style</i> .....	447
<i>Pandas and NumPy Return Different Values for Standard Deviation. Why? ...</i>	448
<i>Visualize Commit History of Git Repo With Beautiful Animations</i> .....	449
<i>Perfplot: Measure, Visualize and Compare Run-time With Ease</i> .....	450
<i>This GUI Tool Can Possibly Save You Hours Of Manual Work</i> .....	451
<i>How Would You Identify Fuzzy Duplicates In A Data With Million Records?....</i>	452
<i>Stop Previewing Raw DataFrames. Instead, Use DataTables.</i> .....	454
<i>🚀 A Single Line That Will Make Your Python Code Faster</i> .....	455
<i>Prettify Word Clouds In Python</i> .....	456
<i>How to Encode Categorical Features With Many Categories?</i> .....	457
<i>Calendar Map As A Richer Alternative to Line Plot</i> .....	458
<i>10 Automated EDA Tools That Will Save You Hours Of (Tedious) Work</i> .....	459
<i>Why KMeans May Not Be The Apt Clustering Algorithm Always</i> .....	460
<i>Converting Python To LaTeX Has Possibly Never Been So Simple</i> .....	461
<i>Density Plot As A Richer Alternative to Scatter Plot</i> .....	462
<i>30 Python Libraries to (Hugely) Boost Your Data Science Productivity</i> .....	463
<i>Sklearn One-liner to Generate Synthetic Data</i> .....	464



<i>Label Your Data With The Click Of A Button</i> .....	465
<i>Analyze A Pandas DataFrame Without Code</i> .....	466
<i>Python One-Liner To Create Sketchy Hand-drawn Plots</i> .....	467
<i>70x Faster Pandas By Changing Just One Line of Code</i> .....	468
<i>An Interactive Guide To Master Pandas In One Go</i> .....	469
<i>Make Dot Notation More Powerful in Python</i> .....	470
<i>The Coolest Jupyter Notebook Hack</i> .....	471
<i>Create a Moving Bubbles Chart in Python</i> .....	472
<i>Skorch: Use Scikit-learn API on PyTorch Models</i> .....	473
<i>Reduce Memory Usage Of A Pandas DataFrame By 90%</i> .....	474
<i>An Elegant Way To Perform Shutdown Tasks in Python</i> .....	475
<i>Visualizing Google Search Trends of 2022 using Python</i> .....	476
<i>Create A Racing Bar Chart In Python</i> .....	477
<i>Speed-up Pandas Apply 5x with NumPy</i> .....	478
<i>A No-Code Online Tool To Explore and Understand Neural Networks</i> .....	479
<i>What Are Class Methods and When To Use Them?</i> .....	480
<i>Make Sklearn KMeans 20x times faster</i> .....	481
<i>Speed-up NumPy 20x with Numexpr</i> .....	482
<i>A Lesser-Known Feature of Apply Method In Pandas</i> .....	483
<i>An Elegant Way To Perform Matrix Multiplication</i> .....	484
<i>Create Pandas DataFrame from Dataclass</i> .....	485
<i>Hide Attributes While Printing A Dataclass Object</i> .....	486
<i>List : Tuple :: Set : ?</i> .....	487
<i>Difference Between Dot and Matmul in NumPy</i> .....	488
<i>Run SQL in Jupyter To Analyze A Pandas DataFrame</i> .....	489
<i>Automated Code Refactoring With Sourcery</i> .....	490
<i>__Post_init__: Add Attributes To A Dataclass Object Post Initialization</i> .....	491
<i>Simplify Your Functions With Partial Functions</i> .....	492
<i>When You Should Not Use the head() Method In Pandas</i> .....	493
<i>DotMap: A Better Alternative to Python Dictionary</i> .....	494
<i>Prevent Wild Imports With __all__ in Python</i> .....	495
<i>Three Lesser-known Tips For Reading a CSV File Using Pandas</i> .....	496
<i>The Best File Format To Store A Pandas DataFrame</i> .....	497



<i>Debugging Made Easy With PySnooper</i> .....	498
<i>Lesser-Known Feature of the Merge Method in Pandas</i> .....	499
<i>The Best Way to Use Apply() in Pandas</i> .....	500
<i>Deep Learning Network Debugging Made Easy</i> .....	501
<i>Don't Print NumPy Arrays! Use Lovely-NumPy Instead.</i> .....	502
<i>Performance Comparison of Python 3.11 and Python 3.10</i> .....	503
<i>View Documentation in Jupyter Notebook</i> .....	504
<i>A No-code Tool To Understand Your Data Quickly</i> .....	505
<i>Why 256 is 256 But 257 is not 257?</i> .....	506
<i>Make a Class Object Behave Like a Function</i> .....	508
<i>Lesser-known feature of Pickle Files</i> .....	510
<i>Dot Plot: A Potential Alternative to Bar Plot</i> .....	512
<i>Why Correlation (and Other Statistics) Can Be Misleading</i> .....	513
<i>Supercharge value_counts() Method in Pandas With Sideload</i> .....	514
<i>Write Your Own Flavor Of Pandas</i> .....	515
<i>CodeSquire: The AI Coding Assistant You Should Use Over GitHub Copilot</i> .....	516
<i>Vectorization Does Not Always Guarantee Better Performance</i> .....	517
<i>In Defense of Match-case Statements in Python</i> .....	518
<i>Enrich Your Notebook With Interactive Controls</i> .....	520
<i>Get Notified When Jupyter Cell Has Executed</i> .....	522
<i>Data Analysis Using No-Code Pandas In Jupyter</i> .....	523
<i>Using Dictionaries In Place of If-conditions</i> .....	524
<i>Clear Cell Output In Jupyter Notebook During Run-time</i> .....	526
<i>A Hidden Feature of Describe Method In Pandas</i> .....	527
<i>Use Slotted Class To Improve Your Python Code</i> .....	528
<i>Stop Analysing Raw Tables. Use Styling Instead!</i> .....	529
<i>Explore CSV Data Right From The Terminal</i> .....	530
<i>Generate Your Own Fake Data In Seconds</i> .....	531
<i>Import Your Python Package as a Module</i> .....	532
<i>Specify Loops and Runs In %%timeit</i> .....	533
<i>Waterfall Charts: A Better Alternative to Line/Bar Plot</i> .....	534
<i>Hexbin Plots As A Richer Alternative to Scatter Plots</i> .....	535
<i>Importing Modules Made Easy with Pyforest</i> .....	536



<i>Analyse Flow Data With Sankey Diagrams</i> .....	538
<i>Feature Tracking Made Simple In Sklearn Transformers</i> .....	540
<i>Lesser-known Feature of f-strings in Python</i> .....	542
<i>Don't Use time.time() To Measure Execution Time</i> .....	543
<i>Now You Can Use DALL-E With OpenAI API</i> .....	544
<i>Polynomial Linear Regression Plot Made Easy With Seaborn</i> .....	545
<i>Retrieve Previously Computed Output In Jupyter Notebook</i> .....	546
<i>Parallelize Pandas Apply() With Swifter</i> .....	547
<i>Create DataFrame Hassle-free By Using Clipboard</i> .....	548
<i>Run Python Project Directory As A Script</i> .....	549
<i>Inspect Program Flow with IceCream</i> .....	550
<i>Don't Create Conditional Columns in Pandas with Apply</i> .....	551
<i>Pretty Plotting With Pandas</i> .....	552
<i>Build Baseline Models Effortlessly With Sklearn</i> .....	553
<i>Fine-grained Error Tracking With Python 3.11</i> .....	554
<i>Find Your Code Hiding In Some Jupyter Notebook With Ease</i> .....	555
<i>Restart the Kernel Without Losing Variables</i> .....	556
<i>How to Read Multiple CSV Files Efficiently</i> .....	557
<i>Elegantly Plot the Decision Boundary of a Classifier</i> .....	559
<i>An Elegant Way to Import Metrics From Sklearn</i> .....	560
<i>Configure Sklearn To Output Pandas DataFrame</i> .....	561
<i>Display Progress Bar With Apply() in Pandas</i> .....	562
<i>Modify a Function During Run-time</i> .....	563
<i>Regression Plot Made Easy with Plotly</i> .....	564
<i>Polynomial Linear Regression with NumPy</i> .....	565
<i>Alter the Datatype of Multiple Columns at Once</i> .....	566
<i>Datatype For Handling Missing Valued Columns in Pandas</i> .....	567
<i>Parallelize Pandas with Pandarallel</i> .....	568
<i>Why you should not dump DataFrames to a CSV</i> .....	569
<i>Save Memory with Python Generators</i> .....	571
<i>Don't use print() to debug your code.</i> .....	572
<i>Find Unused Python Code With Ease</i> .....	574
<i>Define the Correct DataType for Categorical Columns</i> .....	575



<b><i>Transfer Variables Between Jupyter Notebooks</i></b> .....	<b>576</b>
<b><i>Why You Should Not Read CSVs with Pandas</i></b> .....	<b>577</b>
<b><i>Modify Python Code During Run-Time</i></b> .....	<b>578</b>
<b><i>Handle Missing Data With Missingno</i></b> .....	<b>579</b>



# The Must-Know Categorisation of Discriminative Models

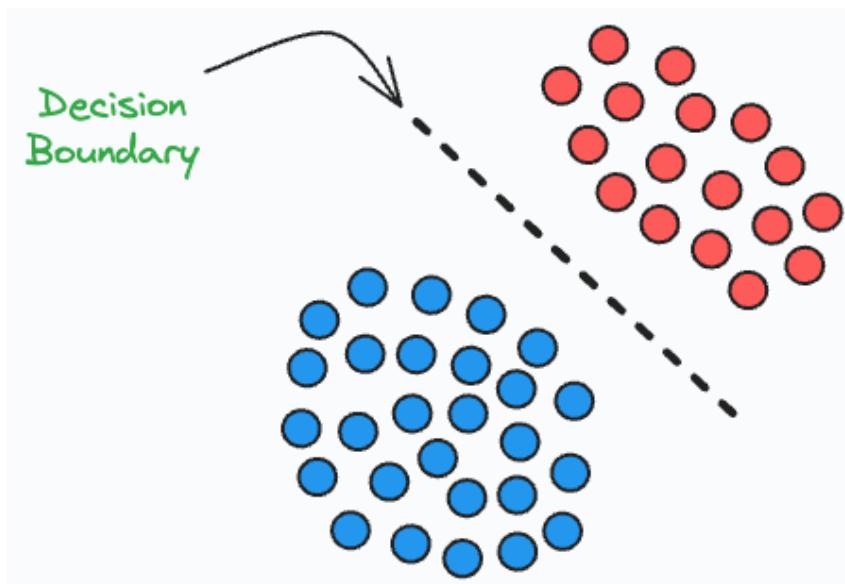
In one of the earlier posts, we discussed [Generative and Discriminative Models.](#)

Today's post dives into a further categorization of **discriminative models**.

Let's understand.

To recap:

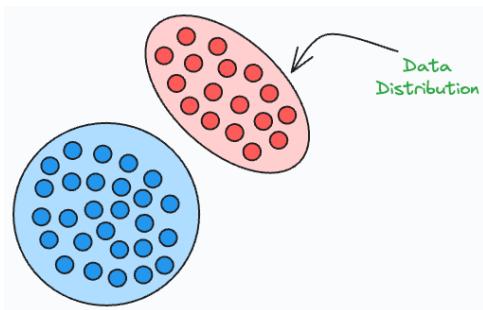
**Discriminative models:**



- learn decision boundaries that separate different classes.
- maximize the conditional probability:  $P(Y|X)$  — Given an input  $X$ , maximize the probability of label  $Y$ .
- are meant explicitly for classification tasks.



## Generative models:

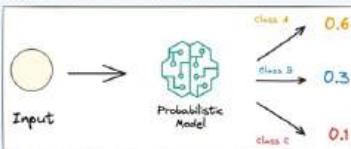


- maximize the joint probability:  $P(X, Y)$
- learn the class-conditional distribution  $P(X|Y)$
- are **typically** not meant for classification tasks, but they can perform classification nonetheless.

In a gist, discriminative models directly learn the function  $f$  that maps an input vector ( $x$ ) to a label ( $y$ ).

### Types of Discriminative Models

#### Probabilistic Models



They provide a probabilistic estimate for each class

Predictions depict model's confidence

They learn  $f(x)$  using posterior class probabilities  $P(Y|X)$

Well-suited when uncertainty is crucial

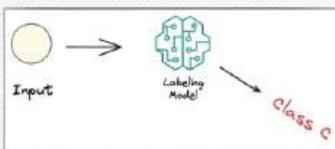
Logistic Regression

Neural Networks

CRFs

?

#### Labeling Models



They directly provide a class prediction

Predictions **DO NOT** indicate a degree of confidence

They learn  $f(x)$  using measures like similarity, rules, etc.

Unsuitable when uncertainty is crucial

Random Forests

kNN

Decision Tree

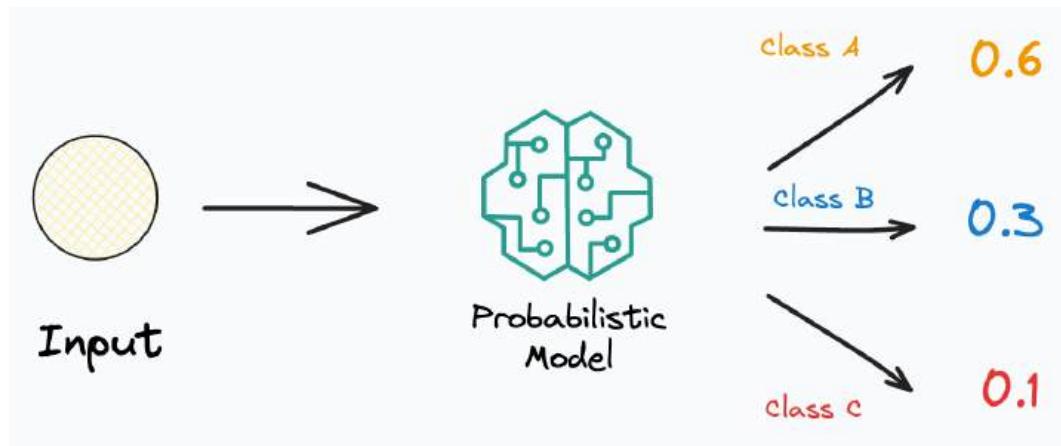
?



They can be further divided into two categories:

- Probabilistic models
- Direct labeling models

## Probabilistic models



As the name suggests, probabilistic models provide a probabilistic estimate for each class.

They do this by learning the posterior class probabilities  $P(Y|X)$ .

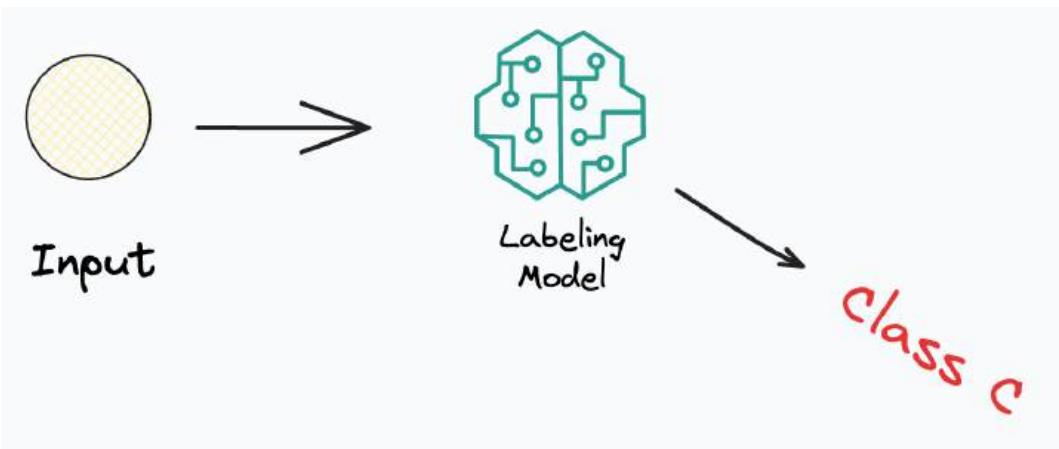
As a result, their predictions depict the model's confidence in predicting a specific class label.

This makes them well-suited in situations when uncertainty is crucial to the problem at hand.

Examples include:

- Logistic regression
- Neural networks
- CRFs

## Labeling models



### Labeling models

In contrast to probabilistic models, labeling models (also called distribution-free classifiers) directly predict the class label — without providing any probabilistic estimate.

As a result, their predictions DO NOT indicate a degree of confidence.

This makes them unsuitable when uncertainty in a model's prediction is crucial.

Examples include:

- Random forests
- kNN
- Decision trees

That being said, it is important to note that these models, in some way, can be manipulated to output a probability.

For instance, Sklearn's decision tree classifier does provide a `predict_proba()` method, as shown below:



`predict_proba(X, check_input=True)`

[source]

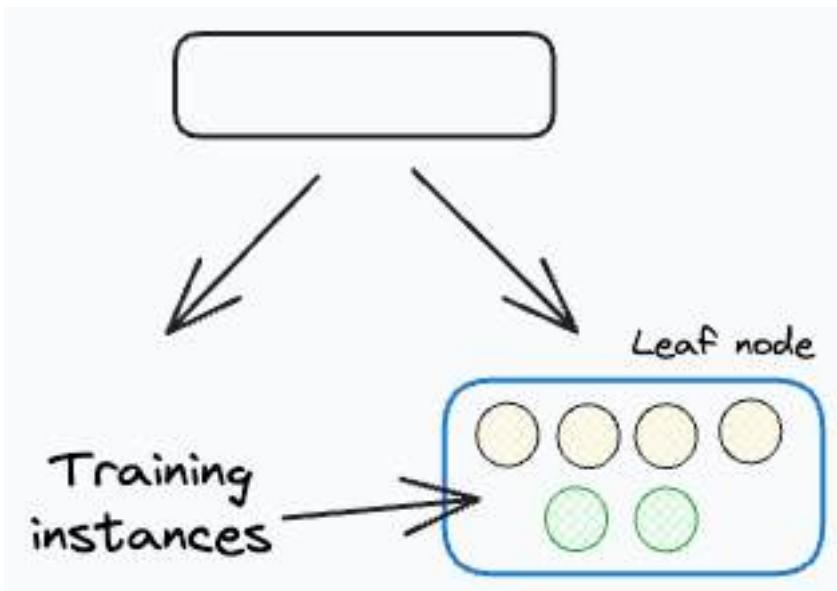
Predict class probabilities of the input samples X.

The predicted class probability is the fraction of samples of the same class in a leaf.

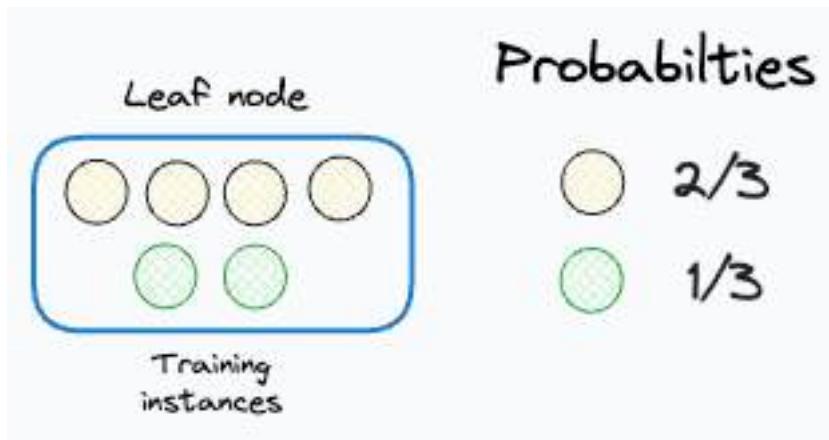
<b>Parameters:</b>	<b>X : {array-like, sparse matrix} of shape (n_samples, n_features)</b> The input samples. Internally, it will be converted to <code>dtype=np.float32</code> and if a sparse matrix is provided to a sparse <code>csr_matrix</code> .
<b>check_input : bool, default=True</b>	Allow to bypass several input checking. Don't use this parameter unless you know what you're doing.
<b>Returns:</b>	<b>proba : ndarray of shape (n_samples, n_classes) or list of n_outputs such arrays if n_outputs &gt; 1</b> The class probabilities of the input samples. The order of the classes corresponds to that in the attribute <code>classes_</code> .

This may appear a bit counterintuitive at first.

In this case, the model outputs the class probabilities by looking at the fraction of **training class labels** in a leaf node.



In other words, say a test instance reaches a specific leaf node for final classification. The model will calculate the probabilities as the fraction of **training class labels** in that leaf node.



Yet, these manipulations do not account for the “true” uncertainty in a prediction.

This is because the uncertainty is the same for all predictions that land in the same leaf node.

Therefore, it is always wise to choose probabilistic classifiers when uncertainty is paramount.

👉 Over to you: Can you add one more model for probabilistic and labeling models?

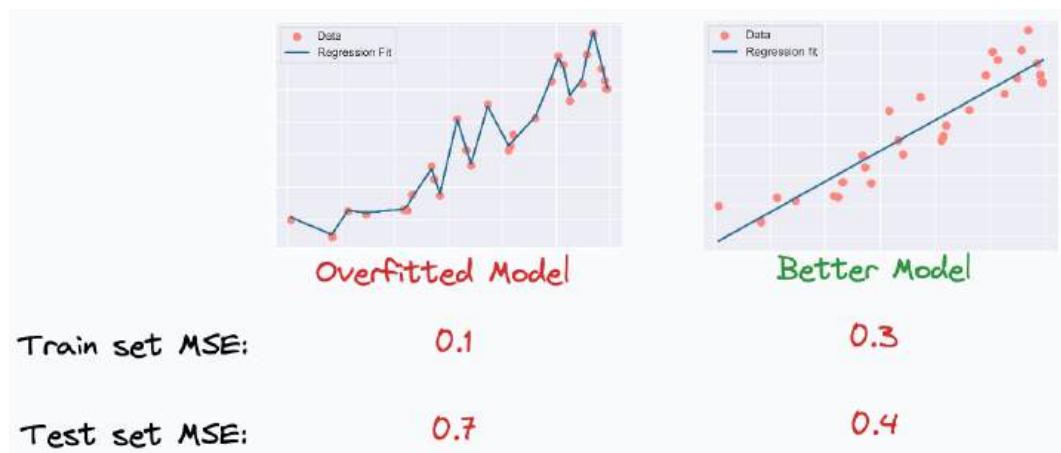


# Where Did The Regularization Term Originate From?

One of the major aspects of training any reliable ML model is avoiding **overfitting**.

In a gist, overfitting occurs when a model learns to perform exceptionally well on the training data.

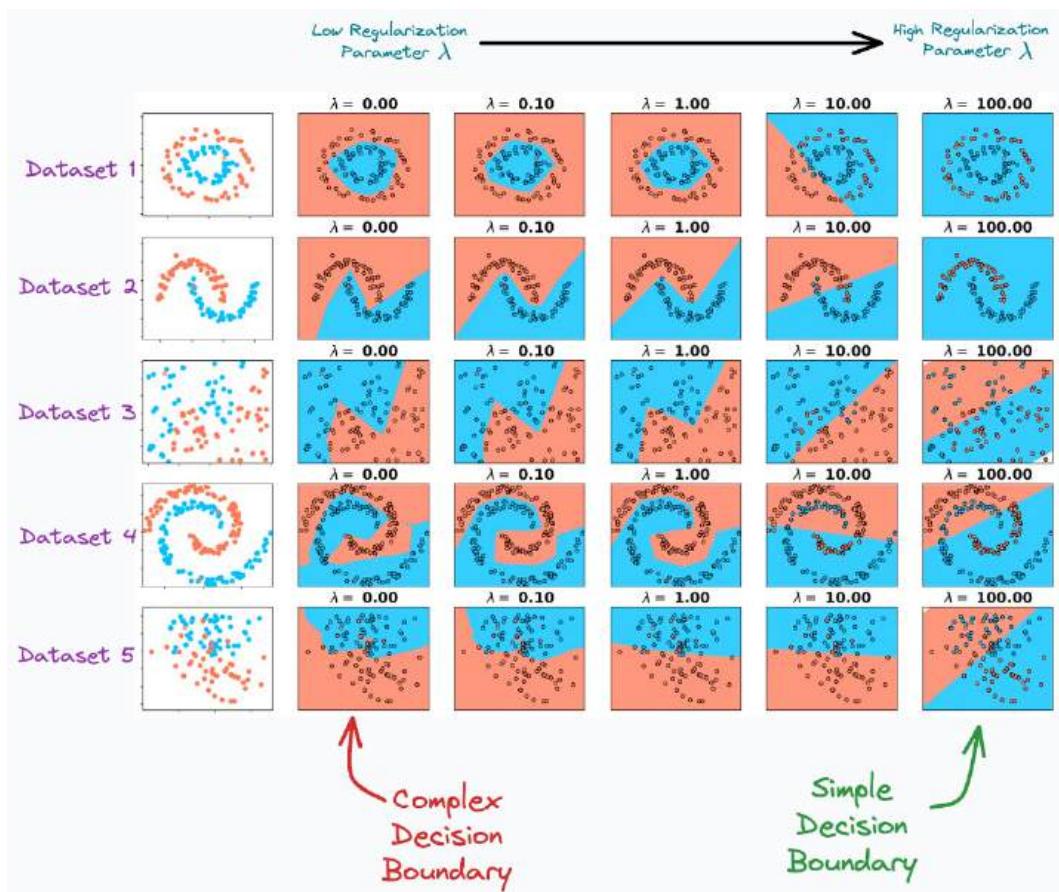
This may happen because the model is trying too hard to capture all **unrelated and random noise** in our training dataset, as shown below:



And one of the most common techniques to avoid overfitting is **regularization**.

Simply put, the core objective of regularization is to penalize the model for its complexity.

In fact, we can indeed validate the effectiveness of regularization experimentally, as shown below:



As we move to the right, the regularization parameter increases. As a result, the model creates a simpler decision boundary on all 5 datasets.

Now, if you have taken any ML course or read any tutorials about this, the most common they teach is to add a penalty (or regularization) term to the cost function, as shown below:

Loss function with L2 regularization

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m \left( y^{(i)} - \hat{y}^{(i)} \right)^2 + \lambda \sum_{j=1}^K \|\theta_j\|^2$$

Penalty term

But why?



In other words, have you ever wondered why we are taught to add a squared term to the loss function (when using L2 regularization)?

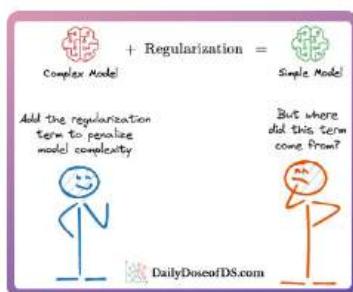
In my experience, most tutorials never bother to cover it, and readers are always expected to embrace these notions as a given.

Yet, there are many questions to ask here:

- Where did this regularization term originate from? How was it derived for the first time?
- What does the regularization term precisely measure?
- Why do we add this regularization term to the loss?
- Why do we square the parameters (specific to L2 regularization)? Why not any other power?
- Is there any probabilistic evidence that justifies the effectiveness of regularization?

Turns out, there is a concrete probabilistic justification for using regularization.

And if you are curious, then this is precisely the topic of today's machine learning deep dive: "[\*\*The Probabilistic Origin of Regularization.\*\*](#)"



Machine Learning Aug 19, 2023

## [\*\*The Probabilistic Origin of Regularization\*\*](#)

Where did the regularization term come from?

 Avi Chawla

## [\*\*The Probabilistic Origin of Regularization\*\*](#)

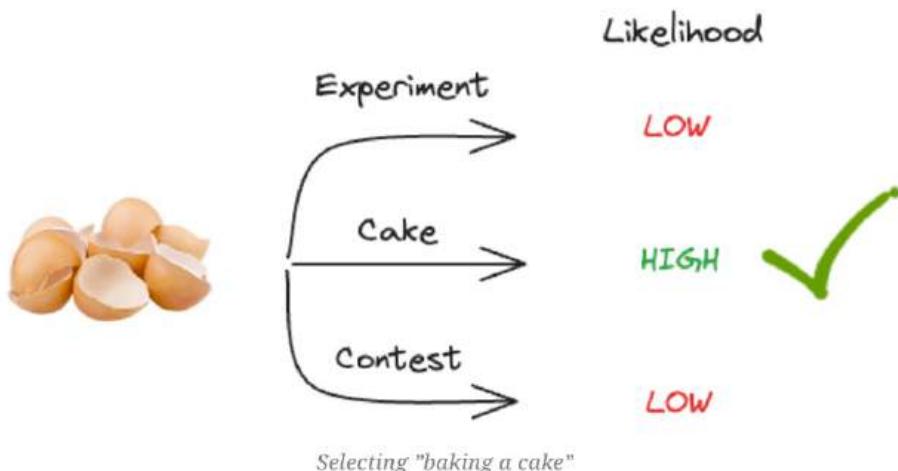
While most of the community appreciates the importance of regularization, in my experience, very few learn about its origin and the mathematical formulation behind it.



It can't just appear out of nowhere, can it?

Thus, the objective of this deep dive is to help you build a solid intuitive, and logical understanding of regularisation — **purely from a probabilistic perspective**.

Thus, even though it's more likely to have generated the evidence, it's less likely to have happened in the first place. This means we should still declare “baking a cake” as the more likely event.



But what makes us believe that baking a cake is still more likely to have produced the evidence?

It's regularization.

Image taken from the [The Probabilistic Origin of Regularization](#) article

👉 Interested folks can read it here: [The Probabilistic Origin of Regularization](#).



# How to Create The Elegant Moving Bubbles Chart in Python?

I often come across the moving bubbles chart when I am scrolling LinkedIn.

I am sure you would have seen them too.

It is elegant animation that depicts the movements of entities across time. They are particularly useful for determining when clusters appear in the data and at what state(s).

I always wondered how one can create them in Python.

Turns out, there's a pretty simple way to do it just three lines of Python using [D3Blocks](#).

The library utilizes the graphics of the popular **d3js Javascript library** to create visually appealing charts with only a few lines of Python code.

To create a moving bubbles chart, you can use the `d3.movingbubbles()` method.

The input should be a Pandas DataFrame. Each row should represent the state of a sample at a particular timestamp, as depicted below:

	Sample ID	Timestamp	Sample's State
0	1	00:00	Sleeping
1	1	01:00	Travel
2	2	00:00	Sleeping
3	2	01:00	Sleeping
⋮			
N-1	100	00:00	Eating
N	100	01:00	Work



After aligning the DataFrame in the desired format, you can create the moving bubbles chart as follows:

```
from d3blocks import D3Blocks

d3 = D3Blocks()

d3.movingbubbles(df,
                  datetime = "Timestamp",
                  sample_id = "Sample ID",
                  state = "Sample State",
                  filepath = "moving.html")
```

This will create an HTML file. You can preview it in a browser or open it in Jupyter directly using the IPython library.

Isn't that cool?



# Gradient Checkpointing: Save 50-60% Memory When Training a Neural Network

Neural networks primarily use memory in two ways:

- Storing model weights
- During training:
  - Forward pass to compute and store activations of all layers
  - Backward pass to compute gradients at each layer

This restricts us from training larger models and also limits the max batch size that can potentially fit into memory.

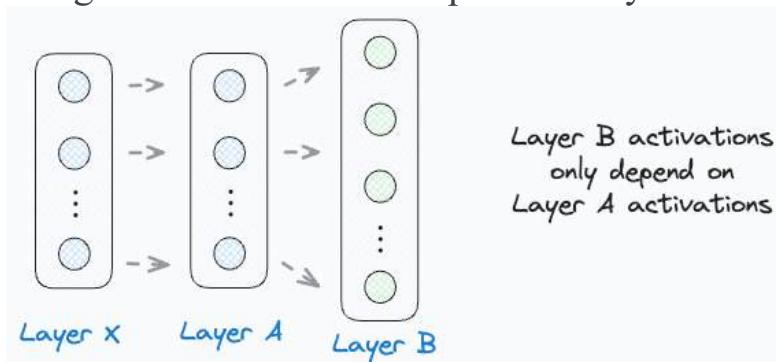
**Gradient checkpointing** is an incredible technique to reduce the memory overheads of neural nets.

Here, we run the forward pass normally and the core idea is to optimize the backpropagation step.

Let's understand how it works.

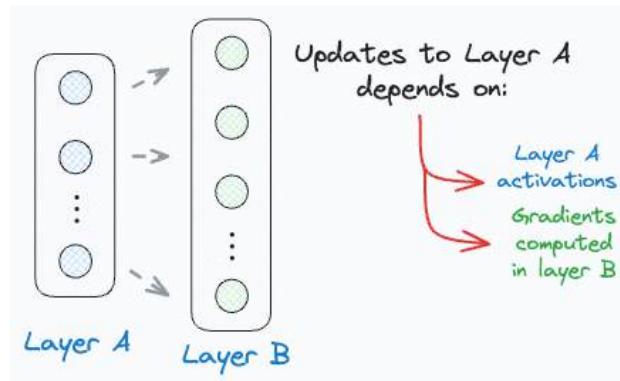
We know that in a neural network:

- The activations of a specific layer can be solely computed using the activations of the previous layer.





- Updating the weights of a layer only depends on two things:



- The activations of that layer.
- The gradients computed in the next (right) layer.

Gradient checkpointing exploits these ideas to optimize backpropagation:

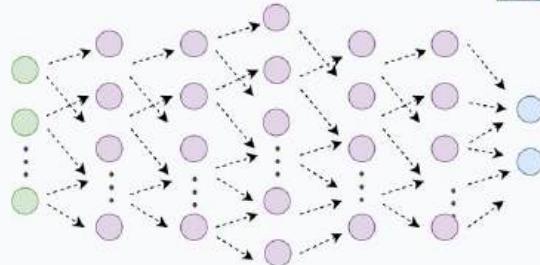
- Divide the network into segments before backpropagation
- In each segment:
  - Only store the activations of the first layer.
  - Discard the rest of the activations.
- When updating the weights of layers in a segment, recompute its activations using the first layer in that segment.

This is depicted in the image below:



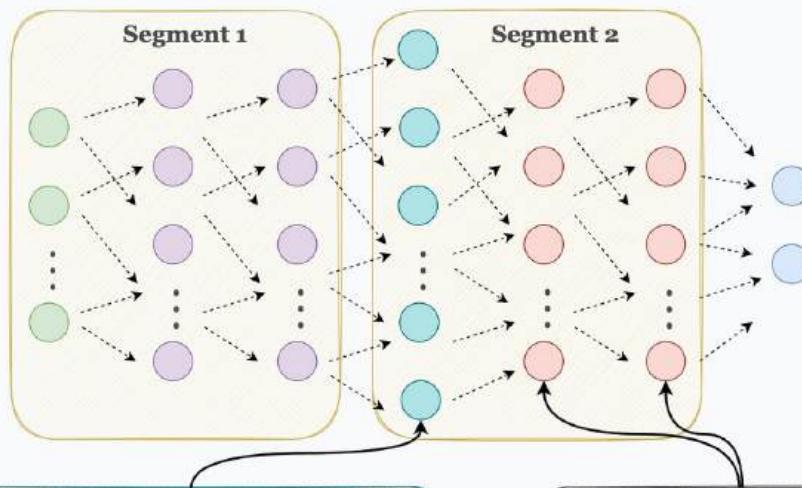
## Gradient Checkpointing: Save 50-60% Memory When Training a Neural Net

**Step 1) Run the forward pass normally**



blog.DailyDoseofDS.com

**Step 2) Divide the network into segments before backpropagation**



**Step 3.1) Only store the activations of first layer of each segment in memory**

**Step 3.2) Discard the activations of other layers in the segment**

Saves approx.  
**50-60% memory**

**Step 4) Recompute**  
using   
**ONLY WHEN NEEDED**

As shown above:

- First, we divide the network into 2 segments.
- Next, we only keep the activations of the first layer in each segment in memory.
- We discard the activations of other layers in the segment.



- When updating the weights of red layers, we recompute their activations using the activations of the cyan layer.

**Recompute**



**using**



Recomputing the activations only when they are needed tremendously reduces the memory requirement.

Essentially, we don't need to store all the intermediate activations in memory.

This allows us to train the network on larger batches of data.

Typically, gradient checkpointing can reduce memory usage by **50-60%**, which is massive.

Of course, this does come at a cost of slightly increased run-time. This can typically range between **15-25%**.

It is because we compute some activations twice.

So there's always a tradeoff between memory and run-time.

Yet, gradient checkpointing is an extremely powerful technique to train larger models without resorting to more intensive techniques like distributed training, for instance.

Thankfully, gradient checkpointing is also implemented by many open-source deep learning frameworks like [Pytorch](#), etc.

👉 Over to you: What are some ways you use to optimize a neural network's training?



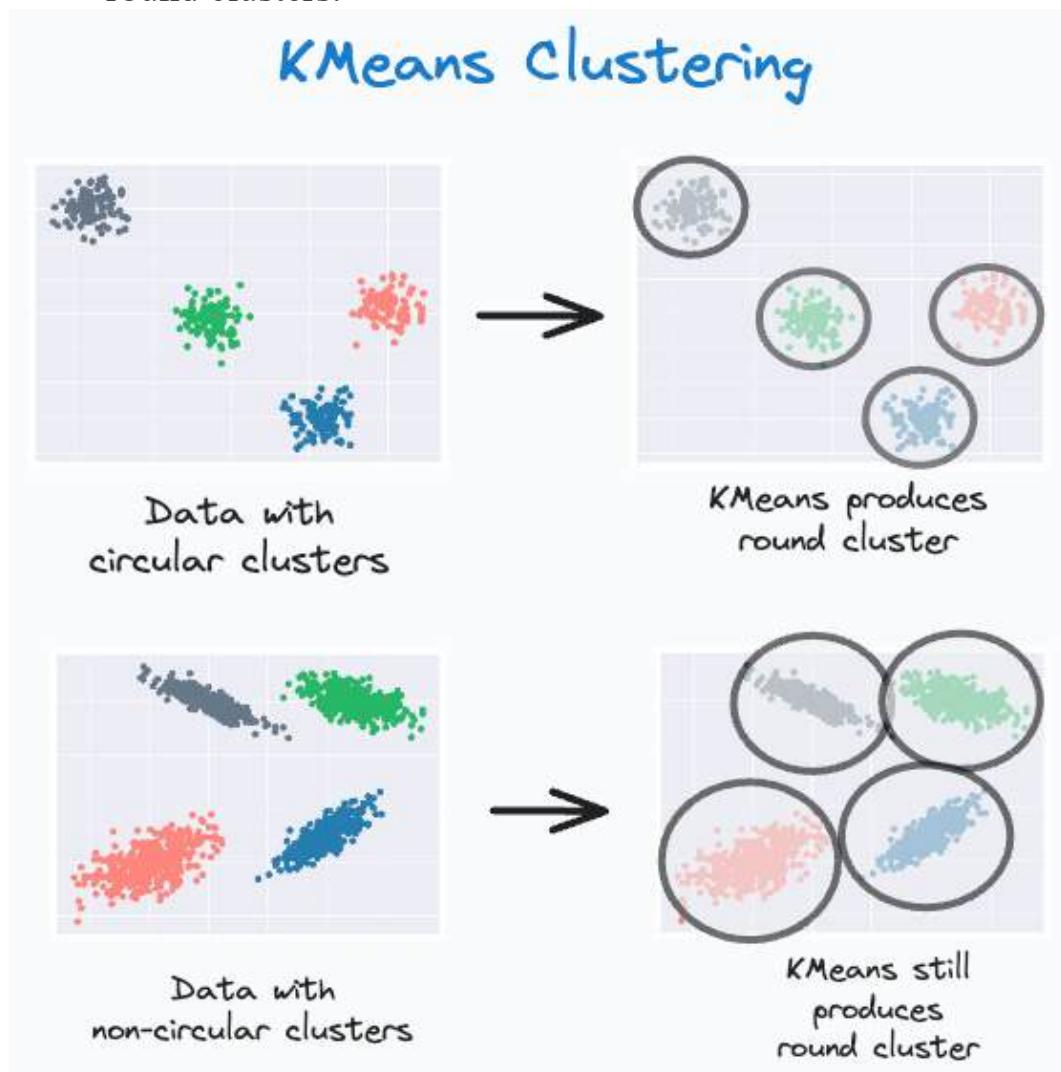
# Gaussian Mixture Models: The Flexible Twin of KMeans

KMeans is widely used for its simplicity and effectiveness as a clustering algorithm.

But it has many limitations.

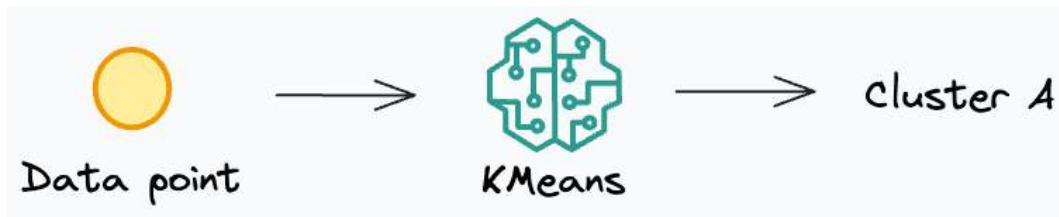
To begin:

- It does not account for cluster variance
- It can only produce spherical clusters. As shown below, even if the data has non-circular clusters, it still produces round clusters.





- It performs a hard assignment. There are no probabilistic estimates of each data point belonging to each cluster.



These limitations often make KMeans a non-ideal choice for clustering.

Gaussian Mixture Models are often a superior algorithm in this respect.

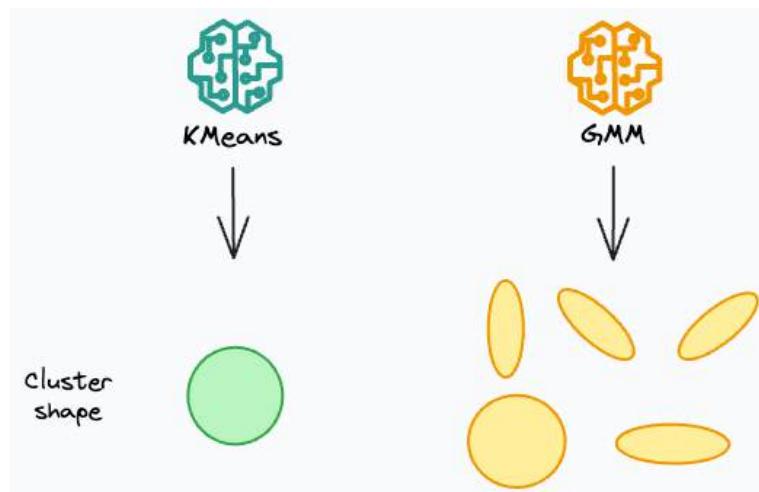
As the name suggests, they can cluster a dataset that has a mixture of many Gaussian distributions.

They can be thought of as a more flexible twin of KMeans.

The primary difference is that:

- KMeans learns **centroids**.
- Gaussian mixture models learn a **distribution**.

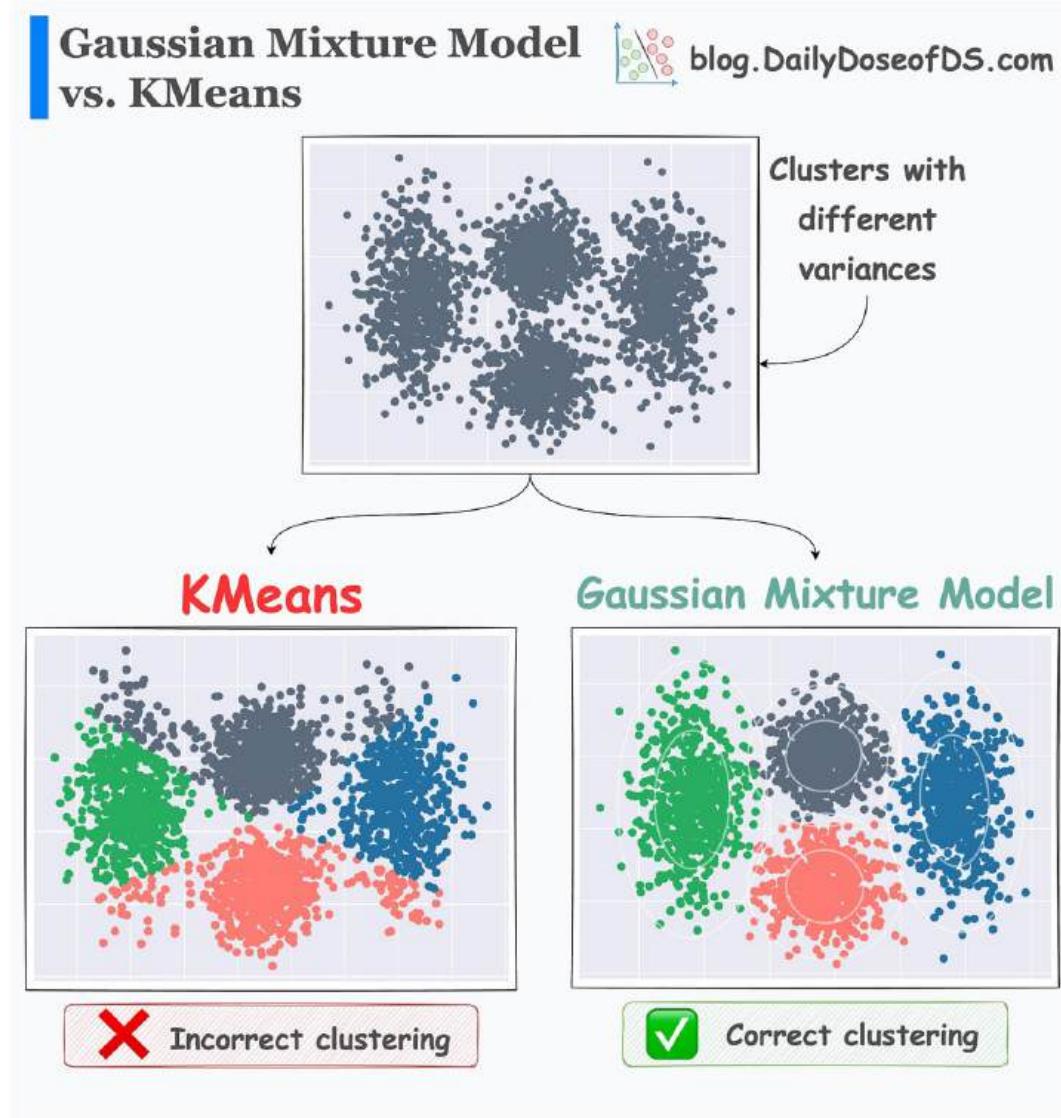
For instance, in 2 dimensions:





- KMeans can only create circular clusters
- GMM can create oval-shaped clusters.

The effectiveness of GMMs over KMeans is evident from the image below.



- KMeans just relies on distance and ignores the distribution of each cluster
- GMM learns the distribution and produces better clustering.

But how does it exactly work, and why is it so effective?



What is the core intuition behind GMMs?

How do they model the data distribution so precisely?

If you are curious, then this is precisely what we are learning in [today's extensive machine learning deep dive.](#)



## [Gaussian Mixture Models Article](#)

The entire idea and formulation of Gaussian mixture models appeared extremely compelling and intriguing to me when I first learned about them.

The notion that a single model can learn diverse data distributions is truly captivating.

Learning about them has been extremely helpful to me in building more flexible and reliable clustering algorithms.

Thus, understanding how they work end-to-end will be immensely valuable if you are looking to expand your expertise beyond traditional algorithms like KMeans, DBSCAN, etc.

Thus, today's article covers:

- The shortcomings of KMeans.
- What is the motivation behind GMMs?
- How do GMMs work?
- The intuition behind GMMs.

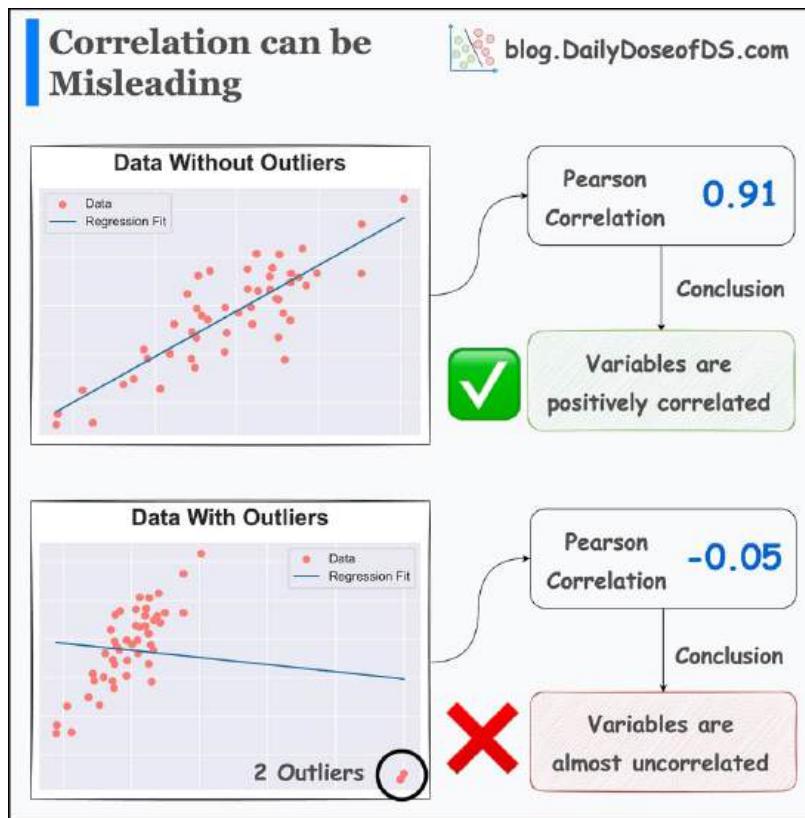


- Plotting dummy multivariate Gaussian distributions to better understand GMMs.
- The end-to-end mathematical formulation of GMMs.
- How to use Expectation-Maximization to model data using GMMs?
- **Coding a GMM from scratch (without sklearn).**
- Comparing results of GMMs with KMeans.
- How to determine the optimal number of clusters for GMMs?
- Some practical use cases of GMMs.
- Takeaways.

👉 Interested folks can read it here: [Gaussian Mixture Models \(GMM\)](#).



# Why Correlation (and Other Summary Statistics) Can Be Misleading



Many data scientists solely rely on the correlation matrix to study the association between variables.

But unknown to them, the obtained statistic can be heavily driven by outliers.

This is evident from the image above.

The addition of just two outliers drastically changed:

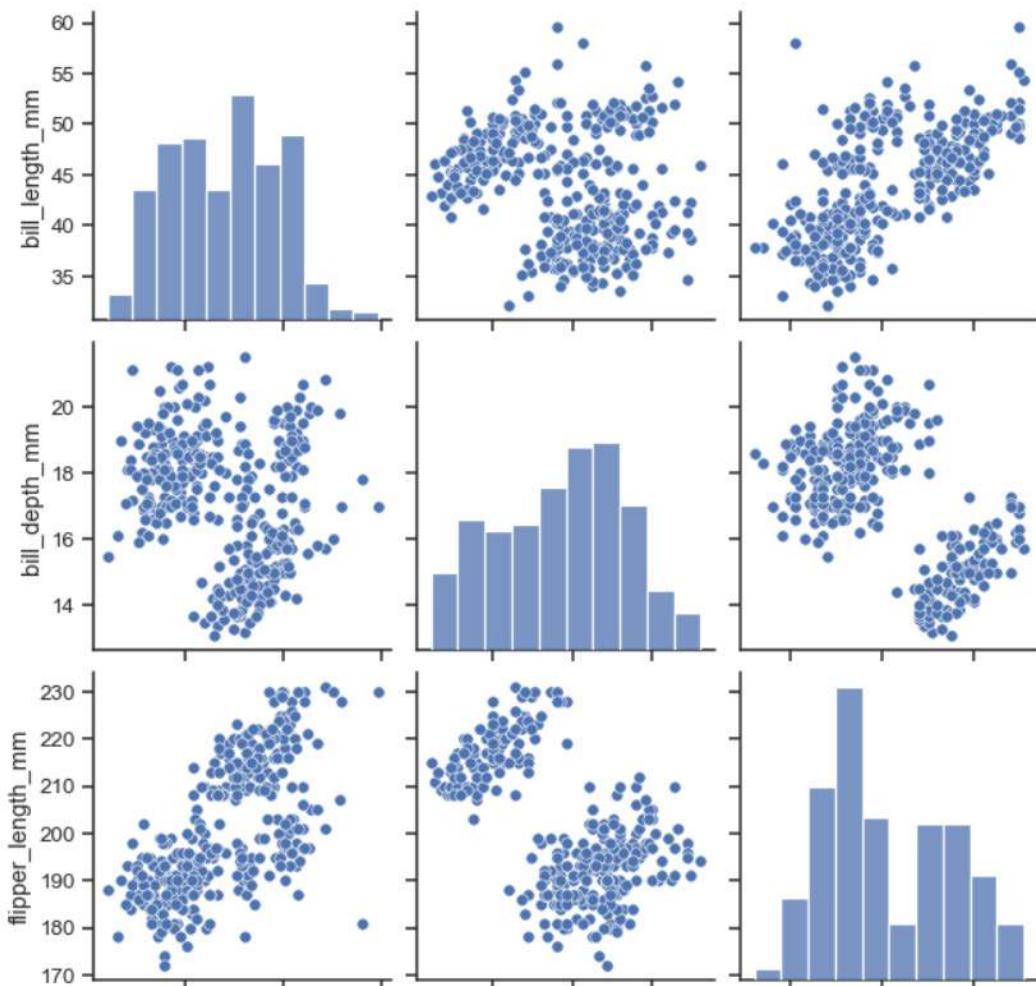
- the correlation
- the regression fit

Thus, plotting the data is highly important.



This can save you from drawing wrong conclusions, which you may have drawn otherwise by solely looking at the summary statistics.

One thing that I often do when using a correlation matrix is creating a PairPlot as well (shown below).



This lets me infer if the scatter plot of two variables and their corresponding correlation measure resonate with each other or not.

👉 Over to you: What are some other measures you take when using summary statistics?



# MissForest: A Better Alternative To Zero (or Mean) Imputation

Replacing (imputing) missing values with mean or zero or any other fixed value:

- alters summary statistics
- changes the distribution
- inflates the presence of a specific value

This can lead to:

- inaccurate modeling
- incorrect conclusions, and more.

Instead, always try to impute missing values with more precision.

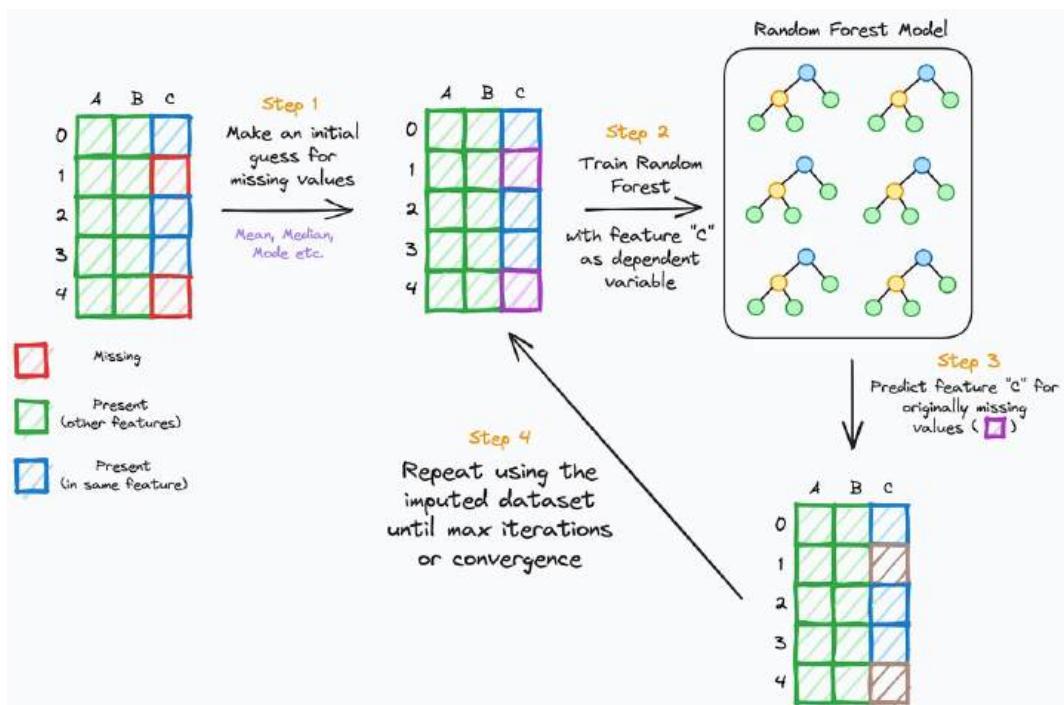
In one of the earlier posts, we discussed [kNN imputer](#). Today's post builds on that by addressing its limitations, which are:

1. High run-time for imputation — especially for high-dimensional datasets.
2. Issues with distance calculation in case of categorical non-missing features.
3. Requires feature scaling, etc.

[\*\*MissForest\*\*](#) imputer is another reliable choice for missing value imputation.

As the name suggests, it imputes missing values using the Random Forest algorithm.

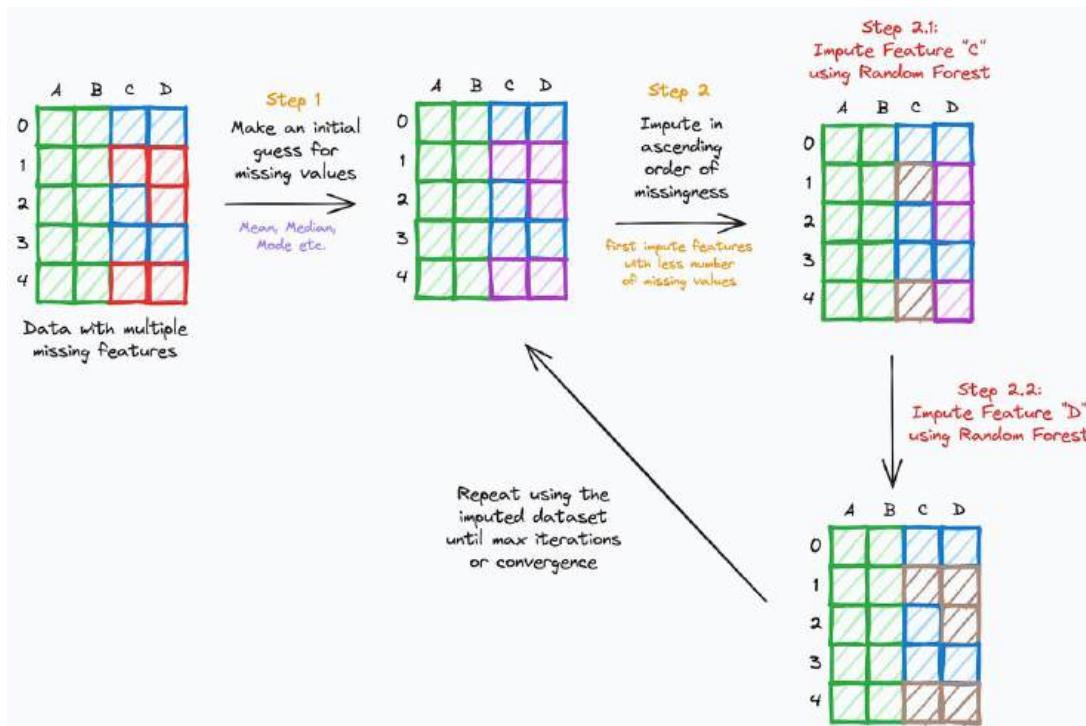
The following figure depicts how it works:



Visual illustration of MissForest imputer

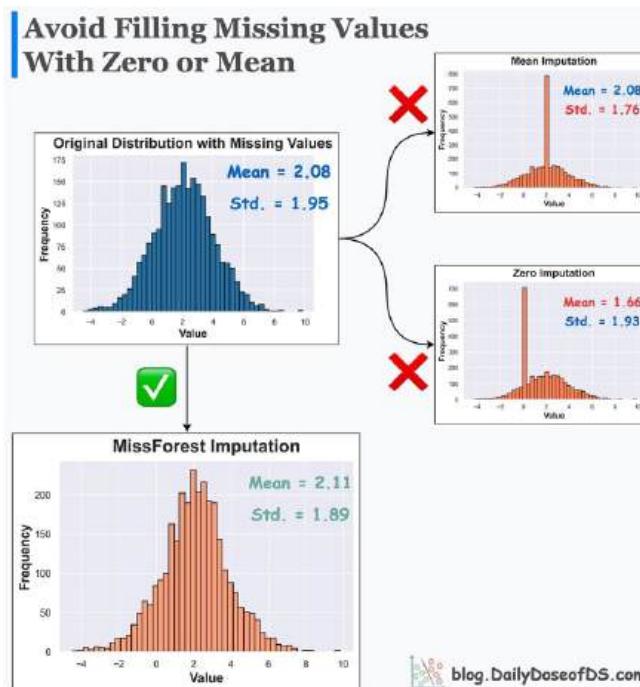
- **Step 1:** To begin, impute the missing feature with a random guess — Mean, Median, etc.
- **Step 2:** Model the missing feature using Random Forest.
- **Step 3:** Impute ONLY originally missing values using Random Forest's prediction.
- **Step 4:** Back to Step 2. Use the imputed dataset from Step 3 to train the next Random Forest model.
- **Step 5:** Repeat until convergence (or max iterations).

In case of multiple missing features, the idea (somewhat) stays the same:



- Impute features sequentially in increasing order of missingness — features with fewer missing values are imputed first.

Its effectiveness over Mean/Zero imputation is evident from the image below.





- Mean/Zero alters the summary statistics and distribution.
- MissForest imputer preserves them.

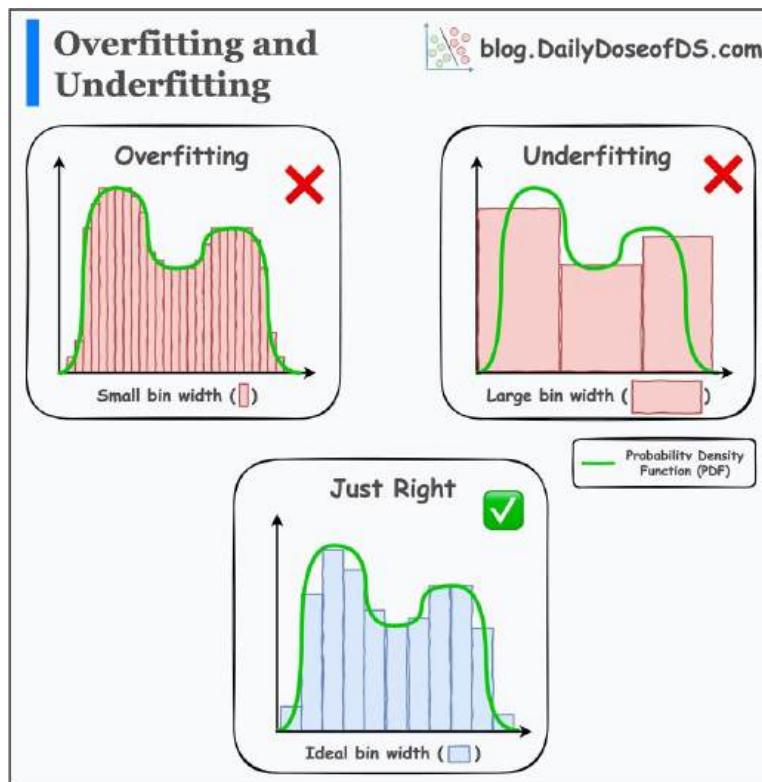
What's more, MissForest can impute even if the data has categorical non-missing features.

MissForest is based on Random Forest, so one can impute from categorical and continuous data.

Get started with MissForest imputer: [MissingPy MissForest](#).



# A Visual and Intuitive Guide to The Bias-Variance Problem



The concepts of overfitting and underfitting are pretty well understood by most folks.

Yet, here's another neat way to understand them intuitively.

Imagine you want to estimate a probability density function (PDF) using a histogram.

Your estimation entirely depends on the bin width:

- Creating small bins will overfit the PDF. This leads to high variance.
- Creating large bins will underfit the PDF. This leads to high bias.

This is depicted in the image above.



Overall, the whole bias-variance problem is about finding the optimal bin width.

I first read this analogy in the book “**All of Statistics**” a couple of years back and found it to be pretty intuitive and neat.

Here’s the book if anyone’s interested in learning more: [All of Statistics PDF](#). Page 306 inspired today’s post.

Hope that helped :)



# The Most Under-appreciated Technique To Speed-up Python

Python's default interpreter — CPython, isn't smart.

It serves as a standard interpreter for Python and offers no built-in optimization.

Instead, use the **Cython** module.

CPython and Cython are different. Don't get confused between the two.

**Cython** converts your Python code into C, which is fast and efficient.

Steps to use the Cython module:

- Load the Cython module (in a separate cell of the notebook): `%load_ext Cython`.
- Add the Cython magic command at the top of the cell: `%%cython -a`.
- When using functions, specify their parameter data type.

```
def func(int number):  
    ...  
    • Define every variable using the “cdef” keyword and  
    specify its data type.  
cdef int a = 10
```

Once done, Cython will convert your Python code to C, as depicted below:



```
In [24]: %%cython -a
def foo_c(int check):
    cdef int i,j
    cdef int count = 0

    for i in range(10000):
        for j in range(10000):
            if (i+j)%11 == check:
                count += 1

    return count

Out[24]: Generated by Cython 0.29.28
Yellow lines hint at Python interaction.
Click on a line that starts with a '+' to see the C code that Cython generated for it.

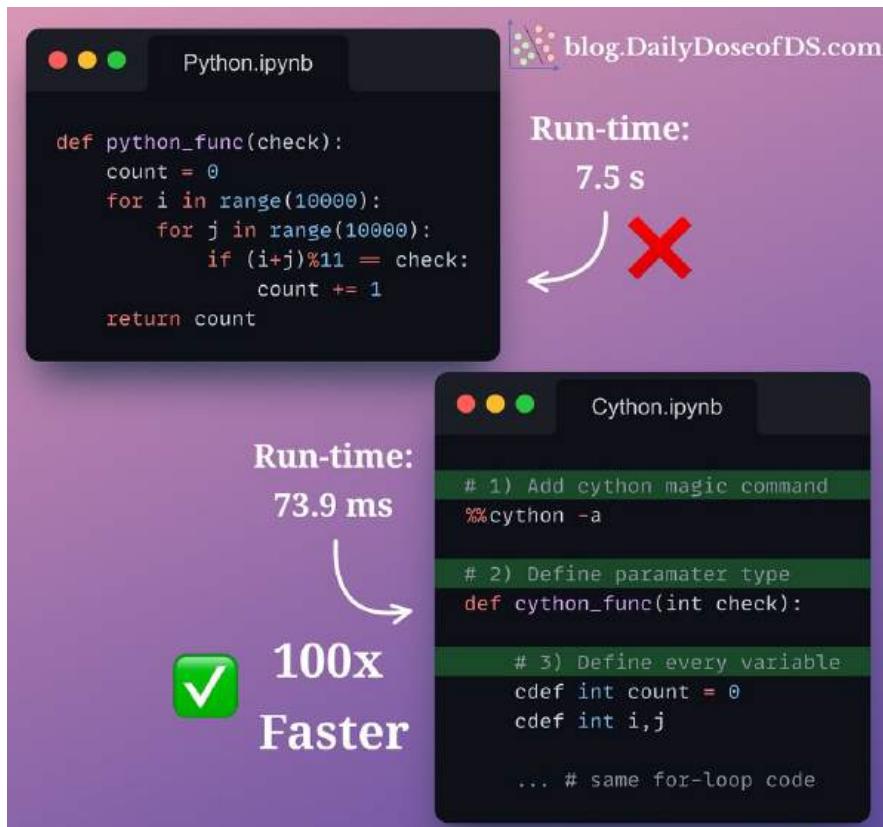
  01:
+02: def foo_c(int check):
  03:
  04:     cdef int i,j
  05:     cdef int count = 0
  06:
  07:     for i in range(10000):
  08:         for j in range(10000):
  09:             if (i+j)%11 == check:
  10:                 count += 1
  11:
+12:     return count
```

Cython converts Python code to C

This will run at native machine code speed. Just invoke the method:

```
>>> foo_c(2)
```

The speedup is evident from the image below:





- Python code is slow.
- But Cython provides a 100x speedup.

*Why does this work?*

Essentially, Python is dynamic in nature.

For instance, you can define a variable of a specific type.  
But later, you can change it to some other type.

```
a = 10  
a = "hello" # Perfectly legal in Python
```

These dynamic manipulations come at the cost of run time.  
They also introduce memory overheads.

However, Cython lets you restrict Python's dynamicity.

We avoid the above overheads by explicitly specifying the variable data type.

```
cdef int a = 10  
a = "hello" ## Raises error
```

The above declaration restricts the variable to a specific data type. This means the program would never have to worry about dynamic allocations.

This speeds up run-time and reduces memory overheads.

Isn't that cool?

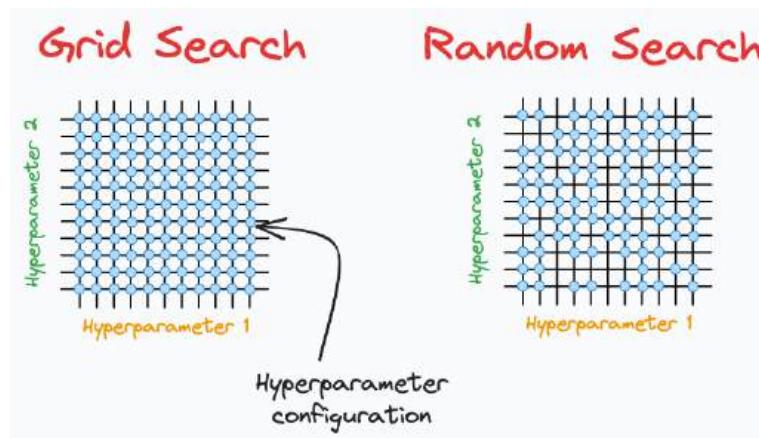


# The Overlooked Limitations of Grid Search and Random Search

Hyperparameter tuning is a tedious task in training ML models.

Typically, we use two common approaches for this:

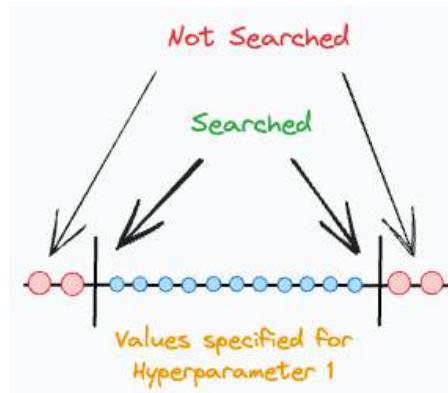
- Grid search
- Random search



But they have many limitations.

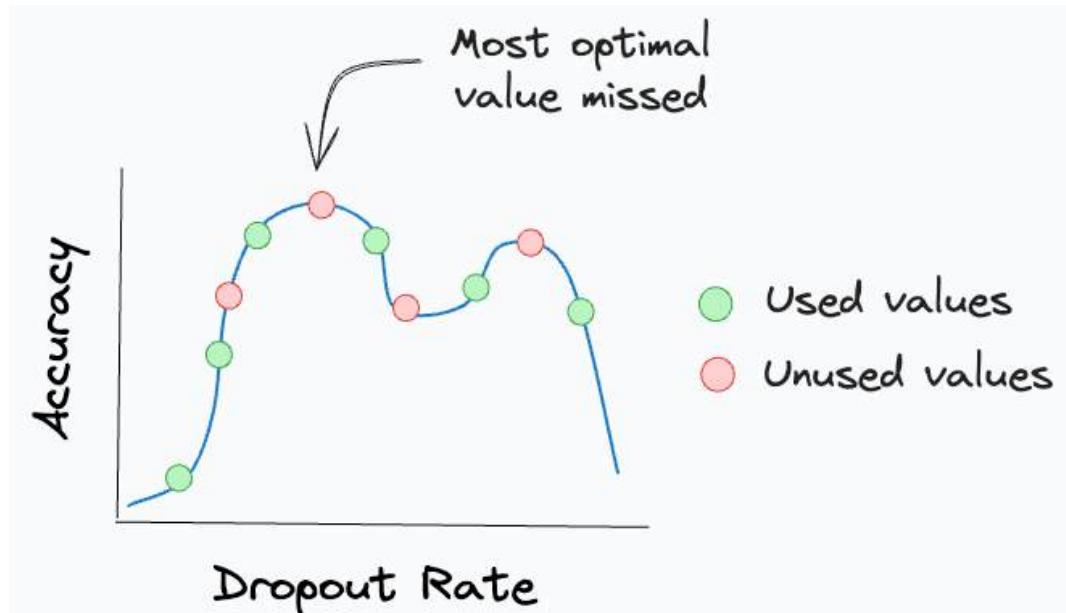
For instance:

- Grid search performs an exhaustive search over all combinations. This is computationally expensive.
- Grid search and random search are restricted to the specified hyperparameter range. Yet, the ideal hyperparameter may exist outside that range.





They can ONLY perform discrete searches, even if the hyperparameter is continuous.



Grid search and random search can only try discrete values for continuous hyperparameters

To this end, **Bayesian Optimization** is a highly underappreciated yet immensely powerful approach for tuning hyperparameters.

It uses **Bayesian statistics** to estimate the distribution of the best hyperparameters.

This allows it to take informed steps to select the next set of hyperparameters. As a result, it gradually converges to an optimal set of hyperparameters much faster.

The efficacy is evident from the image below.



## Limitation of Grid Search and Random Search

 blog.DailyDoseofDS.com

	Grid Search	Random Search	Bayesian Optimization	
Total Iterations	720	360	100	7x less iterations 
Total Run-time	242s	113s	48s	5x less run-time 
Best Trial Index	667	221	83	Optimal config. found earlier 
Best F1 score	0.93	0.93	0.93	Same score 

Bayesian optimization leads the model to the same F1 score but:

- it takes 7x fewer iterations
- it executes 5x faster
- it reaches the optimal configuration earlier

But how does it exactly work, and why is it so effective?

What is the core intuition behind Bayesian optimization?

How does it optimally reduce the search space of the hyperparameters?

If you are curious, then this is precisely what we are learning in [today's extensive machine learning deep dive.](#)

Random Search vs.  
Grid Search vs.  
Bayesian Optimization

 DailyDoseofDS.com

Machine Learning Aug 11, 2023

	Grid Search	Random Search	Bayesian Optimization	
Total Iterations	720	360	100	7x less Iterations 
Total Run-time	242s	113s	48s	5x less run-time 
Best Trial Index	667	221	83	Optimal config. found earlier 
Best F1 score	0.93	0.93	0.93	Same score 

### Bayesian Optimization for Hyperparameter Tuning

The caveats of grid search and random search and how Bayesian optimization addresses them.

 Avi Chawla

[Bayesian Optimization Article](#)



The idea behind Bayesian optimization appeared to be extremely compelling to me when I first learned it a few years back.

Learning about this optimized hyperparameter tuning and utilizing them has been extremely helpful to me in building large ML models quickly.

Thus, learning about Bayesian optimization will be immensely valuable if you envision doing the same.

Thus, today's article covers:

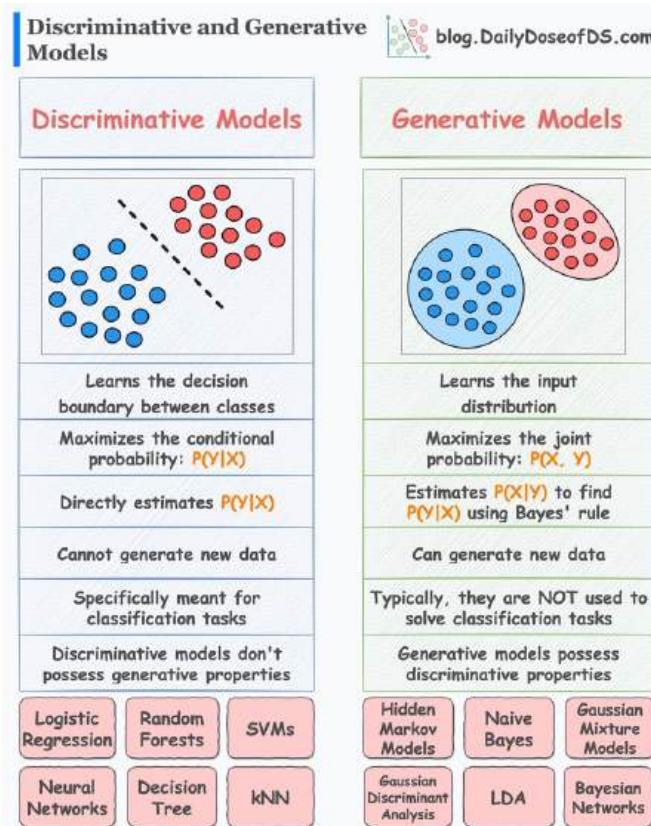
- Issues with traditional hyperparameter tuning approaches.
- What is the motivation for Bayesian optimization?
- How does Bayesian optimization work?
- The intuition behind Bayesian optimization.
- Results from the research paper that proposed Bayesian optimization for hyperparameter tuning.
- **A hands-on Bayesian optimization experiment.**
- Comparing Bayesian optimization with grid search and random search.
- Analyzing the results of Bayesian optimization.
- Best practices for using Bayesian optimization.

👉 Interested folks can read it here: [Bayesian Optimization for Hyperparameter Tuning](#).

Hope you will learn something new today :)



# An Intuitive Guide to Generative and Discriminative Models in Machine Learning



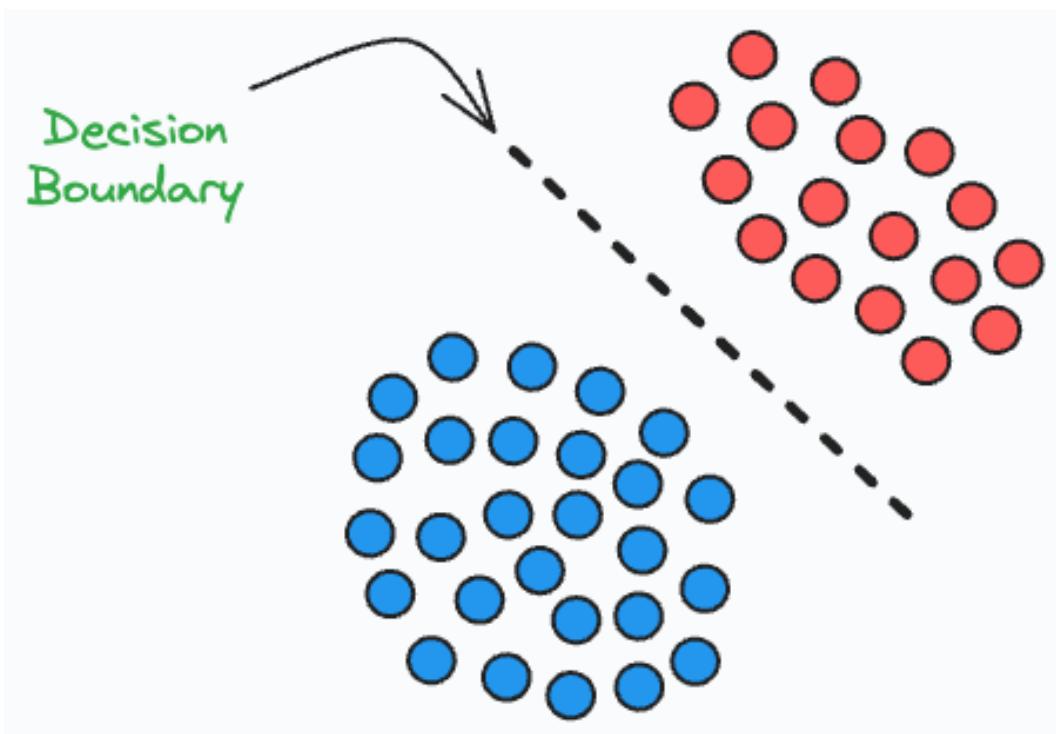
Many machine learning models can be classified into two categories:

- Generative
- Discriminative

This is depicted in the image above.

Today, let's understand what they are.

## Discriminative models



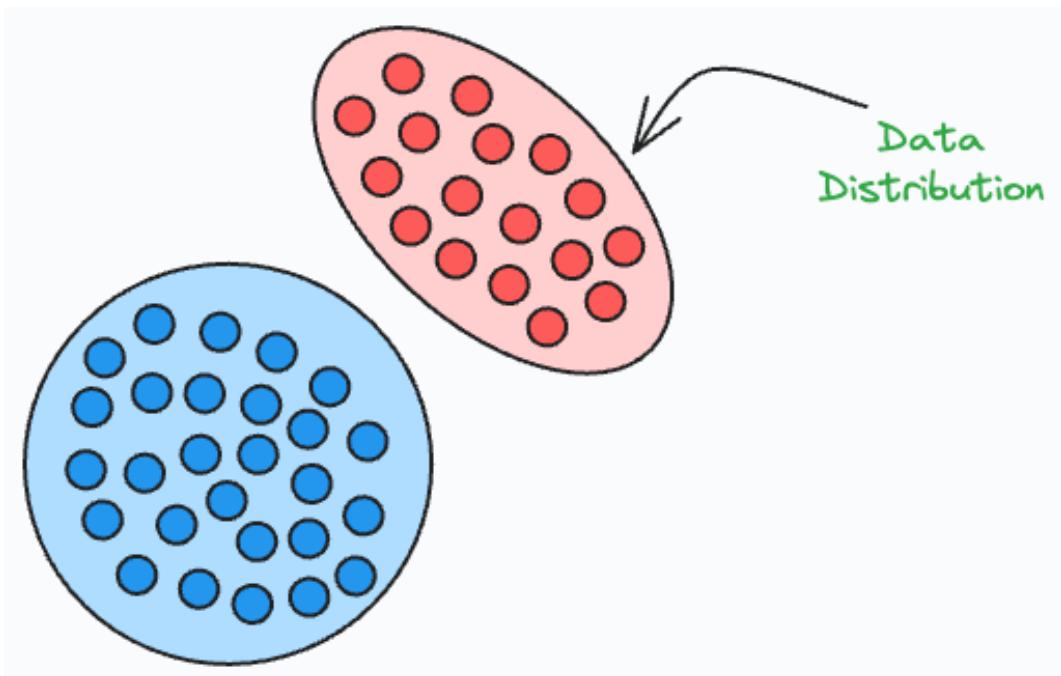
Discriminative models:

- learn decision boundaries that separate different classes.
- maximize the conditional probability:  $P(Y|X)$  — Given an input  $X$ , maximize the probability of label  $Y$ .
- are meant explicitly for classification tasks.

Examples include:

- Logistic regression
- Random Forest
- Neural Networks
- Decision Trees, etc.

### Generative models



Generative models:

- maximize the joint probability:  $P(X, Y)$
- learn the class-conditional distribution  $P(X|Y)$
- are **typically** not meant for classification tasks.

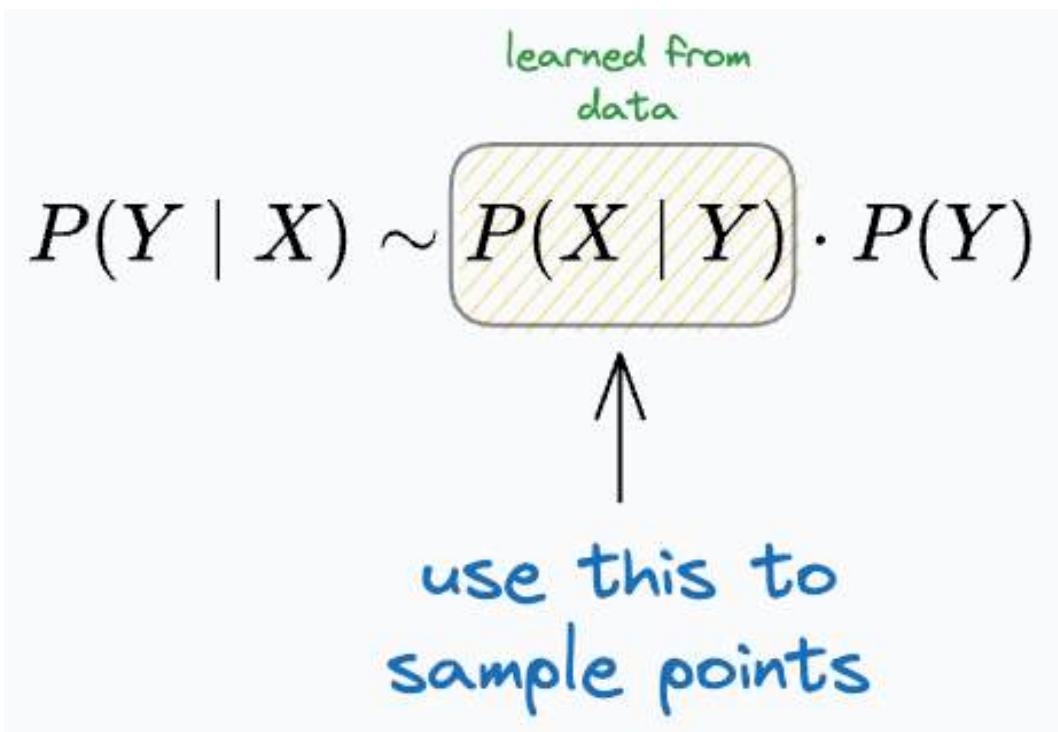
Examples include:

- Naive Bayes
- Linear Discriminant Analysis (LDA)
- Gaussian Mixture Models, etc.

We covered Joint and Conditional probability before. Read this post if you wish to learn what they are: [A Visual Guide to Joint, Marginal and Conditional Probabilities](#).

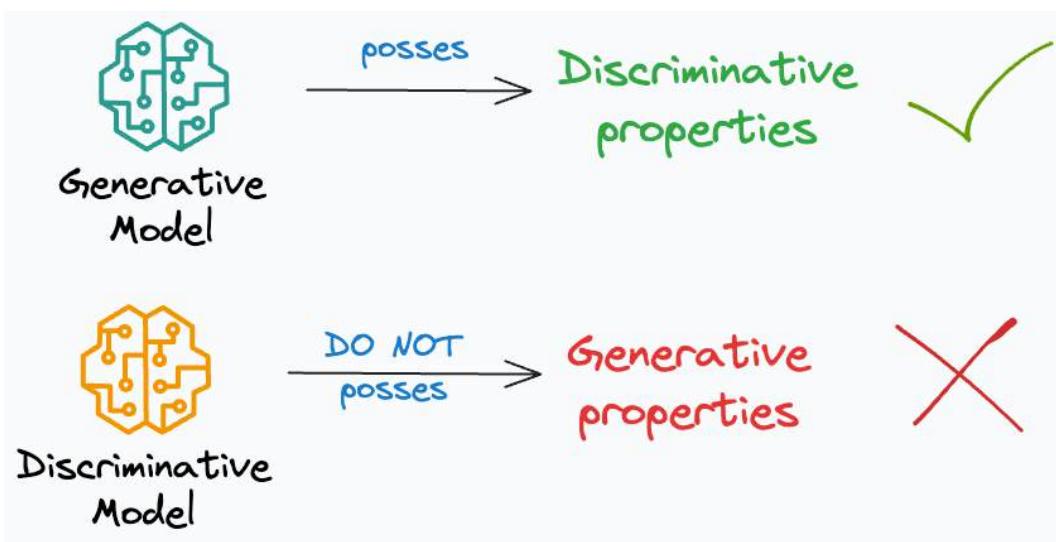
---

As generative models learn the underlying distribution, they can generate new samples.



However, this is not possible with discriminative models.

Furthermore, generative models possess discriminative properties, i.e., they can be used for classification tasks (if needed).



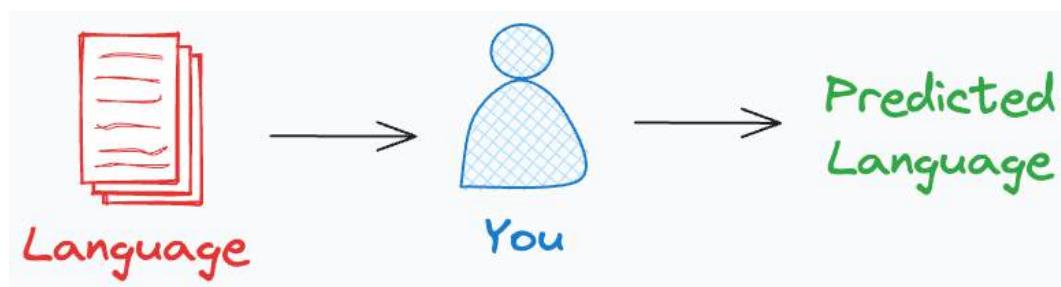


However, discriminative models do not possess generative properties.

---

Let's consider an example.

Imagine yourself as a language classification system.



There are two ways you can classify languages.

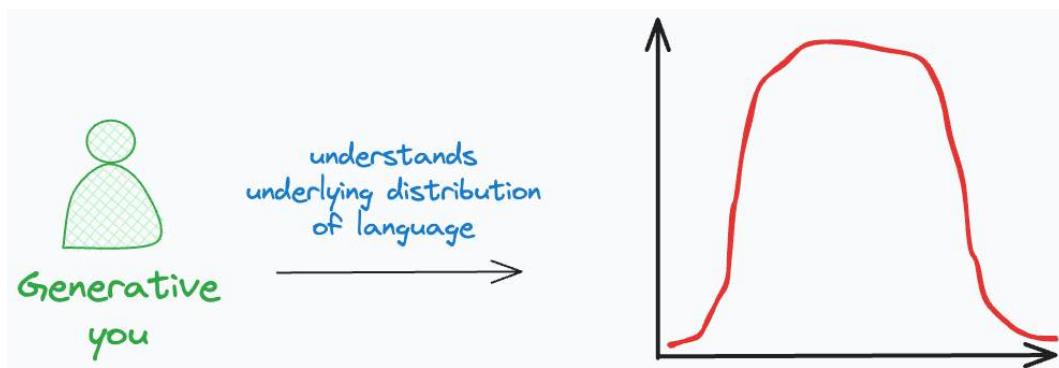
1. Learn every language and then classify a new language based on acquired knowledge.
2. Understand some distinctive patterns in each language without truly learning the language. Once done, classify a new language.

Can you figure out which of the above is generative and which one is discriminative?

---

The first approach is **generative**. This is because you have learned the underlying distribution of each language.

In other words, you learned the joint distribution  $P(\text{Words}, \text{Language})$ .

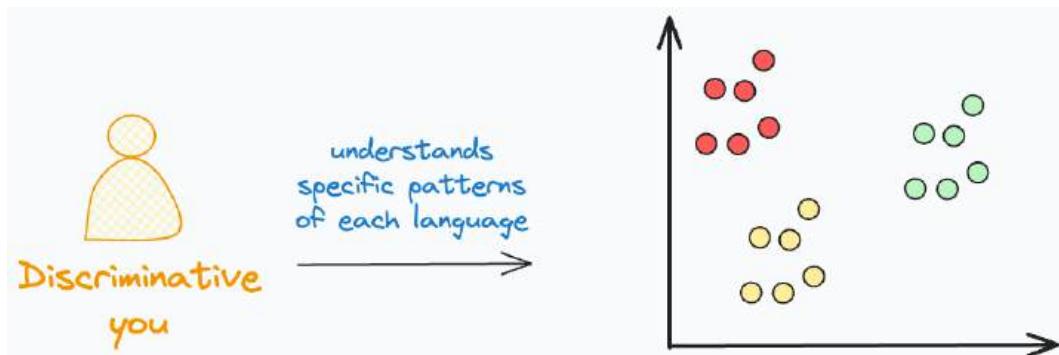


Moreover, as you understand the underlying distribution, now you can generate new sentences, can't you?

The second approach is a **discriminative approach**. This is because you only learned specific distinctive patterns of each language.

It is like:

- If so and so words appear, it is likely “Langauge A.”
- If this specific set of words appear, it is likely “Langauge B.”
- and so on.



In other words, you learned the conditional distribution  $P(\text{Language}|\text{Words})$ .

Here, can you generate new sentences now? No, right?



This is the difference between generative and discriminative models.

Also, the above description might persuade you that generative models are more generally useful, but it is not true.

This is because generative models have their own modeling complications.

For instance, typically, generative models require more data than discriminative models.

Relate it to the language classification example again.

Imagine the amount of data you would need to learn all languages (generative approach) vs. the amount of data you would need to understand some distinctive patterns (discriminative approach).

Typically, discriminative models outperform generative models in classification tasks.



# Feature Scaling is NOT Always Necessary

Feature scaling is commonly used to improve the performance and stability of ML models.

This is because it scales the data to a standard range. This prevents a specific feature from having a strong influence on the model's output.



Different scales of columns

For instance, in the image above, the scale of **Income** could massively impact the overall prediction. Scaling both features to the same range can mitigate this and improve the model's performance.

But is it always necessary?

While feature scaling is often crucial, knowing when to do it is also equally important.

Note that many ML algorithms are unaffected by scale. This is evident from the image below.



## Feature Scaling is NOT Always Necessary

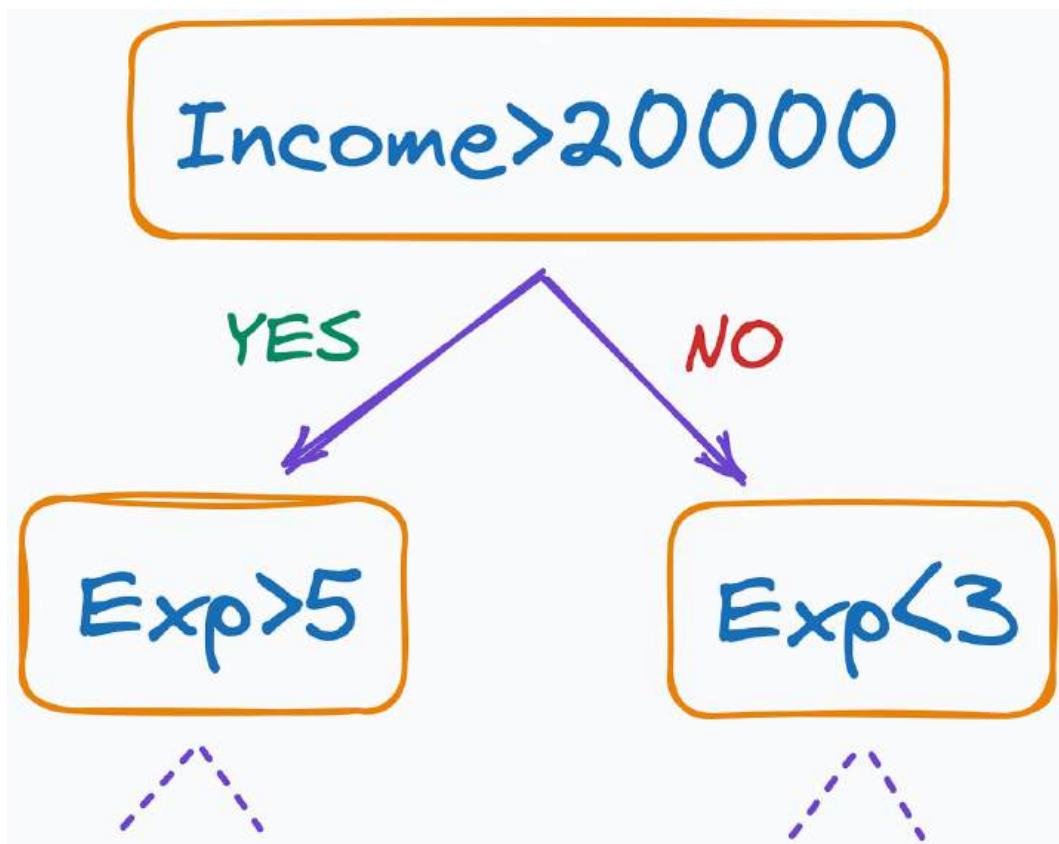


Algorithm	Test Set Classification Performance		
	Without Feature Scaling	With Feature Scaling	Scaling Required?
Logistic Regression	0.53	0.70	YES
Support Vector Classifier	0.72	0.94	YES
MLP Classifier	0.73	0.89	YES
kNN Classifier	0.66	0.93	YES
Decision Tree	0.83	0.83	NO
Random Forest	0.91	0.91	NO
Gradient Boosting	0.86	0.86	NO
Naive Bayes	0.75	0.75	NO

As shown above:

- Logistic regression, SVM Classifier, MLP, and kNN do better with feature scaling.
- Decision trees, Random forests, Naive bayes, and Gradient boosting are unaffected.

Consider a decision tree, for instance. It splits the data based on thresholds determined solely by the feature values, regardless of their scale.



Decision tree

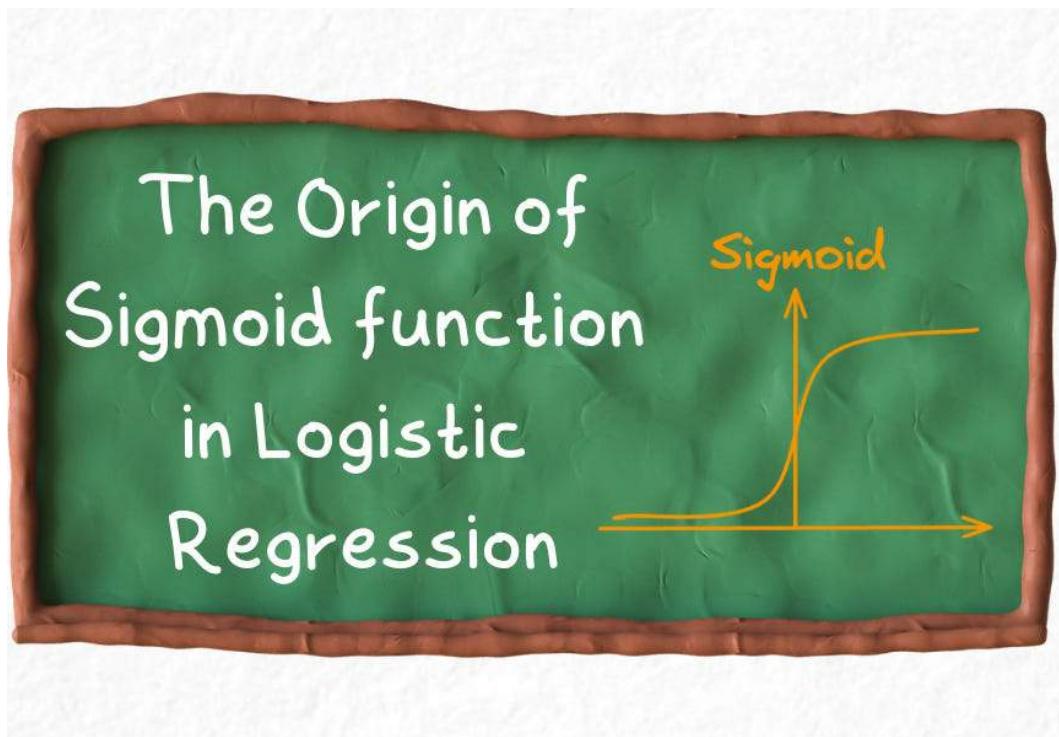
Thus, it's important to understand the nature of your data and the algorithm you intend to use.

You may never need feature scaling if the algorithm is insensitive to the scale of the data.

👉 Over to you: What other algorithms typically work well without scaling data? Let me know :)



## Why Sigmoid in Logistic Regression?



Logistic regression returns the probability of a binary outcome (0 or 1).

We all know logistic regression does this using the **sigmoid function**.

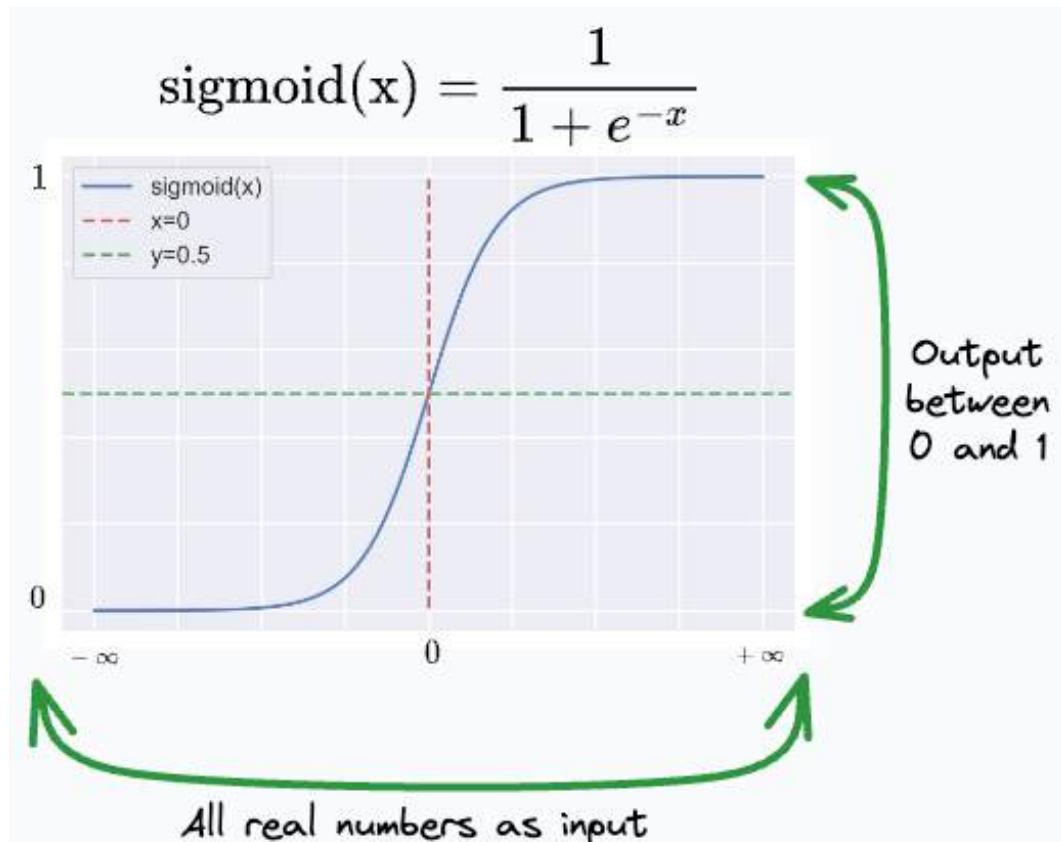
$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

But why?



In other words, have you ever wondered why we use Sigmoid in logistic regression?

The most common reason we get to hear is that Sigmoid maps all real values to the range [0,1].



Sigmoid maps all real values to the range [0,1]

But there are infinitely many functions that can do that.

### What is so special about Sigmoid?

What's more, how can we be sure that the output of Sigmoid is indeed a probability?

See, as discussed above, logistic regression output is interpreted as a probability.



But this raises an essential question: “Can we confidently treat the output of sigmoid as a genuine probability?”

It is important to consider that not every numerical value lying within the interval of [0,1] guarantees that it is a legitimate probability.

In other words, just outputting a number between [0,1] isn’t sufficient for us to start interpreting it as a probability.

Instead, the interpretation must stem from the formulation of logistic regression and its assumptions.

So where did the Sigmoid come from?

**If you have never understood this, then...**

This is precisely what we are discussing in this today’s article, which is **available for free for everyone**.

Simplifying further, we get the following:

$$\begin{aligned} p(y = 1 \mid X) &= \frac{1}{1 + \frac{\exp\left(-\frac{(x+2)^2}{2}\right)}{\exp\left(-\frac{(x-3)^2}{2}\right)}} \\ &= \frac{1}{1 + \exp\left(-\frac{(x+2)^2}{2} + \frac{(x-3)^2}{2}\right)} \\ &= \frac{1}{1 + \exp(-5x + \frac{5}{2})} \end{aligned}$$

Assuming  $z = 5x - \frac{5}{2}$ , we get:

$$p(y = 1 \mid X) = \frac{1}{1 + \exp(-z)}$$

And if you notice closely, this is precisely the sigmoid function!

$$p(y = 1 \mid X) = \frac{1}{1 + \exp(-z)} = \text{sigmoid}(z)$$

Taken from the [Sigmoid Article](#)



## We are covering:

- The common misinterpretations that explain the origin of Sigmoid.
- Why are these interpretations wrong?
- What an ideal output of logistic regression should look like.
- How to formulate the origin of Sigmoid using a generative approach under certain assumptions.
- What if the assumptions don't hold true.
- How the generative approach can be translated into the discriminative approach?
- Best practices while using generative and discriminative approaches.

Hope you will get to learn something new :)

**The article is available for free to everyone.**

👉 Interested folks can read it here: [Why Do We Use Sigmoid in Logistic Regression?](#)



# Build Elegant Data Apps With The Coolest Mito-Streamlit Integration

Personally, I am a big fan of no-code data analysis tools. They are extremely useful in eliminating repetitive code across projects—thereby boosting productivity.

Yet, most no-code tools are often limited in terms of the functionality they support. Thus, flexibility is usually a big challenge while using them.

**Mito** is an incredible open-source tool that lets you analyze data in a spreadsheet interface.

With its latest update, Mito spreadsheets are now compatible with Streamlit-based data apps.

As a result, you can now integrate a Mito sheet directly into a Streamlit data app.

A demo is shown below:

Blog.DailyDoseofDS.com

**Add Mito Spreadsheet to Streamlit**

The screenshot shows the Mito AI interface. At the top, there's a menu bar with options like File, Databases, Columns, Rows, Graphs, Format, Code, View, and Help. Below the menu is a toolbar with various icons. A central workspace has a purple "Import File" button. To the right, a sidebar titled "Mito AI" contains examples such as "create a dataframe named df with sample data" and "import the most recent csv from the current folder". At the bottom, there's a message input field with a placeholder "Send a message..." and a send button. A status bar at the bottom left says "Please import a file to begin".



This is incredibly useful for:

- Creating and sharing interactive data applications
- Allowing non-technical users to explore data
- Automating data manipulation
- Providing instructions for other users as they explore our data
- Presenting visualizations and insights in a data app on the fly, and more.

What's more, Mito recently supercharged its spreadsheet interface with AI. As a result, one can analyze data directly with text prompts.

Isn't that cool?

I'm always curious to read your comments. What do you think about this cool feature addition to Mito? Let me know :)

👉 Get started with Mito-Streamlit integration here: [Mito-Streamlit](#).



# A Simple and Intuitive Guide to Understanding Precision and Recall

I have seen many folks struggling to intuitively understand Precision and Recall.

These fairly straightforward metrics often intimidate many.

Yet, adopting the Mindset Technique can be incredibly helpful.

Let me walk you through it today.

For simplicity, we'll call the "Positive class" as our class of interest.

## Precision

Formally, Precision answers the following question:

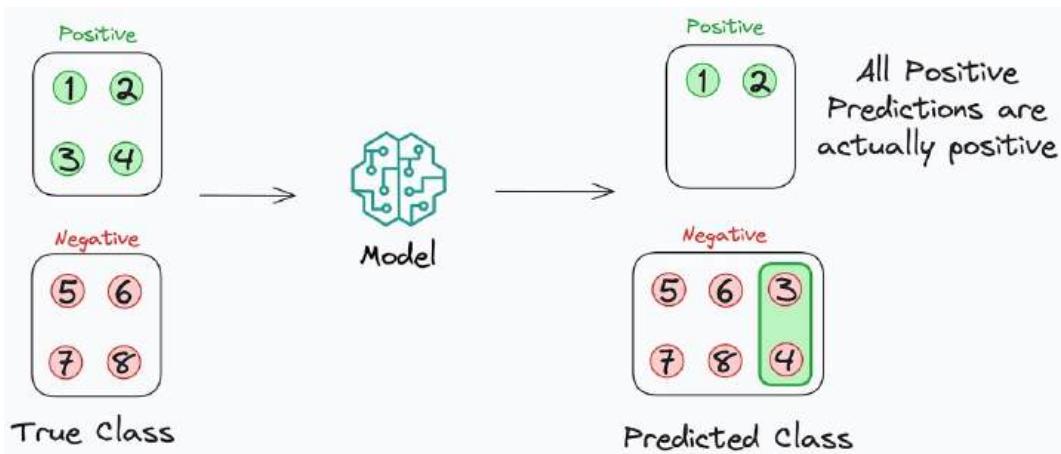
**“What proportion of positive predictions were actually positive?”**

Let's understand that from a mindset perspective.

When you are in a **Precision Mindset**, you don't care about getting every positive sample correctly classified.

But it's important that every positive prediction you get should actually be positive.

The illustration below is an example of high Precision. All positive predictions are indeed positive, even though some positives have been left out.



Precision Mindset: All Positive predictions are actually positive, even though some have been left out

For instance, consider a book recommendation system. Say a positive prediction means you'd like the recommended book.

In a Precision Mindset, you are okay if the model does not recommend all good books in the world.



Precision Mindset: It's okay to miss out on some good books but recommend only good books

But what it recommends should be good.

So even if this system recommended only one book and you liked it, this gives a Precision of 100%.



This is because what it classified as “Positive” was indeed “Positive.”

To summarize, in a high Precision Mindset, all positive predictions should actually be positive.

## Recall

Recall is a bit different. It answers the following question:

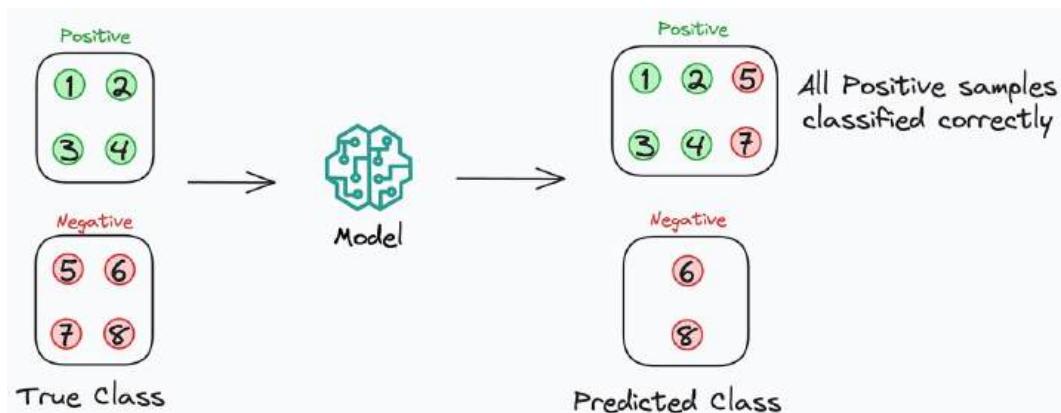
“What proportion of actual positives was identified correctly by the model?”

When you are in a **Recall Mindset**, you care about getting each and every positive sample correctly classified.

It’s okay if some positive predictions were not actually positive.

But all positive samples should get classified as positive.

The illustration below is an example of high recall. All positive samples were classified correctly as positive, even though some were actually negative.

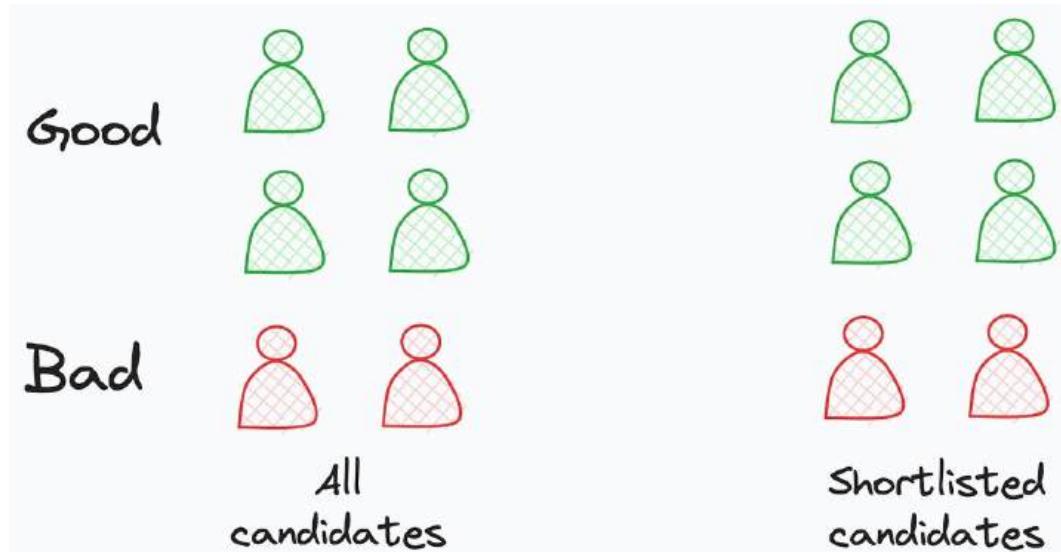


Recall Mindset: All positive samples are correctly classified

For instance, consider an interview shortlisting system based on their resume. A positive prediction means that the candidate should be invited for an interview.



In a **Recall Mindset**, you are okay if the model selects some incompetent candidates.



Recall Mindset: Just focus on correctly classifying all positive samples

But it should not miss out on inviting any skilled candidate.

So even if this system says that all candidates (good or bad) are fit for an interview, it gives you a Recall of 100%.

This is because it didn't miss out on any of the positive samples.

---

Which metric to choose entirely depends on what's important to the problem at hand:

Optimize **Precision** if:

1. You care about getting **ONLY** quality (or positive) predictions.
2. You are okay if some quality (or positive) samples are left out.

Optimize **Recall** if:



1. You care about getting **ALL** quality (or positive) samples correct.
2. You are okay if some non-quality (or negative) samples also come along.

I hope that was helpful :)

👉 Over to you: What analogy did you first use to understand Precision and Recall?



# Skimpy: A Richer Alternative to Pandas' Describe Method



Pandas' describe method is pretty naive.

It hardly highlights any key information about the data.

Instead, try Skimpy.

It is a Jupyter-based tool that provides a standardized and comprehensive data summary.

By invoking a single function, you can generate the above report in seconds.

This includes:

- data shape
- column data types
- column summary statistics
- distribution chart,
- missing stats, etc.

What's more, the summary is grouped by datatypes for faster analysis.

Get started with Skimpy here: [Skimpy](#).



[blog.DailyDoseofDS.com](http://blog.DailyDoseofDS.com)



# A Common Misconception About Model Reproducibility

Today I want to discuss something extremely important about ML model reproducibility.

Imagine you trained an ML model, say a neural network.

It gave a training accuracy of 95% and a test accuracy of 92%.

You trained the model again and got the same performance.

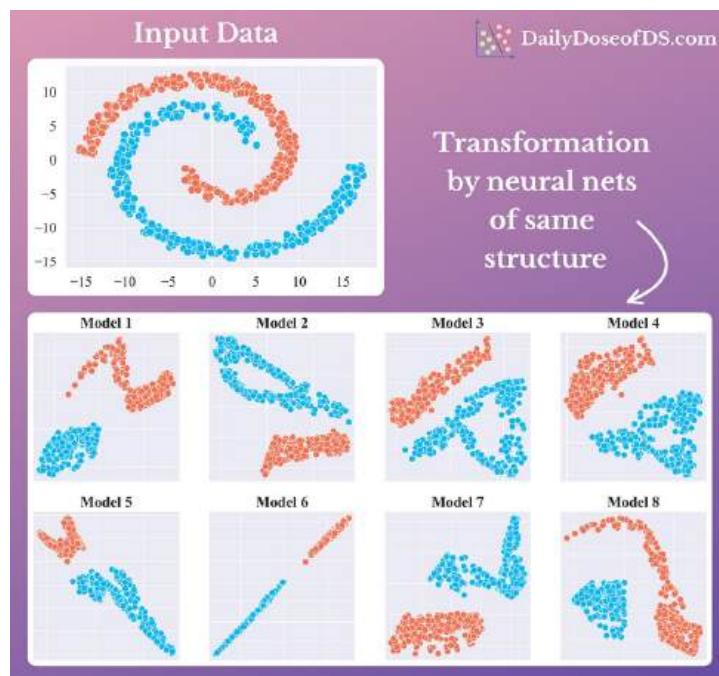
Will you call this a reproducible experiment?

Think for a second before you read further.

---

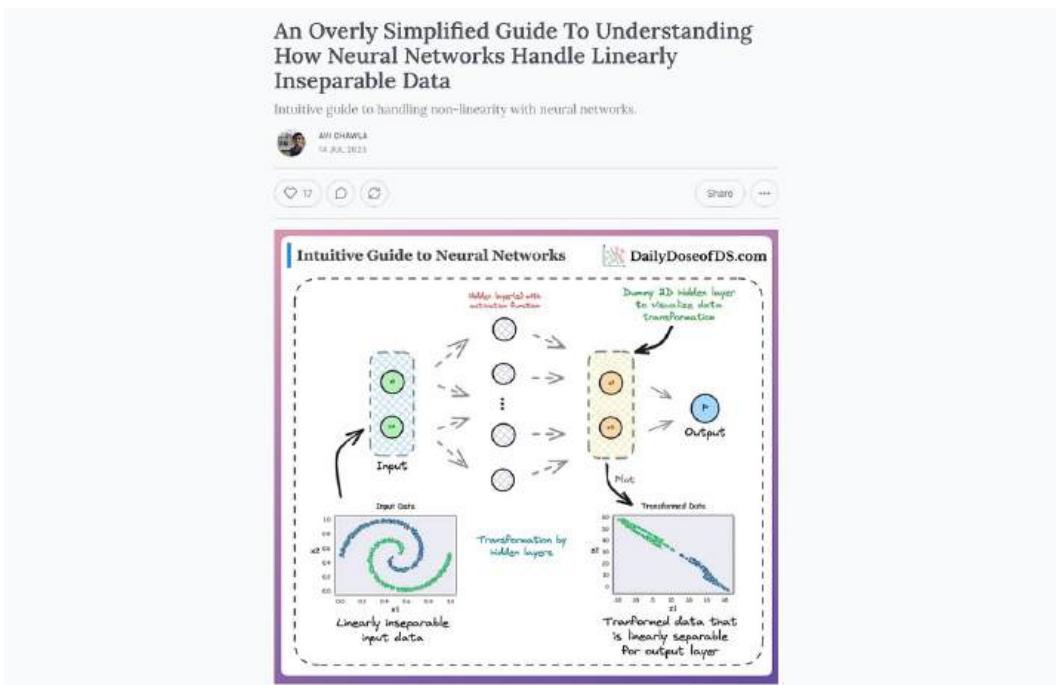
Well, contrary to common belief, this is not what reproducibility means.

To understand better, consider this illustration:





Here, we feed the input data to neural networks with the same architecture but different randomizations. Next, we visualize the transformation using a 2D dummy layer, as I depicted in [one of my previous posts](#) below:



Data transformation in a neural network ([Post Link](#))

All models separate the data pretty well and give 100% accuracy, don't they?

Yet, if you notice closely, each model generates varying data transformations (or decision boundaries).

Now will you call this reproducible?

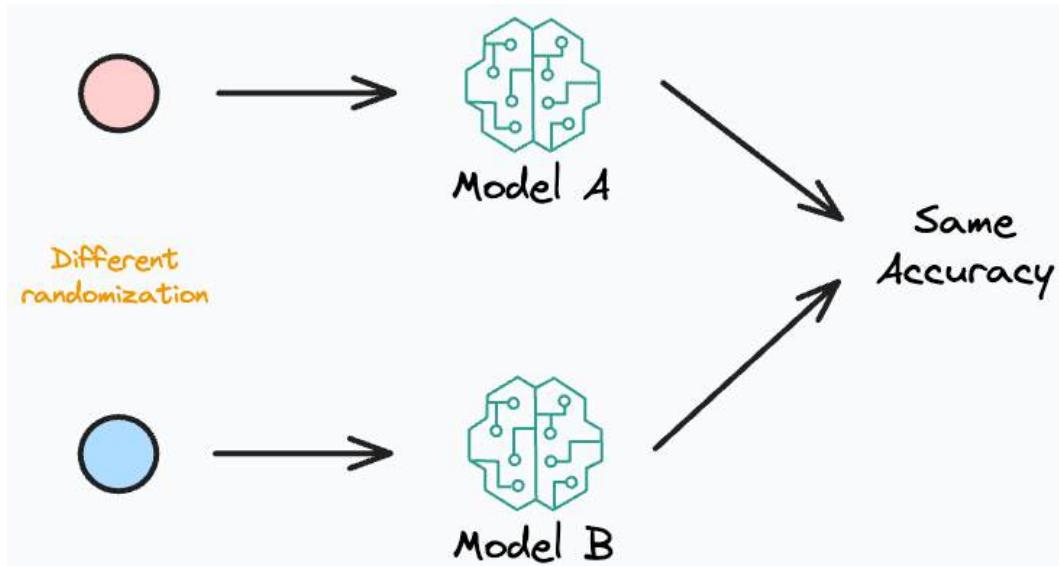
No, right?

**It is important to remember that reproducibility is NEVER measured in terms of performance metrics.**

Instead, reproducibility is ensured when all sources of randomization are reproducible.



It is because two models with the same architecture yet different randomization, can still perform equally well.



Different randomization may still lead to the same accuracy

But that does not make your experiment reproducible.

Instead, it is achieved when all sources of randomization are reproducible.

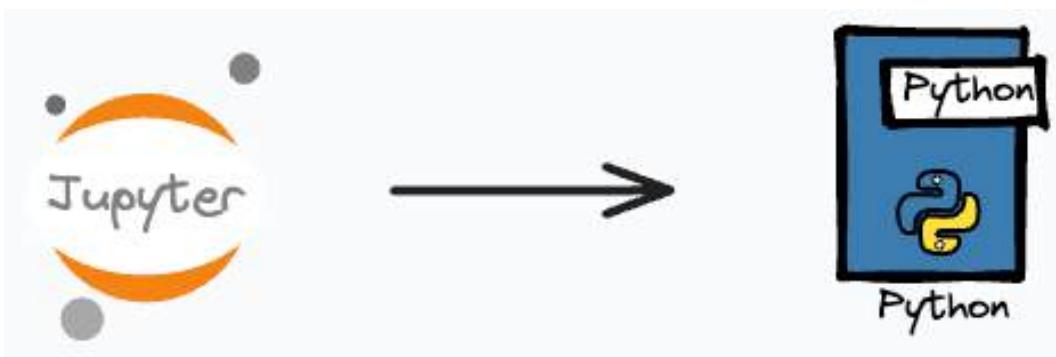
And that is why it is also recommended to set seeds for random generators

Once we do that, reproducibility will automatically follow.

But do you know that besides building a reproducible pipeline, there's another important yet overlooked aspect, especially in data science projects?

**It's testing the pipeline.**

One of the biggest hurdles data science teams face is transitioning their data-driven pipeline from Jupyter Notebooks to an executable, reproducible, error-free, and organized pipeline.



Jupyter to data science pipeline

And this is not something data scientists are particularly fond of doing.

Yet, this is an immensely critical skill that many overlook.

To help you develop that critical skill, this is precisely what we are discussing in **today's member-only blog**.

A screenshot of a blog post titled 'Develop an Elegant Testing Framework For Data Science Projects Using Pytest'. The post is dated Aug 4, 2023, and is categorized under 'Testing'. It is authored by Avi Chawla. The main image shows a purple rounded rectangle containing the 'pytest' logo and two arrows pointing to icons labeled 'Project pipeline' and 'Testing Report'.

### **Blog on testing a data science pipeline using Pytest.**

Testing is already a job that data scientists don't look forward to with much interest.

Considering this, Pytest makes it extremely easy to write test suites, which in turn, immensely helps in developing reliable data science projects.

You will learn the following:

- Why are automation frameworks important?



- What is Pytest?
- How it simplifies pipeline testing?
- How to write and execute tests with Pytest?
- How to customize Pytest's test search?
- How to create an organized testing suite using Pytest markers?
- How to use fixtures to make your testing suite concise and reliable?
- and more.

All in all, building test suites is one of the best skills you can develop to build large and reliable data science pipelines.

👉 Interested folks can read it here: [Develop an Elegant Testing Framework For Python Using Pytest.](#)



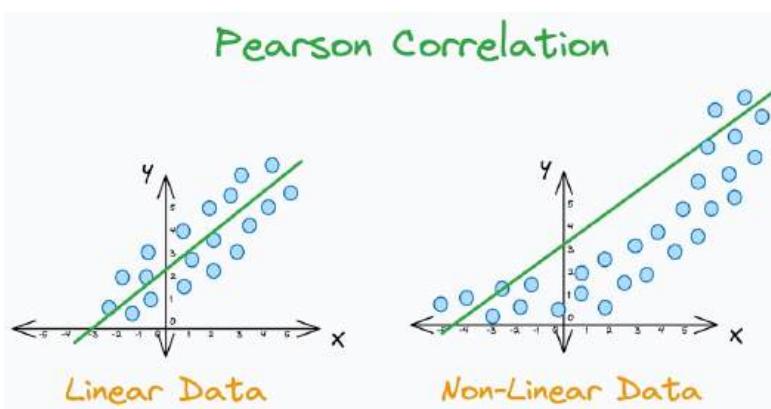
# The Biggest Limitation Of Pearson Correlation Which Many Overlook

Pearson correlation is commonly used to determine the association between two continuous variables.

Many frameworks (in Pandas, for instance) have it as their default correlation metric.

Yet, unknown to many, Pearson correlation:

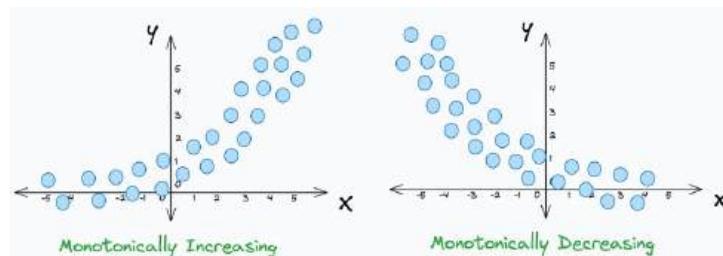
- only measures the linear relationship.
- penalizes a non-linear yet monotonic association.



Pearson correlation only measures the linear relationship

Instead, Spearman correlation is a better alternative.

It assesses monotonicity, which can be linear as well as non-linear.



Monotonicity in data

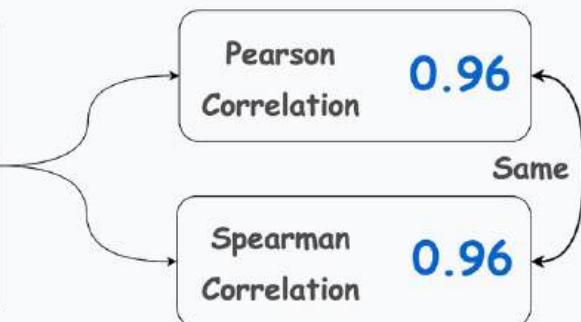
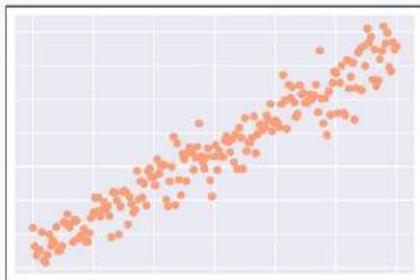


This is evident from the illustration below:

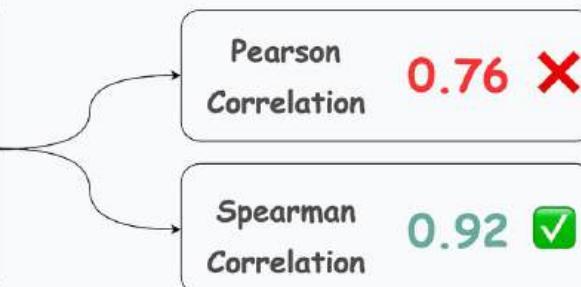
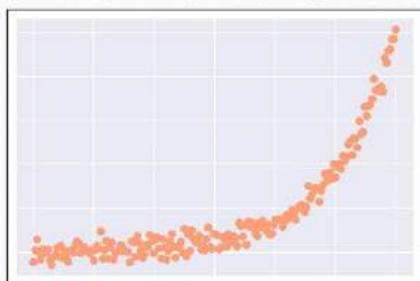
## Limitation of Pearson Correlation



### Linear Data



### Non-linear Data



Pearson vs. Spearman on linear and non-linear data

- Pearson and Spearman correlation is the same on linear data.
- But Pearson correlation underestimates a non-linear association.

Spearman correlation is also useful when data is ranked or ordinal.

👉 Over to you: What are some other alternatives that address Pearson's limitations?



# Gigasheet: Effortlessly Analyse Up to 1 Billion Rows Without Any Code

Traditional Python-based tools become increasingly ineffective and impractical as you move towards scale.



Python-based solutions on small datasets vs. large datasets

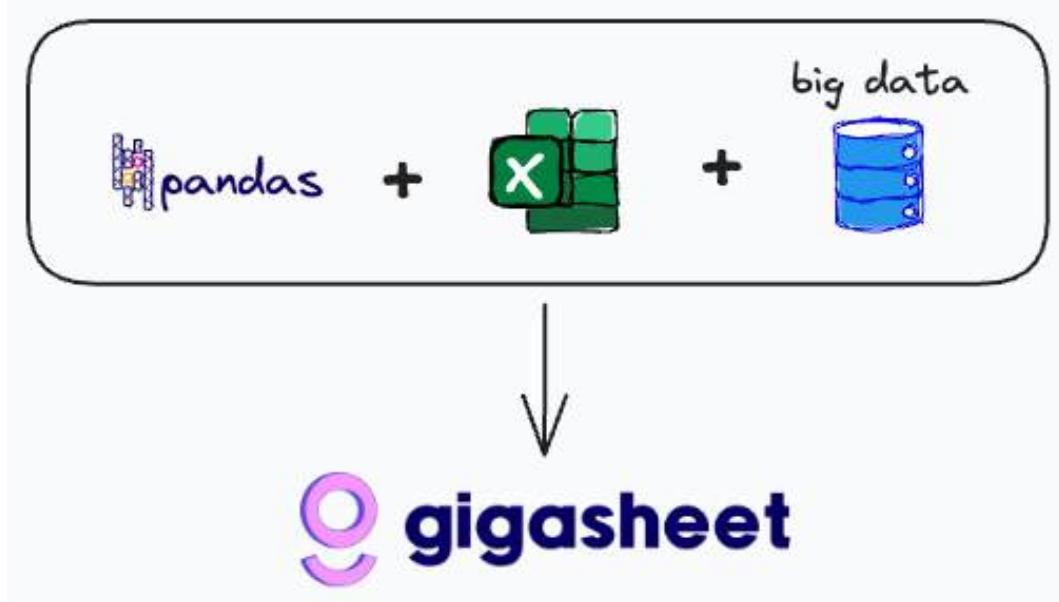
Such cases demand:

- appropriate infrastructure for data storage and manipulation.
- specialized expertise in data engineering, and more.

...which is not feasible at times.

Gigasheet is a no-code tool that seamlessly addresses these pain points.

Think of it like a combination of Excel and Pandas **with no scale limitations**.



As shown below, I used Gigsheet to load a CSV file with **1 Billion rows** and **47 GB** in size, which is massive.

The screenshot shows the Gigsheet library interface. At the top, there's a toolbar with 'Combine', 'Move File', and 'Delete' buttons. Below that is a table header with columns: FILE NAME, ROWS, COLS, SIZE, and LAST MODIFIED. The table contains one row for a file named '1 Billion Dataset.csv'. The 'ROWS' column shows '1.1B', the 'SIZE' column shows '47.02 GB', and the 'LAST MODIFIED' column shows 'Jul 26, 2023'. Both the 'ROWS' and 'SIZE' columns are highlighted with a teal rounded rectangle.

FILE NAME	ROWS	COLS	SIZE	LAST MODIFIED
1 Billion Dataset.csv	1.1B	6	47.02 GB	Jul 26, 2023

Loading 1B rows with [Gigsheet](#).

You can perform any data analysis/engineering tasks by simply interacting with a UI.

Thus, you can do all of the following without worrying about any infra issues:

- Explore any large dataset — **even as big as 1 Billion rows** without code.



1 Billion Dataset.csv > Share

File Data Cleanup Functions Filter Group ↑ Sorted by 2 field(s) Chart Data Enrichments Reset Sheet

#	timestamp	dest_ip	source_ip	port	bytes
9	1578326400021	12.43.98.93	18.85.31.68	79	979
10	1578326400021	14.32.68.107	12.38.62.113	72	1036
11	1578326400022	13.48.126.55	18.100.109.39	123	1506
12	1578326400022	14.51.37.21	16.118.26.44	123	1506
13	1578326400023	14.49.44.92	18.36.97.103	22	1152
14	1578326400023	14.53.76.24	16.73.63.85	118	5775
15	1578326400025	14.37.108.54	17.107.62.181	94	1086
16	1578326400025	14.49.89.121	12.42.39.44	114	2462
17	1578326400031	15.126.63.29	12.52.86.104	119	6752
18	1578326400035	12.45.122.125	17.103.35.100	25	42
19	1578326400035	14.32.69.91	13.58.72.67	57	36215
20	1578326400035	19.92.62.102	12.43.66.128	119	9016
21	1578326400036	14.57.68.122	18.92.54.85	105	13673
22	1578326400038	16.94.88.61	14.58.52.112	94	1361
23	1578326400038	17.49.84.104	12.36.93.128	75	75277

Sum 9,038,86

View 100 Page 1 of 10574773

Result Set: 1,057,477,300 rows

- Perform almost all tabular operations you would typically do, such as:

1 Billion Dataset.csv > Share

File Data Cleanup Functions Filter Group Sort Chart Data Enrichments

#	timestamp	dest_ip	source_ip	port	bytes
1578326400001	13.43.52.51	18.70.112.62	40	57354	
1578326400005	16.79.101.100	12.48.65.39	92	11895	
1578326400007	18.43.118.103	14.51.30.86	27	208	
1578326400011	15.71.108.118	14.50.119.33	57	7496	
1578326400012	14.33.30.103	15.24.31.23	115	20979	
1578326400012	18.121.115.31	13.56.39.74	92	8620	
1578326400014	16.108.75.29	14.34.34.69	65	46033	
1578326400018	12.46.104.126	16.25.76.33	123	1500	

### Execute tabular data operations

- merge,
- plot,
- group,



- sort,
- summary stats, etc.
- Import data from any source like AWS S3, Drive, databases, etc., and analyze it, and more.

What's more, using Gigasheet's Sheet Assistant, you can also interact with your data by providing text instructions.

Lastly, Gigasheet also provides an [API](#). This allows you to:

- automate any repetitive tasks
- schedule imports and exports, and much more.

To summarize, Gigasheet immensely simplifies tabular data exploration tasks.

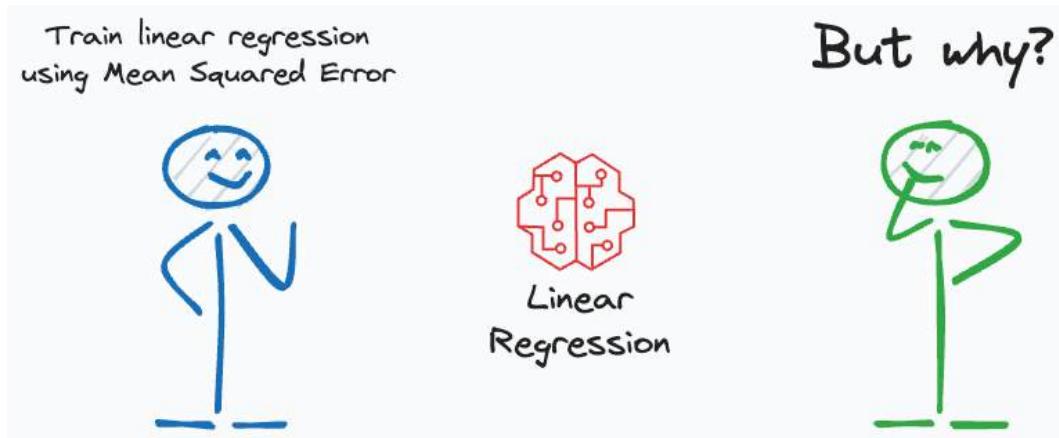
Anyone with or without data engineering skills can use Gigasheet for tabular tasks, directly from a UI.

Isn't that cool?

👉 Get started with Gigasheet here: [Gigasheet](https://gigasheet.com).



# Why Mean Squared Error (MSE)?



Say you wish to train a linear regression model. We know that we train it by minimizing the squared error:

$$\text{Squared Error} = \sum_i^n \frac{(y_i - \theta^T x_i)^2}{n}$$

**But have you ever wondered why we specifically use the squared error?**

See, many functions can potentially minimize the difference between observed and predicted values. But of all the possible choices, what is so special about the squared error?

In my experience, people often say:

- Squared error is differentiable. That is why we use it as a loss function. **WRONG.**
- It is better than using absolute error as squared error penalizes large errors more. **WRONG.**



Sadly, each of these explanations are incorrect.

But approaching it from a probabilistic perspective helps us truly understand why the squared error is the most ideal choice.

Let's begin.

In linear regression, we predict our target variable  $y$  using the inputs  $X$  as follows:

$$y_i = \theta^T x_i + \epsilon_i$$

Here, epsilon is an error term that captures the random noise for a specific data point (i).

We assume the noise is drawn from a Gaussian distribution with zero mean based on the central limit theorem:

$$\epsilon \sim N(0, \sigma^2)$$

Thus, the probability of observing the error term can be written as:

$$p(\epsilon_i) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{\epsilon_i^2}{2\sigma^2}\right)$$



Substituting the error term from the linear regression equation, we get:

$$\epsilon_i = \theta^T x_i - y_i$$

$$p(y_i|x_i; \theta) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i - \theta^T x_i)^2}{2\sigma^2}\right)$$

This is called the distribution of  $y$  given  $x$ ; when parametrized by  $\theta$

For a specific set of parameters  $\theta$ , the above tells us the probability of observing a data point (i).

Next, we can define the likelihood function as follows:

$$L(\theta) = p(y|X; \theta)$$

The likelihood is a function of  $\theta$ . It means that by varying  $\theta$ , we can fit a distribution to the observed data and quantify the likelihood of observing it.

We further write it as a product for individual data points because we assume all observations are independent.



The likelihood of observing all observations is the same as the product of observing individual observations

Thus, we get:

$$\begin{aligned} L(\theta) &= \prod_i^n p(y_i|x_i; \theta) \\ &= \prod_i^n \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i - \theta^T x_i)^2}{2\sigma^2}\right) \end{aligned}$$

Likelihood function

Since the log function is monotonic, we use the log-likelihood and maximize it. This is called **maximum likelihood estimation (MLE)**.

$$\begin{aligned} \log(L(\theta)) &= \log\left(\prod_i^n p(y_i|x_i; \theta)\right) \\ &= \log\left(\prod_i^n \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i - \theta^T x_i)^2}{2\sigma^2}\right)\right) \end{aligned}$$

Taking the log on both sides in the likelihood function

Simplifying, we get:



$$\log(L(\theta)) = \sum_i^n \log\left(\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i - \theta^T x_i)^2}{2\sigma^2}\right)\right)$$

distribute the logarithm

$$= \sum_i^n \log\left(\frac{1}{\sqrt{2\pi\sigma^2}}\right) - \frac{(y_i - \theta^T x_i)^2}{2\sigma^2}$$

distribute the summation

$$= \boxed{n \cdot \log\left(\frac{1}{\sqrt{2\pi\sigma^2}}\right)} - \boxed{\sum_i^n \frac{(y_i - \theta^T x_i)^2}{2\sigma^2}}$$

constant

To reiterate, the objective is to find the  $\theta$  that maximizes the above expression.

But the first term is independent of  $\theta$ . Thus, maximizing the above expression is equivalent to minimizing the second term.

**And if you notice closely, it's precisely the squared error.**

$$\frac{1}{2\sigma^2} \sum_i^n (y_i - \theta^T x_i)^2$$

Thus, you can maximize the log-likelihood by minimizing the squared error.

And this is the origin of least-squares in linear regression.



See, there's clear proof and reasoning behind for using squared error as a loss function in linear regression.

Nothing comes from thin air in machine learning :)

**But did you notice that in this derivation, we made a lot of assumptions?**

Firstly, we assumed the noise was drawn from a Gaussian distribution. **But why?**

$$\epsilon \sim N(0, \sigma^2)$$

We assumed independence of observations. **Why and what if it does not hold true?**

Next, we assumed that each error term is drawn from a distribution with the same variance  $\sigma$ . But what if it looks like this:

$$\epsilon_i \sim N(0, \sigma_i^2)$$

Each error term is drawn from a distribution with a different variance

In that case, the squared error will come out to be:

$$\text{Squared Error} = \sum_i^n \frac{(y_i - \theta^T x_i)^2}{2\sigma_i^2}$$



How to handle this?

This is precisely what I have discussed in [today's member-only blog.](#)

In other words, have you ever wondered about the origin of linear regression assumptions? The assumptions just can't appear from thin air, can they?

Thus today's deep dive walks you through the origin of each of the assumptions of linear regression in a lot of detail.

The thumbnail features a cartoon illustration of a person holding a brain labeled 'Linear Regression'. Text on the left says 'These are the assumptions of linear regression' and 'But where did they come from?'. On the right, it says 'Machine Learning Aug 1, 2023' and 'Where Did The Assumptions of Linear Regression Originate From?'. Below the title is a description: 'The most extensive and in-depth guide to linear regression.' and the author's name 'Avi Chawla'.

## [Blog on the origin of assumptions of linear regression](#)

It covers the following:

- An overview of linear regression and why we use Mean Squared Error in linear regression.
- What is the assumed data generation process of linear regression?
- What are the critical assumptions of linear regression?
- Why error term is assumed to follow a normal distribution?
- Why are these assumptions essential?
- How are these assumptions derived?
- How to validate them?
- What measures can we take if the assumptions are violated?
- Best practices.



All in all, a literal deep-dive on linear regression. The more you will learn, the more you will appreciate the beauty of linear regression :)

👉 Interested folks can read it here: [Where Did The Assumptions of Linear Regression Originate From?](#)



# A More Robust and Underrated Alternative To Random Forests

We know that Decision Trees always overfit.

This is because by default, a decision tree (in sklearn's implementation, for instance), is allowed to grow until all leaves are pure.

As the model correctly classifies ALL training instances, this leads to:

- 100% overfitting, and
- poor generalization

Random Forest address this by introducing randomness in two ways:

- While creating a bootstrapped dataset.
- While deciding a node's split criteria by choosing candidate features randomly.

Yet, the chances of overfitting are still high.

The Extra Trees algorithm is an even more robust alternative to Random Forest.

👉 Note:

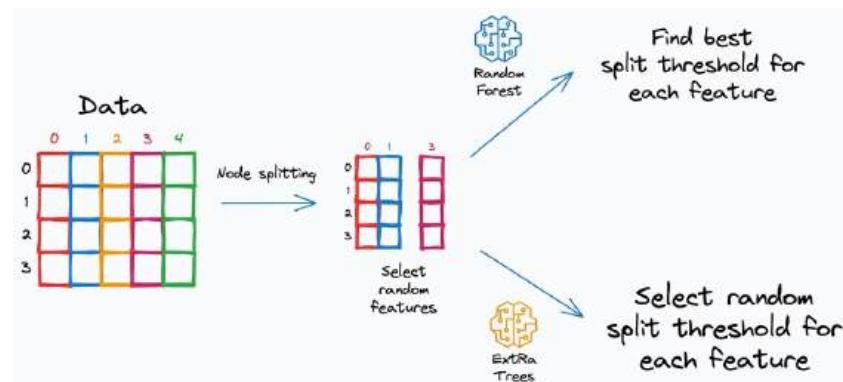
- Extra Trees does not mean more trees.
- Instead, it should be written as **ExtRa**, which means **Extra Randomized**.

ExtRa Trees are Random Forests with an additional source of randomness.

Here's how it works:



- Create a bootstrapped dataset for each tree (same as RF)
- Select candidate features randomly for node splitting (same as RF)
- Now, Random Forest calculates the best split threshold for each candidate feature.
- But ExtRa Trees chooses this split threshold randomly.



### Random Forest vs. ExtRa Trees

- This is the source of extra randomness.
- After that, the best candidate feature is selected.

This further reduces the variance of the model.

The effectiveness is evident from the image below:

A More Robust Alternative to Random Forests			
Algorithm	Train Accuracy	Test Accuracy	Cross Validation Score
Decision Tree	1.00	0.58	0.62
Random Forest	0.98	0.75	0.79
Extra Trees	0.96	0.74	0.80



### Decision Tree vs. Random Forest vs. ExtRa Trees

- Decision Trees entirely overfit
- Random Forests work better
- Extra Trees performs even better

⚠ A cautionary measure while using [ExtRa Trees from Sklearn.](#)

By default, the `bootstrap` flag is set to False.

**`bootstrap : bool, default=False`**

Whether bootstrap samples are used when building trees. If False, the whole dataset is used to build each tree.

Make sure you run it with `bootstrap=True`, otherwise, it will use the whole dataset for each tree.

👉 Over to you: Can you think of another way to add randomness to Random Forest?



# The Most Overlooked Problem With Imputing Missing Values Using Zero (or Mean)

Replacing (imputing) missing values with mean or zero or any other fixed value:

- alters summary statistics
- changes the distribution
- inflates the presence of a specific value

This can lead to:

- inaccurate modeling
- incorrect conclusions, and more.

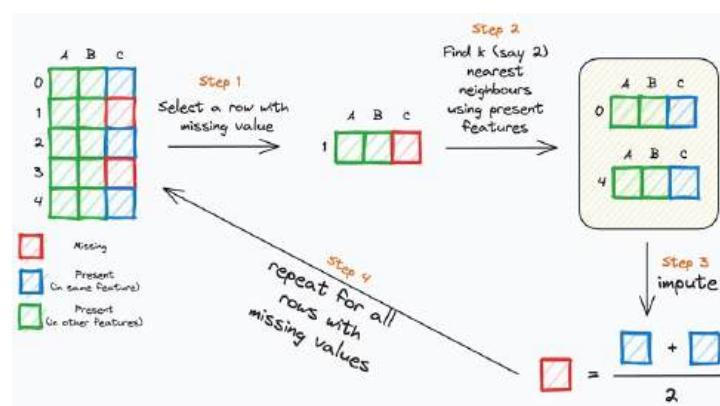
Instead, always try to impute missing values with more precision.

kNN imputer is often a great choice in such cases.

It imputes missing values using the k-Nearest Neighbors algorithm.

Missing features are imputed by running a kNN on non-missing feature values.

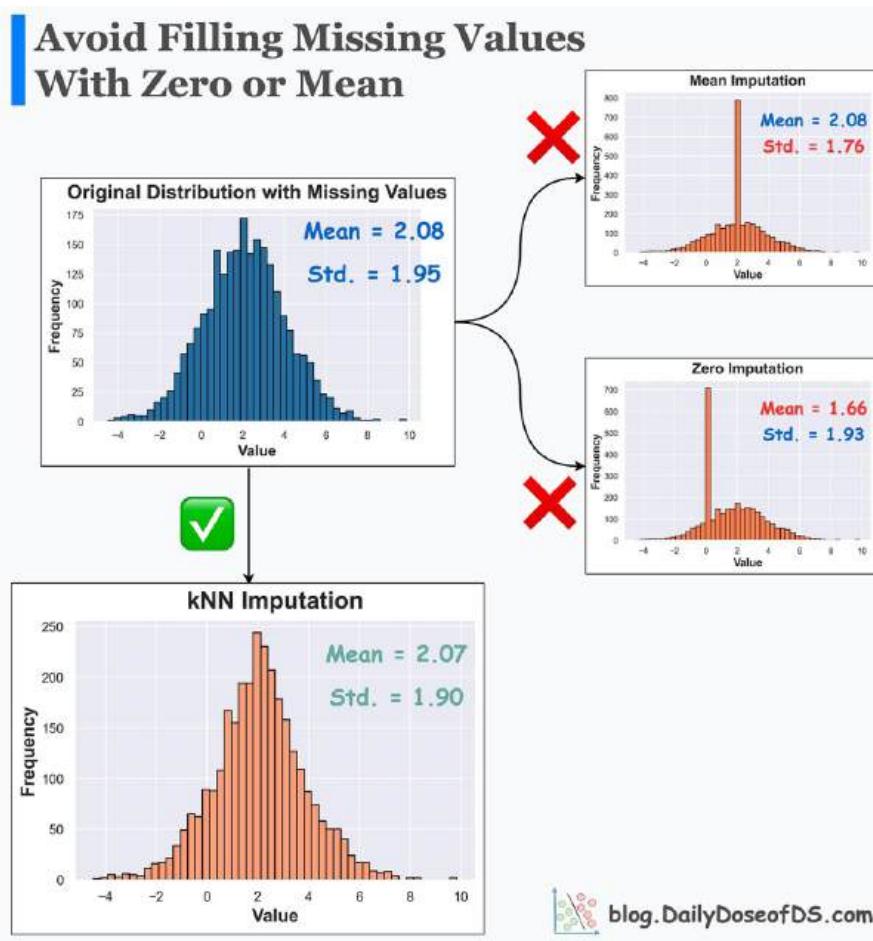
The following image depicts how it works:





- **Step 1:** Select a row ( $r$ ) with a missing value.
- **Step 2:** Find its  $k$  nearest neighbors using the non-missing feature values.
- **Step 3:** Impute the missing feature of the row ( $r$ ) using the corresponding non-missing values of  $k$  nearest neighbor rows.
- **Step 4:** Repeat for all rows with missing values.

Its effectiveness over Mean/Zero imputation is evident from the image below.



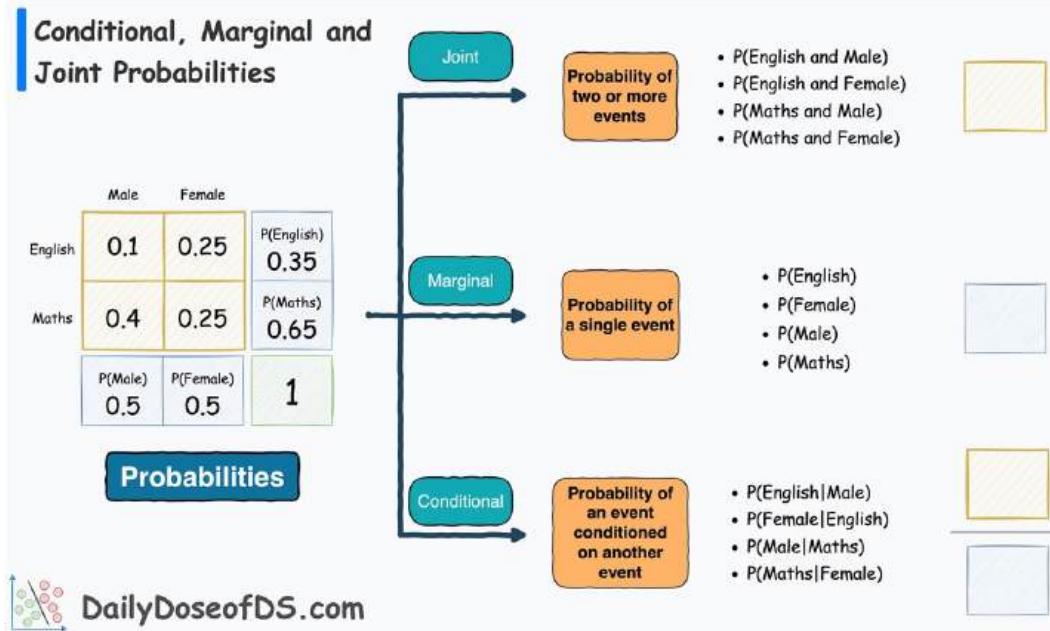
#### Mean and Zero imputation vs. kNN imputation

- Mean/Zero alters the summary statistics and distribution.
- kNN imputer preserves them.

Get started with kNN imputer: [Sklearn Docs.](https://scikit-learn.org/stable/modules/impute.html#knn-imputation)



# A Visual Guide to Joint, Marginal and Conditional Probabilities



This issue had mathematical derivations and many diagrams. Please read it here:  
<https://www.blog.dailydoseofds.com/p/a-visual-guide-to-joint-marginal>



# Jupyter Notebook 7: Possibly One Of The Best Updates To Jupyter Ever

This is probably one of the best updates to Jupyter Notebook ever.

Jupyter has announced the release of Jupyter Notebook 7

The developers call it one of the most significant releases in years.

Here are some highlights:

- **Real-time collaboration:** Share notebooks with others and collaborate. Extremely useful for teams.
- **Interactive debugging:** Debug code cell by cell and inspect variables.
- **Internationalization:** Change language
- **Dark mode**
- **Table of contents**

The demo above shows real-time collaboration between two notebooks.

Isn't that cool?

The update is in beta. You can read more about it here: [Jupyter Notebook 7](#).

👉 Over to you: Which of these new updates is your favorite?

Read the full issue here to watch an animation of real-time collaboration feature:

<https://www.blog.dailydoseofds.com/p/jupyter-notebook-7-one-of-the-best>

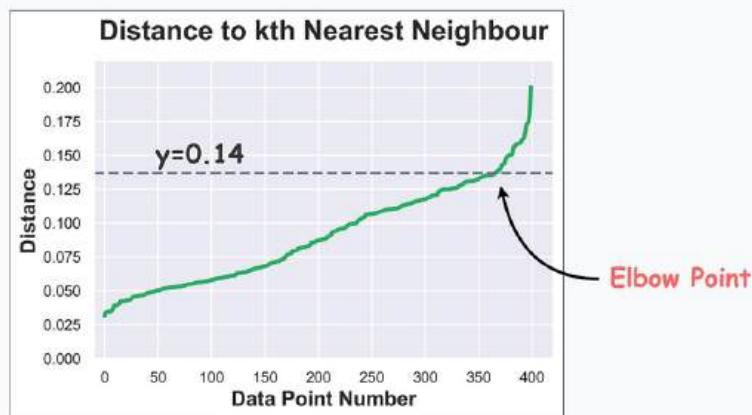


# How to Find Optimal Epsilon Value For DBSCAN Clustering?

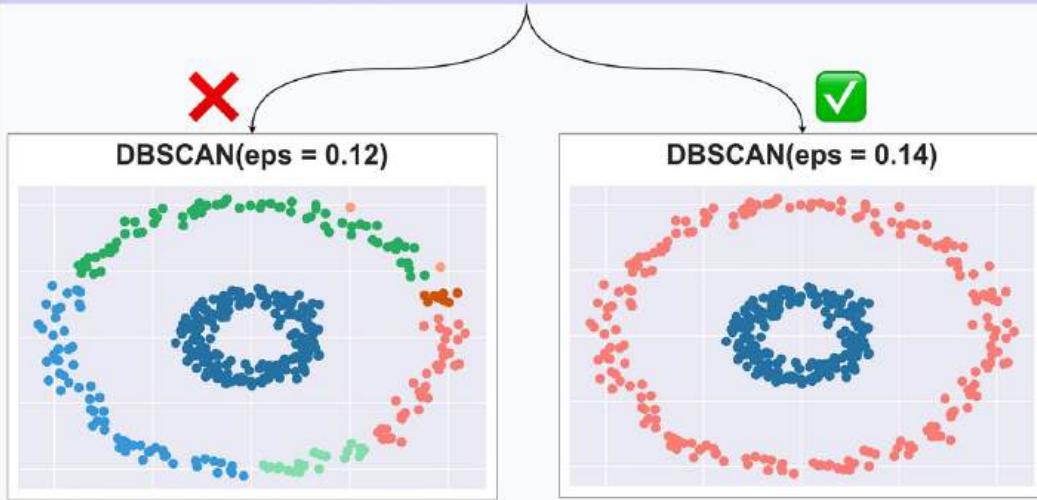
## Find Optimal Epsilon in DBSCAN Clustering



Step 1) Plot distance to  $k^{\text{th}}$  nearest neighbour for every point



Step 2) Select value at elbow point as Epsilon



In DBSCAN, determining the epsilon parameter is often tricky.

Yet, the Elbow curve is often helpful in determining it.

To begin, DBSCAN has three hyperparameters:

1. Epsilon: two points are considered neighbors if they are closer than Epsilon.



2. min\_samples: Min neighbors for a point to be classified as a core point.
3. The distance metric.

We can use the Elbow Curve to find an optimal value of Epsilon:

Set k as the min\_samples hyperparameter.

For every data point, plot the distance to its kth nearest neighbor (in increasing order).

The optimal value of Epsilon is found near the elbow point.

### Why does it work?

Recall that we are measuring the distance to a specific (kth) neighbor for all points.

Thus, the elbow point suggests a distance to a more isolated point or a point in a different cluster.

The point where change is most pronounced hints towards an optimal epsilon.

The efficacy is evident from the image above.

Selecting the elbow value provides better clustering results over another value.

👉 Over to you: What methods do you use to find an optimal epsilon for DBSCAN?



# Why R-squared is a Flawed Regression Metric

R<sup>2</sup> is quite popularly used all across data science and statistics to assess a model.

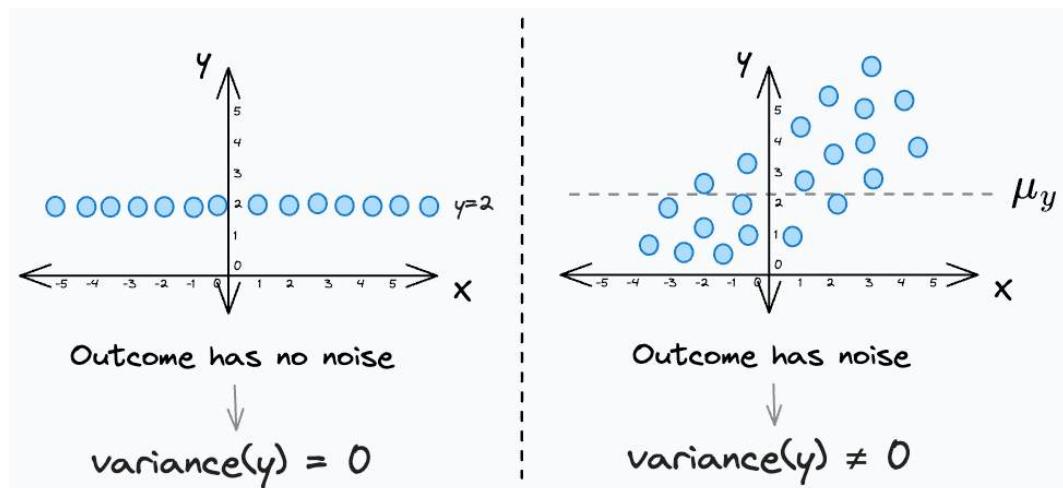
Yet, contrary to common belief, it is often interpreted as a performance metric for evaluating a model, when, in reality, it is not.

Let's understand!

R<sup>2</sup> tells the fraction of variability in the outcome variable captured by a model.

It is defined as follows:

In simple words, variability depicts the noise in the outcome variable ( $y$ ).



**Left:** The outcome variable has zero variance. **Right:** The outcome variable has a non-zero variance.

Thus, the more variability captured, the higher the R<sup>2</sup>.



This means that solely optimizing for R<sup>2</sup> as a performance measure:

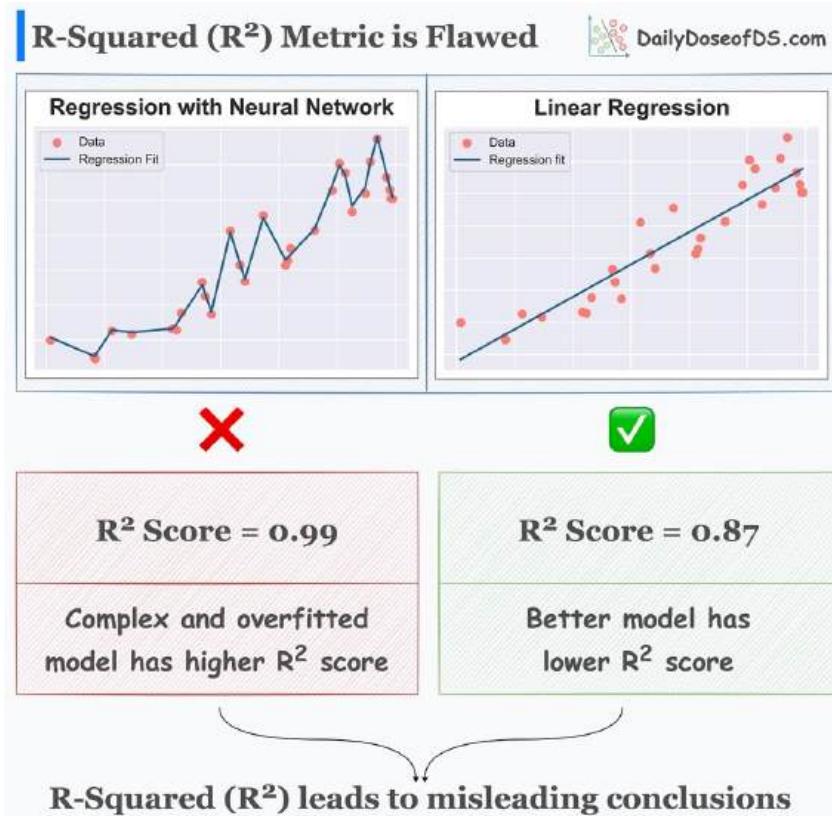
- promotes 100% overfitting.
- leads us to engineer the model in a way that captures random noise instead of underlying patterns.

It is important to note that:

- R<sup>2</sup> is NOT a measure of predictive power.
- Instead, R<sup>2</sup> is a fitting measure.

Thus, you should NEVER use it to measure goodness of fit.

This is evident from the image below:



- An overly complex and overfitted model almost gets a perfect R<sup>2</sup> of 1.
- A better and more generalized model gets a lower R<sup>2</sup> score.



Some other flaws of R<sup>2</sup> are:

- R<sup>2</sup> always increases as you add more features, even if they are random noise.
- In some cases, one can determine R<sup>2</sup> even before fitting a model, which is weird.

👉 Read my full blog on the A-Z of R<sup>2</sup>, what it is, its limitations, and much more here: [Flaws of R<sup>2</sup> Metric](#).

👉 Over to you: What are some other flaws in R<sup>2</sup>?



# 75 Key Terms That All Data Scientists Remember By Heart

## 75 Terms That All Data Scientists Remember by Heart ❤️

By Avi Chawla

DailyDoseofDS.com

<b>C</b> Clustering Confusion Matrix Cross-validation	<b>D</b> Decision Trees Dimensionality Reduction Discriminative Model	<b>E</b> Ensemble EDA Entropy	<b>F</b> Feature Engineering F-score Feature Extraction
<b>G</b> Gradient Descent Gaussian Distribution Gradient Boosting	<b>H</b> Hypothesis Hierarchical Clustering Heteroscedasticity	<b>I</b> Information Gain Independent Variable Imbalance	<b>J</b> Jupyter Joint Probability Jaccard Index
<b>K</b> Kernel Density Estimation KS Test KMeans Clustering	<b>L</b> Likelihood Linear Regression L1/L2 Regularization	<b>M</b> Max Likelihood Estimation Multicollinearity Mutual Information	<b>N</b> Naive Bayes Normalisation Null Hypothesis
<b>O</b> Overfitting Outliers One-hot encoding	<b>P</b> PCA Precision P-value	<b>Q</b> QQ-Plot QR decomposition	<b>R</b> Random Forest Recall ROC Curve
<b>S</b> SVM Standardisation Sampling	<b>T</b> t-SNE T-distribution Type I/II Error	<b>U</b> Underfitting UMAP Uniform Distribution	<b>V</b> Variance Validation Curve Vanishing Gradient
<b>W</b> Word Embedding Word Cloud Weights	<b>X</b> XGBoost XLNet	<b>Y</b> YOLO Yellowbrick	<b>Z</b> Z-score Z-test Zero-shot learning

Data science has a diverse glossary. The sheet lists the 75 most common and important terms that data scientists use almost every day.



Thus, being aware of them is extremely crucial.

- A:
  - **Accuracy:** Measure of the correct predictions divided by the total predictions.
  - **Area Under Curve:** Metric representing the area under the Receiver Operating Characteristic (ROC) curve, used to evaluate classification models.
  - **ARIMA:** Autoregressive Integrated Moving Average, a time series forecasting method.
- B:
  - **Bias:** The difference between the true value and the predicted value in a statistical model.
  - **Bayes Theorem:** Probability formula that calculates the likelihood of an event based on prior knowledge.
  - **Binomial Distribution:** Probability distribution that models the number of successes in a fixed number of independent Bernoulli trials.
- C:
  - **Clustering:** Grouping data points based on similarities.
  - **Confusion Matrix:** Table used to evaluate the performance of a classification model.
  - **Cross-validation:** Technique to assess model performance by dividing data into subsets for training and testing.
- D:
  - **Decision Trees:** Tree-like model used for classification and regression tasks.
  - **Dimensionality Reduction:** Process of reducing the number of features in a dataset while preserving important information.
  - **Discriminative Models:** Models that learn the boundary between different classes.
- E:



- **Ensemble Learning:** Technique that combines multiple models to improve predictive performance.
- **EDA (Exploratory Data Analysis):** Process of analyzing and visualizing data to understand its patterns and properties.
- **Entropy:** Measure of uncertainty or randomness in information.
- F:
  - **Feature Engineering:** Process of creating new features from existing data to improve model performance.
  - **F-score:** Metric that balances precision and recall for binary classification.
  - **Feature Extraction:** Process of automatically extracting meaningful features from data.
- G:
  - **Gradient Descent:** Optimization algorithm used to minimize a function by adjusting parameters iteratively.
  - **Gaussian Distribution:** Normal distribution with a bell-shaped probability density function.
  - **Gradient Boosting:** Ensemble learning method that builds multiple weak learners sequentially.
- H:
  - **Hypothesis:** Testable statement or assumption in statistical inference.
  - **Hierarchical Clustering:** Clustering method that organizes data into a tree-like structure.
  - **Heteroscedasticity:** Unequal variance of errors in a regression model.
- I:
  - **Information Gain:** Measure used in decision trees to determine the importance of a feature.



- **Independent Variable:** Variable that is manipulated in an experiment to observe its effect on the dependent variable.
- **Imbalance:** Situation where the distribution of classes in a dataset is not equal.
- J:
  - **Jupyter:** Interactive computing environment used for data analysis and machine learning.
  - **Joint Probability:** Probability of two or more events occurring together.
  - **Jaccard Index:** Measure of similarity between two sets.
- K:
  - **Kernel Density Estimation:** Non-parametric method to estimate the probability density function of a continuous random variable.
  - **KS Test (Kolmogorov-Smirnov Test):** Non-parametric test to compare two probability distributions.
  - **KMeans Clustering:** Partitioning data into K clusters based on similarity.
- L:
  - **Likelihood:** Chance of observing the data given a specific model.
  - **Linear Regression:** Statistical method for modeling the relationship between dependent and independent variables.
  - **L1/L2 Regularization:** Techniques to prevent overfitting by adding penalty terms to the model's loss function.
- M:
  - **Maximum Likelihood Estimation:** Method to estimate the parameters of a statistical model.



- **Multicollinearity:** A situation where two or more independent variables are highly correlated in a regression model.
- **Mutual Information:** Measure of the amount of information shared between two variables.
- N:
  - **Naive Bayes:** Probabilistic classifier based on Bayes Theorem with the assumption of feature independence.
  - **Normalization:** Scaling data to have a mean of 0 and standard deviation of 1.
  - **Null Hypothesis:** Hypothesis of no significant difference or effect in statistical testing.
- O:
  - **Overfitting:** When a model performs well on training data but poorly on new, unseen data.
  - **Outliers:** Data points that significantly differ from other data points in a dataset.
  - **One-hot encoding:** Process of converting categorical variables into binary vectors.
- P:
  - **PCA (Principal Component Analysis):** Dimensionality reduction technique to transform data into orthogonal components.
  - **Precision:** Proportion of true positive predictions among all positive predictions in a classification model.
  - **p-value:** Probability of observing a result at least as extreme as the one obtained if the null hypothesis is true.
- Q:
  - **QQ-plot (Quantile-Quantile Plot):** Graphical tool to compare the distribution of two datasets.
  - **QR decomposition:** Factorization of a matrix into an orthogonal and an upper triangular matrix.



- R:
  - **Random Forest:** Ensemble learning method using multiple decision trees to make predictions.
  - **Recall:** Proportion of true positive predictions among all actual positive instances in a classification model.
  - **ROC Curve (Receiver Operating Characteristic Curve):** Graph showing the performance of a binary classifier at different thresholds.
- S:
  - **SVM (Support Vector Machine):** Supervised machine learning algorithm used for classification and regression.
  - **Standardisation:** Scaling data to have a mean of 0 and a standard deviation of 1.
  - **Sampling:** Process of selecting a subset of data points from a larger dataset.
- T:
  - **t-SNE (t-Distributed Stochastic Neighbor Embedding):** Dimensionality reduction technique for visualizing high-dimensional data in lower dimensions.
  - **t-distribution:** Probability distribution used in hypothesis testing when the sample size is small.
  - **Type I/II Error:** Type I error is a false positive, and Type II error is a false negative in hypothesis testing.
- U:
  - **Underfitting:** When a model is too simple to capture the underlying patterns in the data.
  - **UMAP (Uniform Manifold Approximation and Projection):** Dimensionality reduction technique for visualizing high-dimensional data.
  - **Uniform Distribution:** Probability distribution where all outcomes are equally likely.
- V:

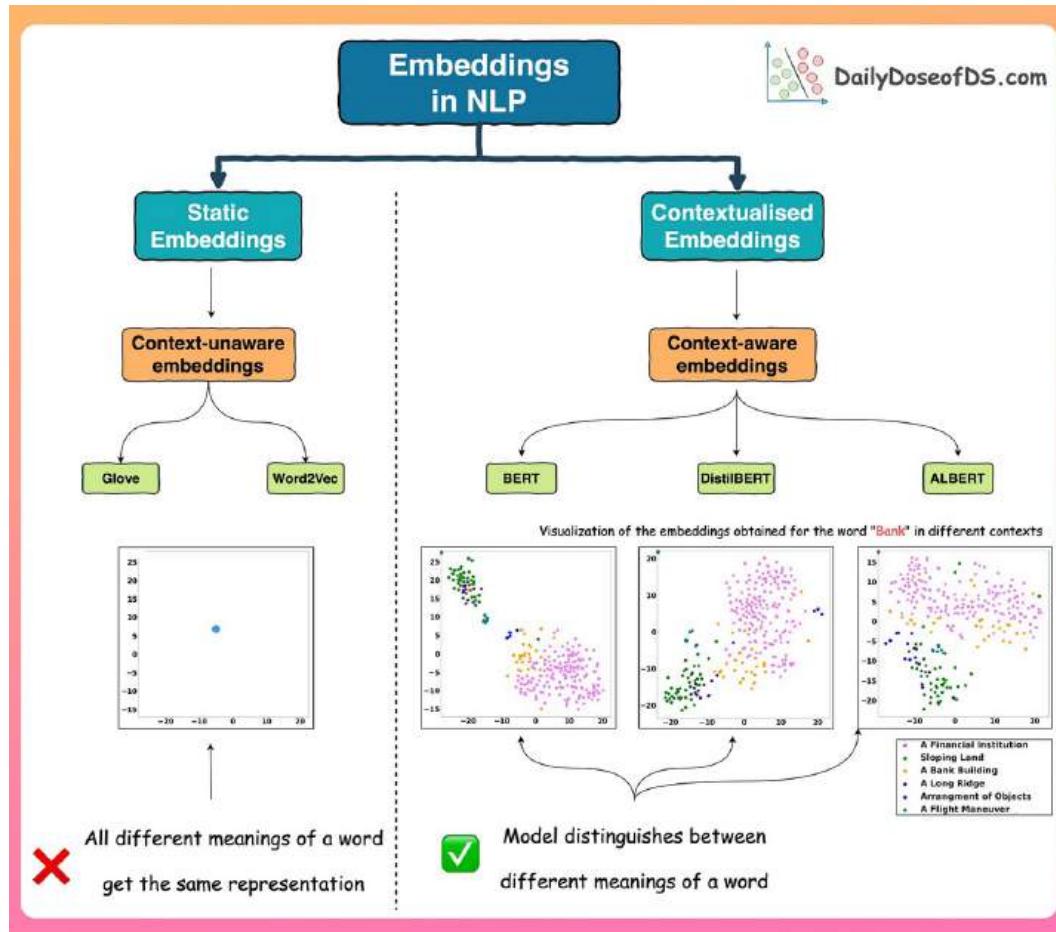


- **Variance:** Measure of the spread of data points around the mean.
- **Validation Curve:** Graph showing how model performance changes with different hyperparameter values.
- **Vanishing Gradient:** Issue in deep neural networks when gradients become very small during training.
- W:
  - **Word embedding:** Representation of words as dense vectors in natural language processing.
  - **Word cloud:** Visualization of text data where word frequency is represented through the size of the word.
  - **Weights:** Parameters that are learned by a machine learning model during training.
- X:
  - **XGBoost:** Extreme Gradient Boosting, a popular gradient boosting library.
  - **XLNet:** Generalized Autoregressive Pretraining of Transformers, a language model.
- Y:
  - **YOLO (You Only Look Once):** Real-time object detection system.
  - **Yellowbrick:** Python library for machine learning visualization and diagnostic tools.
- Z:
  - **Z-score:** Standardized value representing how many standard deviations a data point is from the mean.
  - **Z-test:** Statistical test used to compare a sample mean to a known population mean.
  - **Zero-shot learning:** Machine learning method where a model can recognize new classes without seeing explicit examples during training.

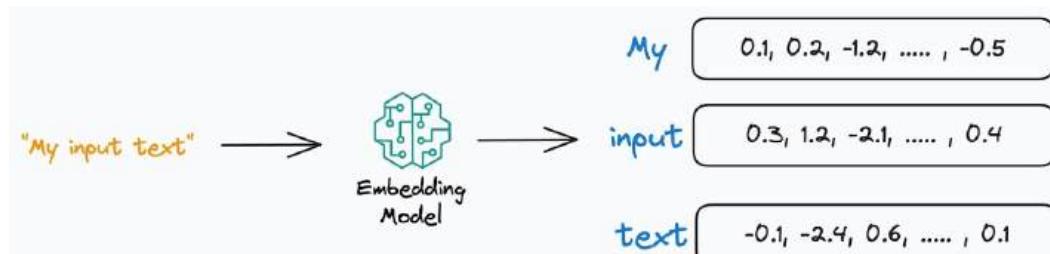
👉 Over to you: Of course, a lot has been left out here. As an exercise, can you add more terms to this?



# The Limitation of Static Embeddings Which Made Them Obsolete



To build models for language-oriented tasks, it is crucial to generate numerical representations (or vectors) for words.



Text to embedding overview



This allows words to be processed and manipulated mathematically and perform various computational operations on words.

The objective of embeddings is to capture semantic and syntactic relationships between words. This helps machines understand and reason about language more effectively.

In the pre-Transformers era, this was primarily done using pre-trained static embeddings.

Essentially, someone would train and release these word embeddings for, say, 100k, or 200k common words using deep learning.

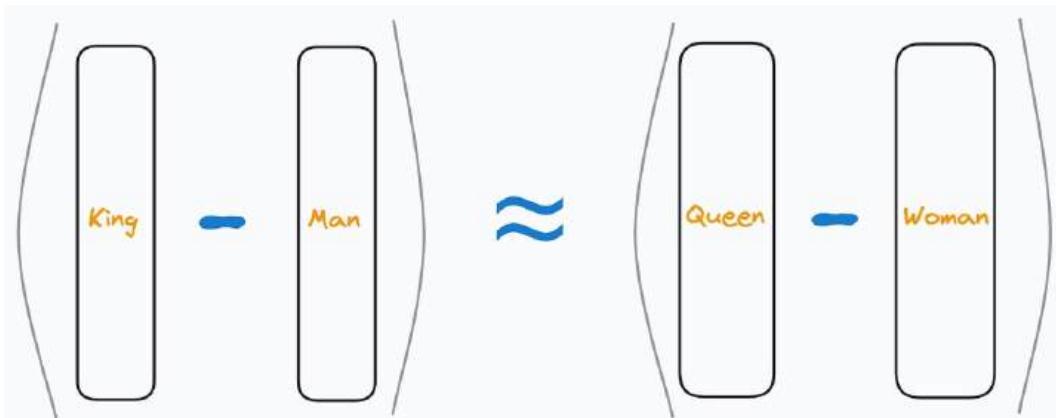
...and other researchers may utilize those embeddings in their projects.

The most popular models at that time (around 2013-2018ish) were:

- Glove
- Word2Vec
- FastText, etc.

These embeddings genuinely showed some promising results in learning the relationships between words.

For instance, running the vector operation  $(\text{King} - \text{Man}) + \text{Woman}$  would return a vector near the word “Queen”.



(King-Man) approximates to (Queen - Woman)

So while these did capture relative representations of words, there was a major limitation.

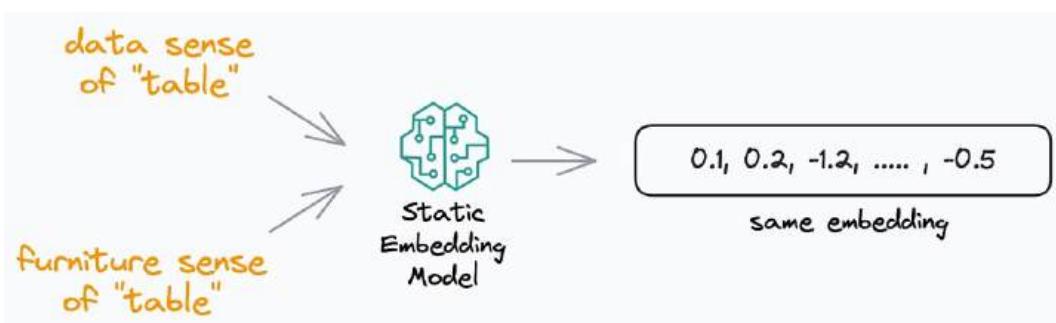
Consider the following two sentences:

- “Convert this data into a **table** in Excel.”
- “Put this bottle on the **table**.”

The word “**table**” conveys two entirely different meanings in the two sentences.

- The first sentence refers to a “**data**” specific sense of the word “table”.
- The second sentence refers to a “**furniture**” specific sense of the word “table”.

Yet, static embedding models assigned them the same representation.



Same embedding for different usages of a word



Thus, these embeddings didn't consider that a word may have different usages in different contexts.

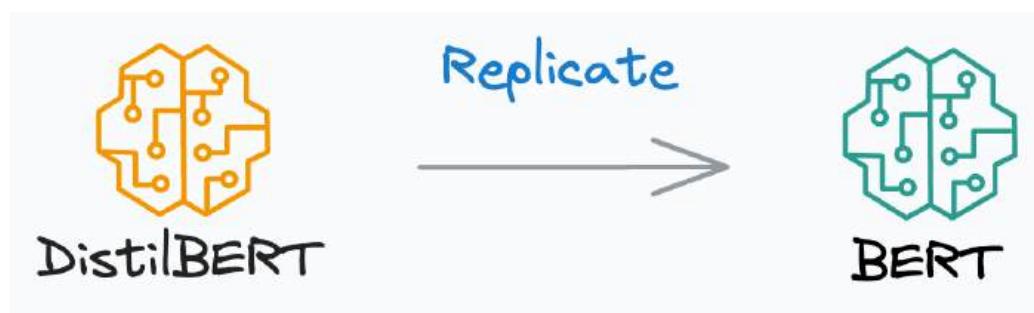
But this changed in the Transformer era, which resulted in contextualized embeddings models powered by Transformers, such as:

- **BERT**: A language model trained using two techniques:



### BERT pre-training

- Masked Language Modeling (MLM): Predict a missing word in the sentence, given the surrounding words.
- Next Sentence Prediction (NSP).
- **DistilBERT**: A simple, effective, and lighter version of BERT which is around 40% smaller:



Training DistilBERT



- Utilizes a common machine learning strategy called student-teacher theory.
- Here, the student is the distilled version of BERT, and the teacher is the original BERT model.
- The student model is supposed to replicate the teacher model's behavior.
- **ALBERT: A Lite BERT (ALBERT).** Uses a couple of optimization strategies to reduce the size of BERT:
  - Eliminates one-hot embeddings at the initial layer by projecting the words into a low-dimensional space.
  - Shares the weights across all the network segments of the Transformer model.

These were capable of generating context-aware representations, thanks to their self-attention mechanism.

This would allow embedding models to dynamically generate embeddings for a word based on the context they were used in.

As a result, if a word would appear in a different context, the model would get a different representation.

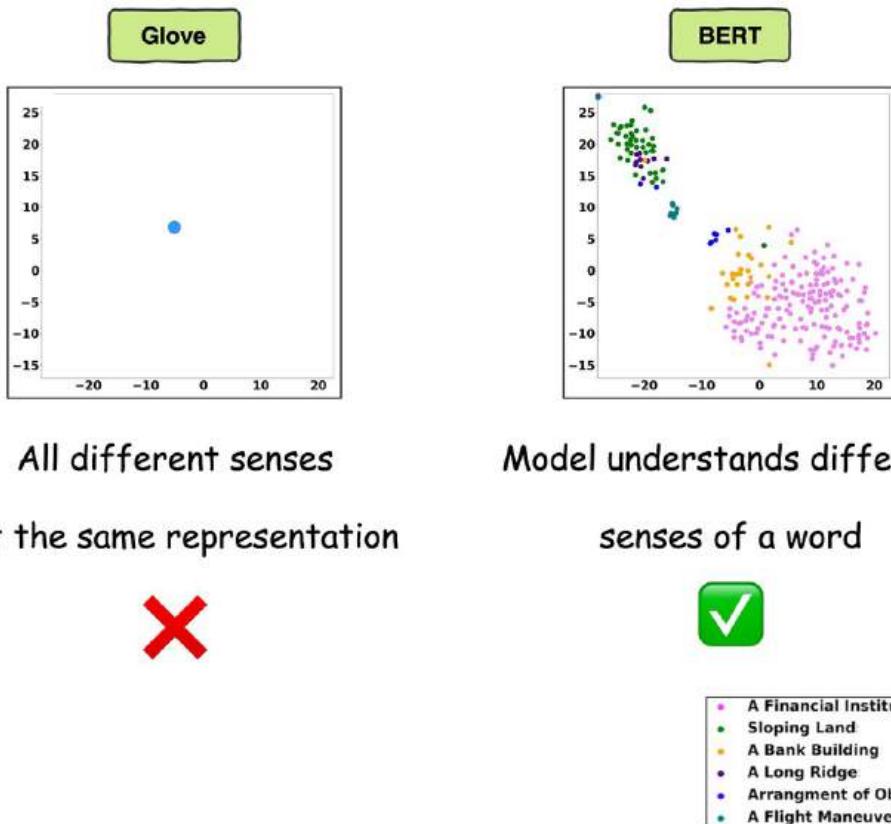
This is precisely depicted in the image below for different uses of the word “Bank”.

For visualization purposes, the embeddings have been projected into 2d space using t-SNE.



Visualization of the embeddings obtained for

the word "**Bank**" in different contexts



Glove vs. BERT on understanding different senses of a word

The static embedding models — Glove and Word2Vec produce the same embedding for different usages of a word.

However, contextualized embedding models don't.

In fact, contextualized embeddings understand the different meanings/senses of the word “Bank”:

- A financial institution
- Sloping land
- A Long Ridge, and more.

Different senses were taken from Princeton's Wordnet database here: [WordNet](#).



As a result, they addressed the major limitations of static embedding models.

For those who wish to learn in more detail, I published a couple of research papers on this intriguing topic:

- [Interpretable Word Sense Disambiguation with Contextualized Embeddings.](#)
- [A Comparative Study of Transformers on Word Sense Disambiguation.](#)

These papers discuss the strengths and limitations of many contextualized embedding models in detail.

👉 Over to you: What do you think were some other pivotal moments in NLP research?



# An Overlooked Technique To Improve KMeans Run-time

The standard KMeans algorithm involves a brute-force approach.

To recall, KMeans is trained as follows:

- Initialize centroids
- Find the nearest centroid for each point
- Reassign centroids
- Repeat until convergence

As a result, the run-time of KMeans depends on four factors:

- Number of iterations (i)
- Number of samples (n)
- Number of clusters (k)
- Number of features (d)

$$O(i * n * k * d)$$

In fact, you can add another factor here — “the repetition factor”, where, we run the whole clustering repeatedly to avoid convergence issues.

But we are ignoring that for now.

While we cannot do much about the first three, reducing the number of features is quite possible, yet often overlooked.

Sparse Random Projection is an efficient projection technique for reducing dimensionality.

Some of its properties are:

- It projects the original data to lower dimensions using a sparse random matrix.



- It provides similar embedding quality while being memory and run-time efficient.
- The similarity and dissimilarity between points are well preserved.

The visual below shows the run-time comparison of KMeans on:

- Standard high-dimensional data, vs.
- Data projected to lower dimensions using Sparse Random Projection.

**Slow KMeans Clustering**

KMeans

```
# High dimensional data
X, y = make_blobs(n_features = 500,
                    centers = 5,
                    n_samples = 10000)

clf = KMeans(n_clusters=5).fit(X)
# Run-time: 251 ms

→ Silhouette Score: 0.82
→ Accuracy: 60%
```

✗

Similar performance and 10x Faster

✓

**KMeans With Data Projection**

```
from sklearn.random_projection
import SparseRandomProjection as SRP

# Reduce input dimensions to 4
srp = SRP(n_components=4)
X_srp = srp(X)

# Fit KMeans on projected data
clf = KMeans(n_clusters=5).fit(X_srp)
# Run-time: 25 ms

→ Silhouette Score: 0.82
→ Accuracy: 60%
```



As shown, Sparse Random Projection provides:

- Similar performance, and
- a MASSIVE run-time improvement of 10x.

This can be especially useful in high-dimensional datasets.

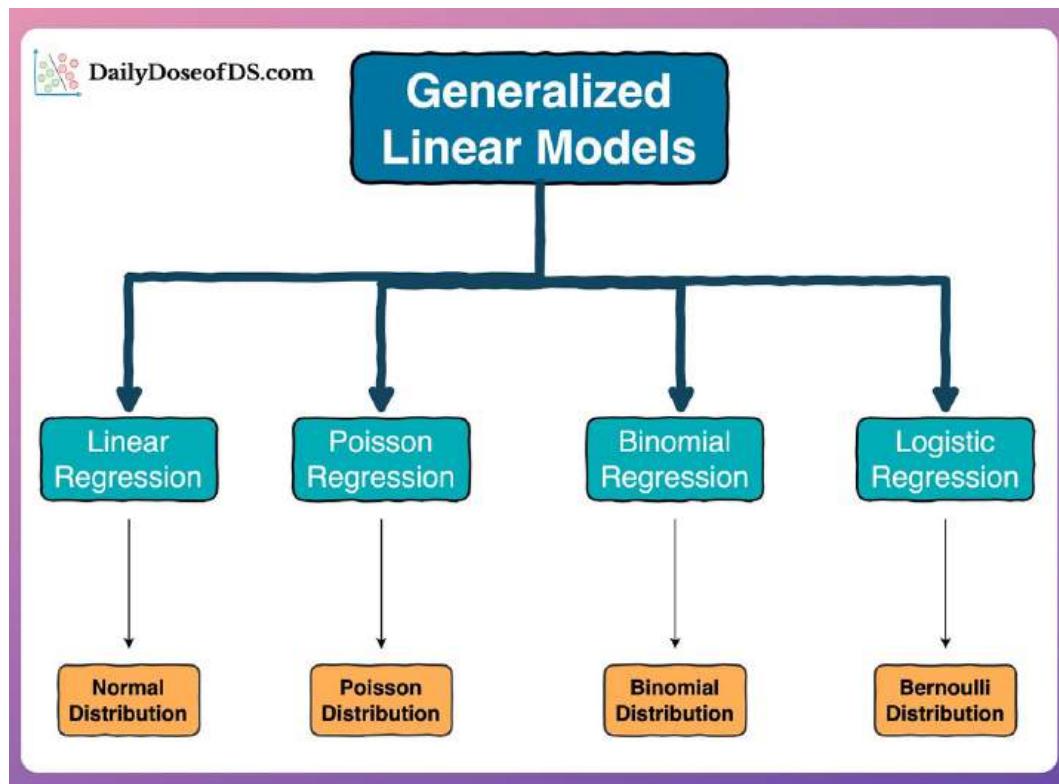
Get started with Sparse Random Projections here: [Sklearn Docs](#).

For more info, here's the paper that discussed it: [Very Sparse Random Projections](#).

👉 Over to you: What are some other ways to improve KMeans run-time?



# The Most Underrated Skill in Training Linear Models



[Yesterday's post on Poisson regression](#) was appreciated by many of you.

Today, I want to build on that and help you cultivate what I think is one of the MOST overlooked and underappreciated skills in developing linear models.

I can guarantee that harnessing this skill will give you so much clarity and intuition in the modeling stages.

But let's do a quick recap of yesterday's post before we proceed.

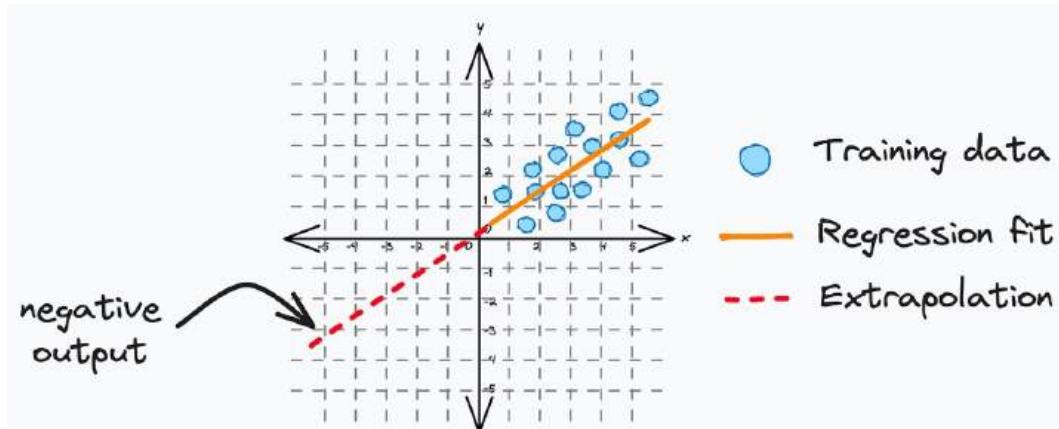
---

## Recap

Having a non-negative response in the training data does not stop linear regression from outputting negative values.



Essentially, you can always extrapolate the regression fit for some inputs to get a negative output.



Extrapolation of the linear regression fit

While this is not an issue per se, negative outputs may not make sense in cases where you can never have such outcomes.

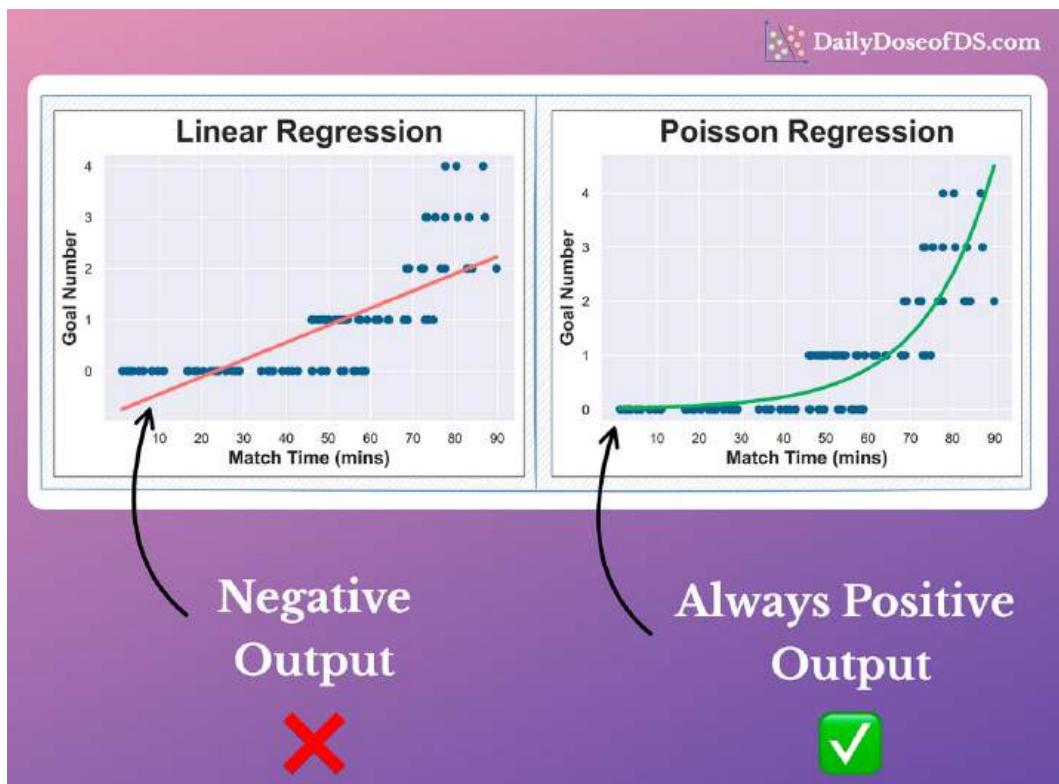
For instance:

- Predicting the number of calls received.
- Predicting the number of cars sold in a year, etc.

More specifically, the issue arises when modeling a count-based response, where a negative output wouldn't make sense.

In such cases, Poisson regression often turns out to be a more suitable linear model than linear regression.

This is evident from the image below:



Please read yesterday's post for in-depth info: [Poisson Regression: The Robust Extension of Linear Regression](#).

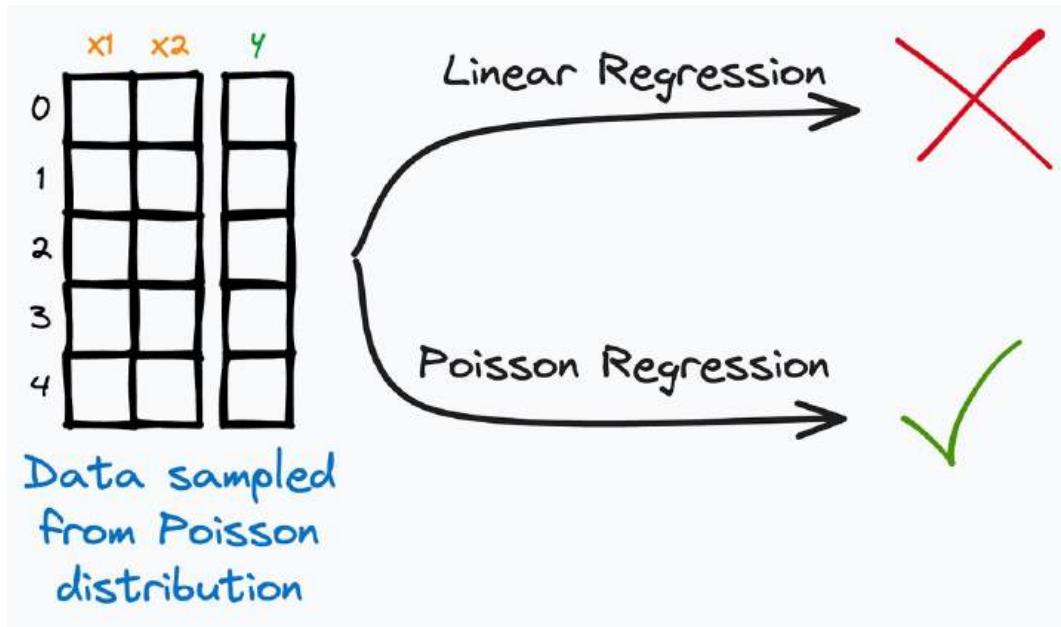
---

Here, I want you to understand that Poisson regression is no magic.

It's just that, in this specific use case, the data generation process didn't perfectly align with what linear regression is designed to handle.

In other words, as soon as we trained a linear regression model above, we inherently assumed that the data was sampled from a normal distribution.

But that was not true in this case.

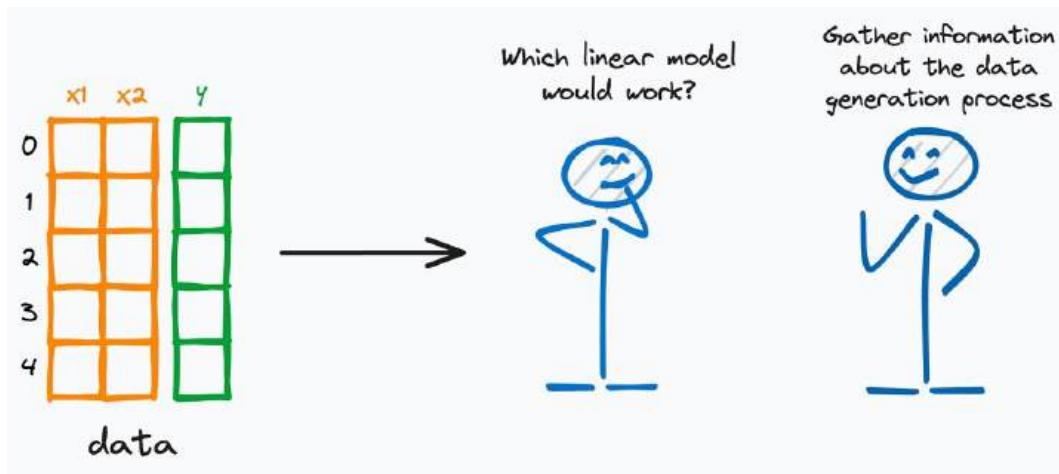


Instead, it came from a Poisson distribution, which is why Poisson regression worked better.

Thus, the takeaway is that whenever you train linear models, **always always and always** think about the data generation process.

This goes like this:

- Okay, I have this data.
- I want to fit a linear model through it.
- What information do I get from the label about the **data generation process** that can help me select an appropriate linear model?



You'd start appreciating the importance of data generation when you'd realize that literally EVERY extension of linear regression (or a member of the generalized linear model family) stems from altering the data generation process.

For instance:

- If the data generation process involves a **Normal distribution** → you get linear regression.
- If the data has only positive integers in the response variable, maybe it came from a **Poisson distribution** → and this gives us Poisson regression. This is precisely what we discussed yesterday.
- If the data has only two targets — 0 and 1, maybe it was generated using **Bernoulli distribution** → and this gives rise to logistic regression.
- If the data has finite and fixed categories (0, 1, 2,...n), then this hints towards **Binomial distribution** → and we get Binomial regression.

See...

Every linear model makes an assumption and is then derived from an underlying data generation process.



Linear Regression -----> Assumes Normal distribution

Logistic Regression -----> Assumes Bernoulli distribution

Poisson Regression -----> Assumes Poisson distribution

Binomial Regression -----> Assumes Binomial distribution

Thus, developing a habit of stepping back and thinking about the data generation process will give you so much clarity in the modeling stages.

I am confident this will help you get rid of that annoying and helpless habit of relentlessly using a specific sklearn algorithm without truly knowing why you are using it.

Consequently, you'd know which algorithm to use and, most importantly, **why**.

This improves your credibility as a data scientist and allows you to approach data science problems with intuition and clarity rather than hit-and-trial.

Hope you learned something new.



# Poisson Regression: The Robust Extension of Linear Regression

## Shortcomings of Linear Regression

DailyDoseofDS.com

**Linear Regression**

Goal Number

Match Time (mins)

Count-based Data  
Linear Regression fit

**Poisson Regression**

Goal Number

Match Time (mins)

Count-based Data  
Poisson Regression fit

**✗ Linear Regression**

- Unsuitable for modeling count data
- May return negative output for some inputs
- Errors must follow a symmetric distribution

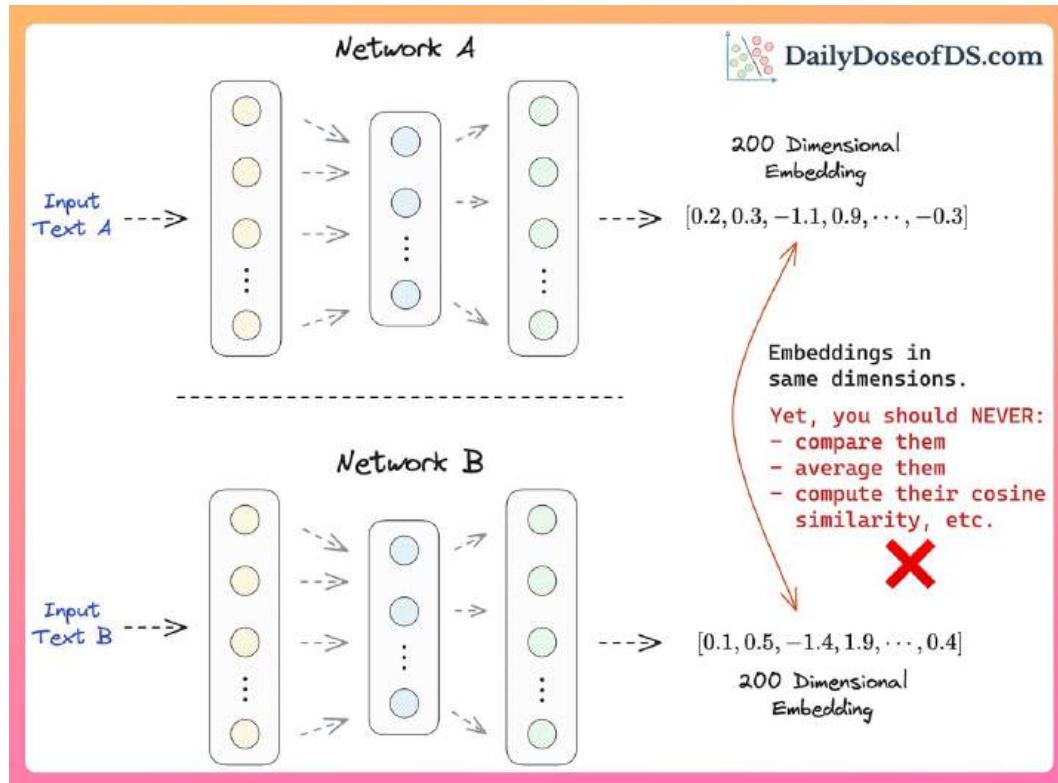
**✓ Poisson Regression**

- Specifically designed to model count data
- All outputs are non-negative
- Works well even if errors are asymmetrically distributed

Read the full issue here: <https://www.blog.dailydoseofds.com/p/poisson-regression-the-robust-extension>

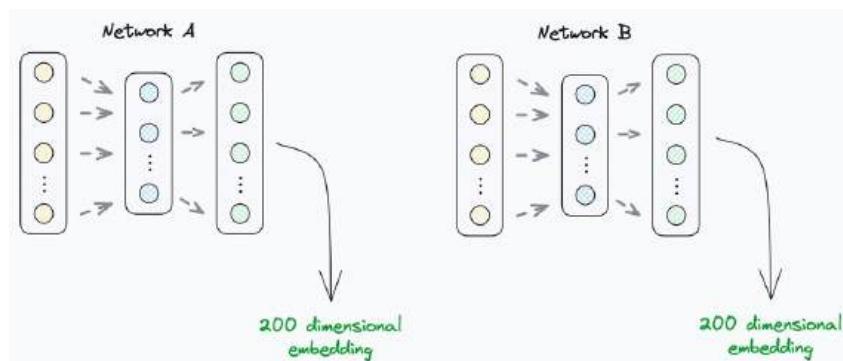


# The Biggest Mistake ML Folks Make When Using Multiple Embedding Models



Imagine you have two different models (or sub-networks) in your whole ML pipeline.

Both generate a representation/embedding of the input in the same dimensions (say, 200).





These could also be pre-trained models used to generate embeddings—Bert, XLNet, etc.

Here, many folks get tempted to make them interact.

They would:

- compare these representations
- compute their Euclidean distance
- compute their cosine similarity, and more.

The rationale is that the representations have the same dimensions. Thus, they can seamlessly interact.

However, that is NOT true, and you should NEVER do that.

Why?

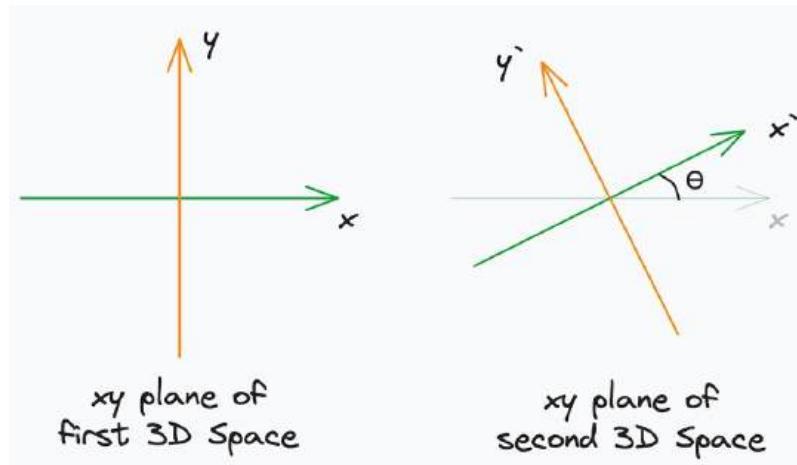
Even though these embeddings have the same length, they are out of space.

Out of space means that their axes are not aligned.

To simplify, imagine both embeddings were in a 3D space.

Now, assume that their z-axes are aligned.

But the x-axis of one of them is at an angle to the x-axis of the other.



As a result, coordinates from these two spaces are no longer comparable.

Similarly, comparing the embeddings from two networks would inherently assume that all axes are perfectly aligned.

But this is highly unlikely because there are infinitely many ways axes may orient relative to each other.

Thus, the representations can NEVER be compared, unless they are generated by the same model.

This is a mistake that may cause some serious trouble in your ML pipeline.

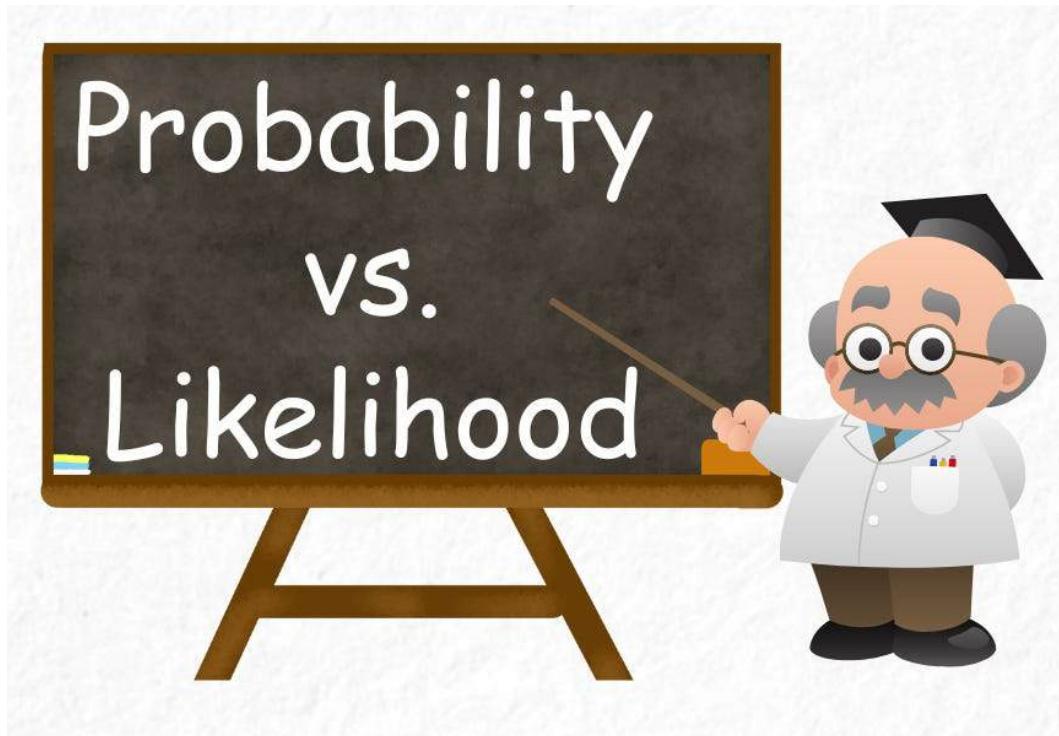
Also, it can easily go unnoticed, so it is immensely crucial to be aware of this.

Hope that helped!

👉 Over to you: How do you typically handle embeddings from multiple models?



# Probability and Likelihood Are Not Meant To Be Used Interchangeably



In data science and statistics, folks often use “probability” and “likelihood” interchangeably.

However, Likelihood and probability DO NOT convey the same meaning.

And the misunderstanding is somewhat understandable, given that they carry similar meanings in our regular language.

While writing today’s newsletter, I searched for their meaning in the Cambridge Dictionary.

Here’s what it says:

- **Probability:** the level of possibility of something happening or being true/ ([Source](#))
- **Likelihood:** the chance that something will happen. ([Source](#))



It amused me that “likelihood” is the only synonym of “probability”.

## probability

noun [ C or U ]

UK / prob.ə'bil.e.ti/ US / prɑ:bə'bil.e.ti/

Add to word list

C1

the level of possibility of something happening or being true:

- *What is the probability of winning?*
- *The probability of getting all the answers correct is about one in ten.*
- *There's a high/strong probability (that) (= it is very likely that) she'll be here.*
- *Until yesterday, the project was just a possibility, but now it has become a real probability (= it is likely to happen).*

Synonym  
likelihood



Anyway.

In my opinion, it is crucial to understand that probability and Likelihood convey very different meanings in data science and statistics.

Let's understand!

---

**Probability** is used in contexts where you wish to know the possibility/odds of an event.

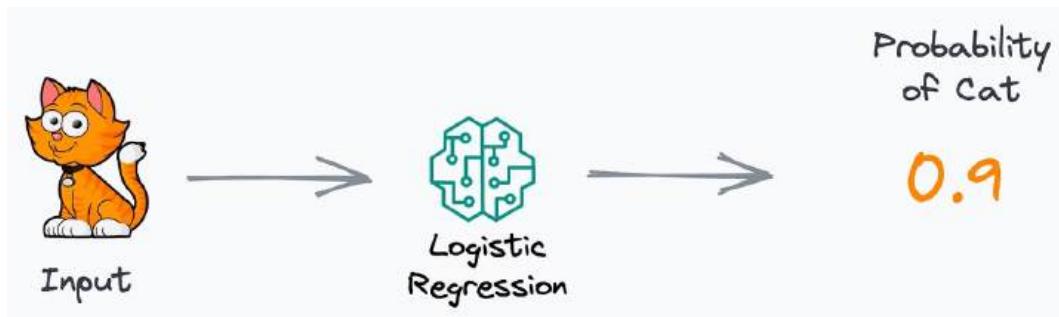
For instance, what is the:

- Probability of obtaining an even number in a die roll?
- Probability of drawing an ace of diamonds from a deck?
- and so on...

When translated to ML, probability can be thought of as:



- What is the probability that a transaction is fraud?
- What is the probability that an image depicts a cat?
- and so on...



Essentially, many classification models, like logistic regression or a neural network, etc., assign the **probability** of a specific label to an input.

When calculating probability, the model's parameters are known. Also, we assume that they are trustworthy.

For instance, to determine the probability of a head in a coin toss, we assume and trust that it is a fair coin.

---

**Likelihood**, on the other hand, is about explaining events that have already occurred.

Unlike probability (where parameters are known and assumed to be trustworthy)...

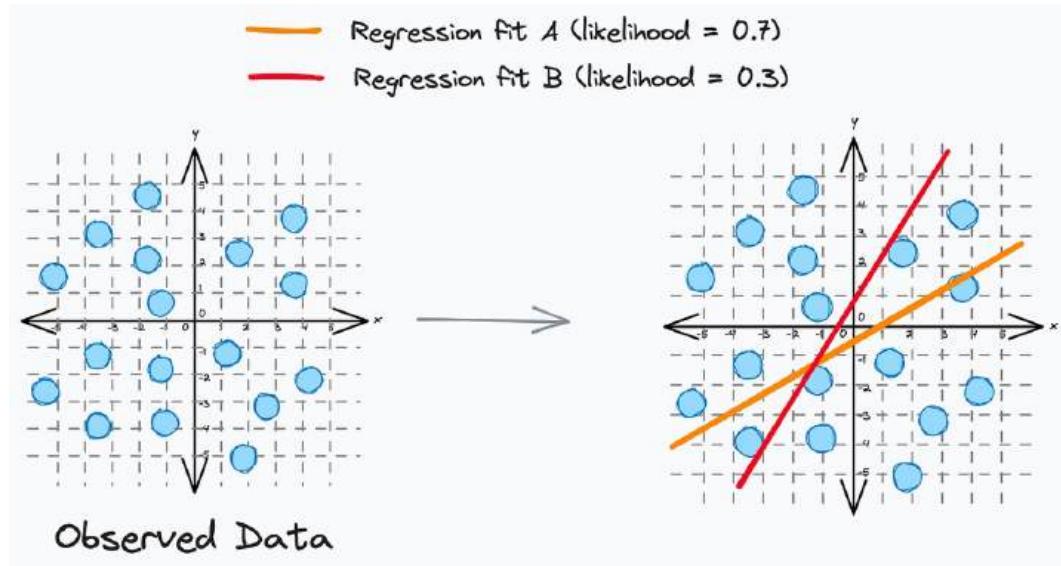
Likelihood helps us determine if we can trust the parameters in a model based on the observed data.

Here's how we use it in the context of data science and machine learning.

Assume you have collected some 2D data and wish to fit a straight line with two parameters — slope ( $m$ ) and intercept ( $c$ ).



Here, Likelihood is defined as the support provided by a data point for some particular parameter values in your model.

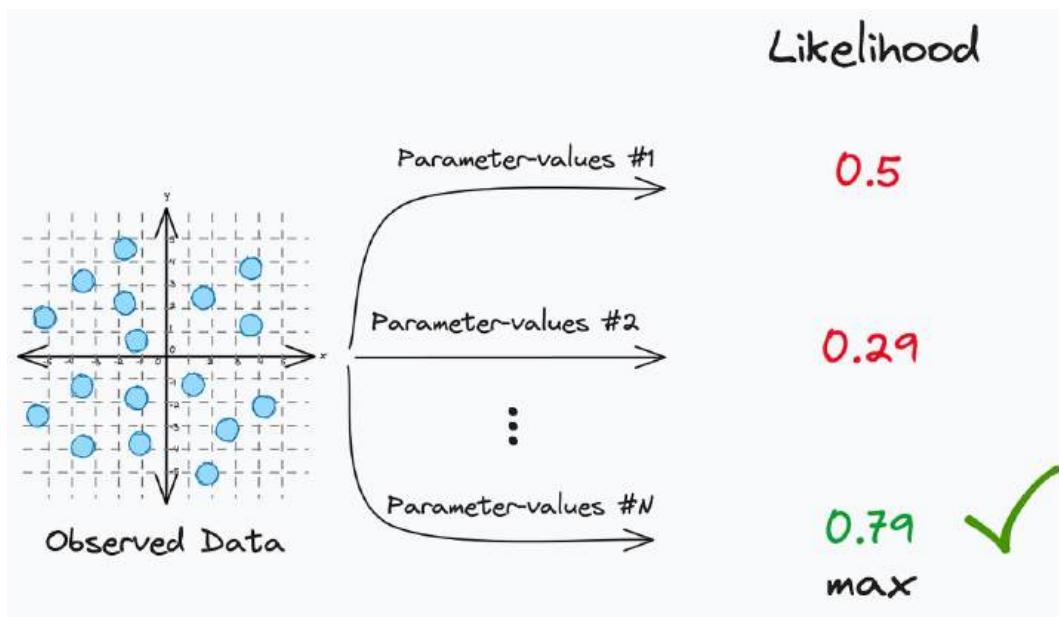


Here, you will ask questions like:

- If I model this data with the parameters:
  - $m=2$  and  $c=1$ , what is the Likelihood of observing the data?
  - $m=3$  and  $c=2$ , what is the Likelihood of observing the data?
  - and so on...

The above formulation popularly translates into the maximum likelihood estimation (MLE).

In maximum likelihood estimation, you have some observed data and you are trying to determine the specific set of parameters ( $\theta$ ) that maximize the Likelihood of observing the data.



Using the term “likelihood” is like:

- I have a possible explanation for my data. (In the above illustration, “explanation” can be thought of as the parameters you are trying to determine)
- How well my explanation explains what I’ve already observed? This is precisely quantified using Likelihood.

For instance:

- **Observation:** The outcomes of 10 coin tosses are “HHHHHHHHTH”.
- **Explanation:** I think it is a fair coin ( $p=0.5$ ).
- What is the Likelihood that my explanation is true based on the observed data?

---

To summarize...

It is immensely important to understand that in data science and statistics, Likelihood and probability DO NOT convey the same meaning.



As explained above, they are pretty different.

In Probability:

- We determine the possibility of an event.
- We know the parameters associated with the event and assume them to be trustworthy.

In Likelihood:

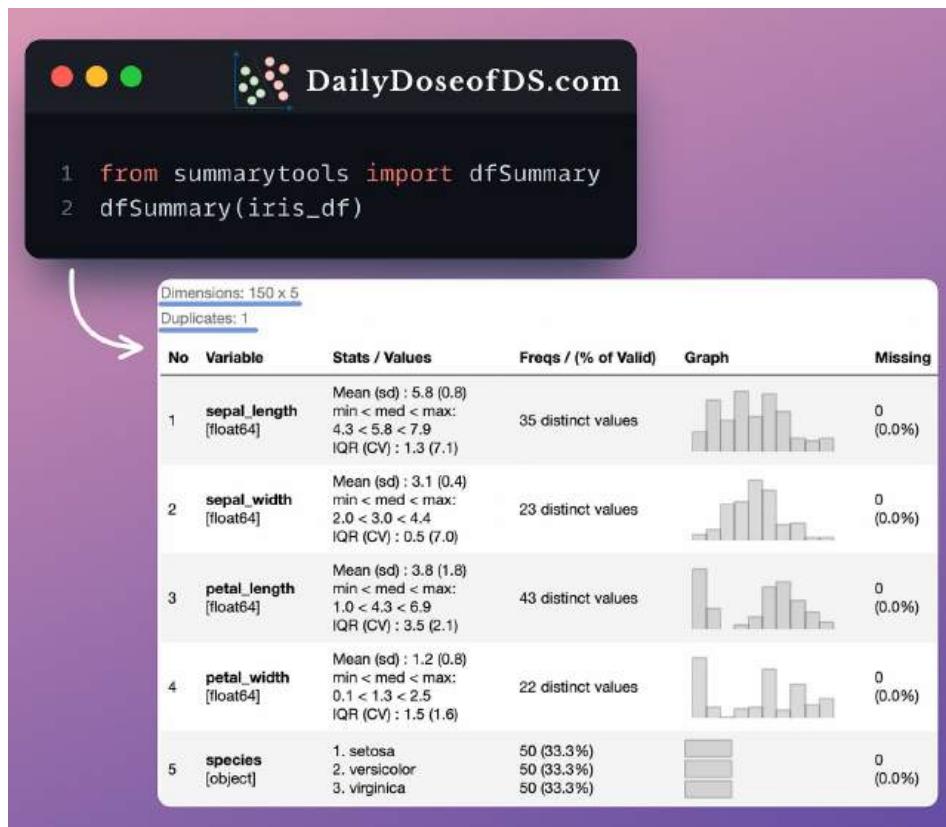
- We have some observations.
- We have an explanation (or parameters).
- Likelihood helps us quantify whether the explanation is trustworthy.

Hope that helped!

👉 Over to you: I would love to hear your explanation of probability and Likelihood.



# SummaryTools: A Richer Alternative To Pandas' Describe Method.



Summarytools is a Jupyter-based tool that provides a standardized and comprehensive data summary.

By invoking a single function, you can generate the above report in seconds.

This includes:

- column statistics,
- data type info,
- frequency,
- distribution chart, and
- and missing stats.

Get started with Summary Tools here: [Summary Tools](#).



# 40 NumPy Methods That Data Scientists Use 95% of the Time

40 NumPy Methods That Data Scientists Use 95% of the Time		DailyDoseofDS.com																																																		
<b>NumPy Array Creation Methods</b>		<b>Mathematical Operations</b>																																																		
<table border="1"><thead><tr><th>Method</th><th>Description</th></tr></thead><tbody><tr><td><code>np.array(&lt;list&gt;)</code></td><td>NumPy array from Python list</td></tr><tr><td><code>np.array(&lt;list-of-lists&gt;)</code></td><td>NumPy array from list of lists</td></tr><tr><td><code>np.array(&lt;pandas-series&gt;)</code></td><td>NumPy array from PD Series</td></tr><tr><td><code>df.values</code></td><td>NumPy array from DataFrame</td></tr><tr><td><code>np.zeros(&lt;size&gt;)</code></td><td>NumPy array of all zeros</td></tr><tr><td><code>np.ones(&lt;size&gt;)</code></td><td>NumPy array of all ones</td></tr><tr><td><code>np.eye(&lt;size&gt;)</code></td><td>Identity NumPy array</td></tr><tr><td><code>np.arange(&lt;start&gt;, &lt;stop&gt;, &lt;step&gt;)</code></td><td>Equally spaced NumPy array with specific step</td></tr><tr><td><code>np.linspace(&lt;start&gt;, &lt;stop&gt;, &lt;count&gt;)</code></td><td>Equally spaced NumPy array with specific size</td></tr><tr><td><code>np.random.randint(&lt;low&gt;, &lt;high&gt;, &lt;size&gt;)</code></td><td>NumPy array of random ints</td></tr><tr><td><code>np.random.random(&lt;size&gt;)</code></td><td>NumPy array of random floats</td></tr></tbody></table>		Method	Description	<code>np.array(&lt;list&gt;)</code>	NumPy array from Python list	<code>np.array(&lt;list-of-lists&gt;)</code>	NumPy array from list of lists	<code>np.array(&lt;pandas-series&gt;)</code>	NumPy array from PD Series	<code>df.values</code>	NumPy array from DataFrame	<code>np.zeros(&lt;size&gt;)</code>	NumPy array of all zeros	<code>np.ones(&lt;size&gt;)</code>	NumPy array of all ones	<code>np.eye(&lt;size&gt;)</code>	Identity NumPy array	<code>np.arange(&lt;start&gt;, &lt;stop&gt;, &lt;step&gt;)</code>	Equally spaced NumPy array with specific step	<code>np.linspace(&lt;start&gt;, &lt;stop&gt;, &lt;count&gt;)</code>	Equally spaced NumPy array with specific size	<code>np.random.randint(&lt;low&gt;, &lt;high&gt;, &lt;size&gt;)</code>	NumPy array of random ints	<code>np.random.random(&lt;size&gt;)</code>	NumPy array of random floats	<table border="1"><thead><tr><th>Method</th><th>Description</th></tr></thead><tbody><tr><td><code>np.sin(&lt;np-array&gt;)</code></td><td rowspan="3">Trigonometric Functions</td></tr><tr><td><code>np.cos(&lt;np-array&gt;)</code></td></tr><tr><td><code>np.tan(&lt;np-array&gt;)</code></td></tr><tr><td><code>np.floor(&lt;np-array&gt;)</code></td><td>Element-wise floor value</td></tr><tr><td><code>np.ceil(&lt;np-array&gt;)</code></td><td>Element-wise ceiling value</td></tr><tr><td><code>np.rint(&lt;np-array&gt;)</code></td><td>Round to nearest int</td></tr><tr><td><code>np.round(&lt;np-array&gt;, &lt;decimal-places&gt;)</code></td><td>Round to decimal places</td></tr><tr><td><code>np.exp(&lt;np-array&gt;)</code></td><td>Element-wise exponent</td></tr><tr><td><code>np.log(&lt;np-array&gt;)</code></td><td>Element-wise logarithm</td></tr><tr><td><code>np.sqrt(&lt;np-array&gt;)</code></td><td>Element-wise square root</td></tr><tr><td><code>np.sum(&lt;np-array&gt;, &lt;axis&gt;)</code></td><td>Sum along an axis</td></tr><tr><td><code>np.mean(&lt;np-array&gt;, &lt;axis&gt;)</code></td><td>Mean along an axis</td></tr><tr><td><code>np.std(&lt;np-array&gt;, &lt;axis&gt;)</code></td><td>Std. dev along an axis</td></tr></tbody></table>	Method	Description	<code>np.sin(&lt;np-array&gt;)</code>	Trigonometric Functions	<code>np.cos(&lt;np-array&gt;)</code>	<code>np.tan(&lt;np-array&gt;)</code>	<code>np.floor(&lt;np-array&gt;)</code>	Element-wise floor value	<code>np.ceil(&lt;np-array&gt;)</code>	Element-wise ceiling value	<code>np.rint(&lt;np-array&gt;)</code>	Round to nearest int	<code>np.round(&lt;np-array&gt;, &lt;decimal-places&gt;)</code>	Round to decimal places	<code>np.exp(&lt;np-array&gt;)</code>	Element-wise exponent	<code>np.log(&lt;np-array&gt;)</code>	Element-wise logarithm	<code>np.sqrt(&lt;np-array&gt;)</code>	Element-wise square root	<code>np.sum(&lt;np-array&gt;, &lt;axis&gt;)</code>	Sum along an axis	<code>np.mean(&lt;np-array&gt;, &lt;axis&gt;)</code>	Mean along an axis	<code>np.std(&lt;np-array&gt;, &lt;axis&gt;)</code>	Std. dev along an axis
Method	Description																																																			
<code>np.array(&lt;list&gt;)</code>	NumPy array from Python list																																																			
<code>np.array(&lt;list-of-lists&gt;)</code>	NumPy array from list of lists																																																			
<code>np.array(&lt;pandas-series&gt;)</code>	NumPy array from PD Series																																																			
<code>df.values</code>	NumPy array from DataFrame																																																			
<code>np.zeros(&lt;size&gt;)</code>	NumPy array of all zeros																																																			
<code>np.ones(&lt;size&gt;)</code>	NumPy array of all ones																																																			
<code>np.eye(&lt;size&gt;)</code>	Identity NumPy array																																																			
<code>np.arange(&lt;start&gt;, &lt;stop&gt;, &lt;step&gt;)</code>	Equally spaced NumPy array with specific step																																																			
<code>np.linspace(&lt;start&gt;, &lt;stop&gt;, &lt;count&gt;)</code>	Equally spaced NumPy array with specific size																																																			
<code>np.random.randint(&lt;low&gt;, &lt;high&gt;, &lt;size&gt;)</code>	NumPy array of random ints																																																			
<code>np.random.random(&lt;size&gt;)</code>	NumPy array of random floats																																																			
Method	Description																																																			
<code>np.sin(&lt;np-array&gt;)</code>	Trigonometric Functions																																																			
<code>np.cos(&lt;np-array&gt;)</code>																																																				
<code>np.tan(&lt;np-array&gt;)</code>																																																				
<code>np.floor(&lt;np-array&gt;)</code>	Element-wise floor value																																																			
<code>np.ceil(&lt;np-array&gt;)</code>	Element-wise ceiling value																																																			
<code>np.rint(&lt;np-array&gt;)</code>	Round to nearest int																																																			
<code>np.round(&lt;np-array&gt;, &lt;decimal-places&gt;)</code>	Round to decimal places																																																			
<code>np.exp(&lt;np-array&gt;)</code>	Element-wise exponent																																																			
<code>np.log(&lt;np-array&gt;)</code>	Element-wise logarithm																																																			
<code>np.sqrt(&lt;np-array&gt;)</code>	Element-wise square root																																																			
<code>np.sum(&lt;np-array&gt;, &lt;axis&gt;)</code>	Sum along an axis																																																			
<code>np.mean(&lt;np-array&gt;, &lt;axis&gt;)</code>	Mean along an axis																																																			
<code>np.std(&lt;np-array&gt;, &lt;axis&gt;)</code>	Std. dev along an axis																																																			
<b>NumPy Array Manipulation Methods</b>		<b>Matrix and Vector Operations</b>																																																		
<table border="1"><thead><tr><th>Method</th><th>Description</th></tr></thead><tbody><tr><td><code>array.reshape(&lt;new-shape&gt;)</code></td><td>Reshape NumPy Array</td></tr><tr><td><code>array.transpose() OR array.T</code></td><td>Transpose NumPy Array</td></tr><tr><td><code>np.concatenate(&lt;np-arrays&gt;, &lt;axis&gt;)</code></td><td>Concatenate NumPy Arrays</td></tr><tr><td><code>np.flatten(&lt;Nd-np-array&gt;)</code></td><td>Flatten a NumPy Array</td></tr><tr><td><code>np.unique(&lt;np-array&gt;, &lt;axis&gt;)</code></td><td>Find unique elements</td></tr><tr><td><code>array.tolist()</code></td><td>NumPy Array to List</td></tr></tbody></table>		Method	Description	<code>array.reshape(&lt;new-shape&gt;)</code>	Reshape NumPy Array	<code>array.transpose() OR array.T</code>	Transpose NumPy Array	<code>np.concatenate(&lt;np-arrays&gt;, &lt;axis&gt;)</code>	Concatenate NumPy Arrays	<code>np.flatten(&lt;Nd-np-array&gt;)</code>	Flatten a NumPy Array	<code>np.unique(&lt;np-array&gt;, &lt;axis&gt;)</code>	Find unique elements	<code>array.tolist()</code>	NumPy Array to List	<table border="1"><thead><tr><th>Method</th><th>Description</th></tr></thead><tbody><tr><td><code>np.dot(&lt;np-array1&gt;, &lt;np-array2&gt;)</code></td><td>Dot Product</td></tr><tr><td><code>np.matmul(&lt;np-array1&gt;, &lt;np-array2&gt;)</code></td><td rowspan="4">Matrix Multiplication <math>\text{np-array1} @ \text{np-array2}</math></td></tr><tr><td><code>np.linalg.norm(&lt;np-array&gt;)</code></td></tr></tbody></table>	Method	Description	<code>np.dot(&lt;np-array1&gt;, &lt;np-array2&gt;)</code>	Dot Product	<code>np.matmul(&lt;np-array1&gt;, &lt;np-array2&gt;)</code>	Matrix Multiplication $\text{np-array1} @ \text{np-array2}$	<code>np.linalg.norm(&lt;np-array&gt;)</code>																													
Method	Description																																																			
<code>array.reshape(&lt;new-shape&gt;)</code>	Reshape NumPy Array																																																			
<code>array.transpose() OR array.T</code>	Transpose NumPy Array																																																			
<code>np.concatenate(&lt;np-arrays&gt;, &lt;axis&gt;)</code>	Concatenate NumPy Arrays																																																			
<code>np.flatten(&lt;Nd-np-array&gt;)</code>	Flatten a NumPy Array																																																			
<code>np.unique(&lt;np-array&gt;, &lt;axis&gt;)</code>	Find unique elements																																																			
<code>array.tolist()</code>	NumPy Array to List																																																			
Method	Description																																																			
<code>np.dot(&lt;np-array1&gt;, &lt;np-array2&gt;)</code>	Dot Product																																																			
<code>np.matmul(&lt;np-array1&gt;, &lt;np-array2&gt;)</code>	Matrix Multiplication $\text{np-array1} @ \text{np-array2}$																																																			
<code>np.linalg.norm(&lt;np-array&gt;)</code>																																																				
<b>Search Methods</b>		<b>Sorting Methods</b>																																																		
<table border="1"><thead><tr><th>Method</th><th>Description</th></tr></thead><tbody><tr><td><code>np.argmax(&lt;np-array&gt;, &lt;axis&gt;)</code></td><td>Max Element Index</td></tr><tr><td><code>np.argmin(&lt;np-array&gt;, &lt;axis&gt;)</code></td><td>Min Element Index</td></tr><tr><td><code>np.where(&lt;condition&gt;, &lt;true-return-value&gt;, &lt;false-return-value&gt;)</code></td><td>Conditional Search and Replacement</td></tr><tr><td><code>np.nonzero(&lt;np-array&gt;)</code></td><td>Index of non-zero elements</td></tr></tbody></table>		Method	Description	<code>np.argmax(&lt;np-array&gt;, &lt;axis&gt;)</code>	Max Element Index	<code>np.argmin(&lt;np-array&gt;, &lt;axis&gt;)</code>	Min Element Index	<code>np.where(&lt;condition&gt;, &lt;true-return-value&gt;, &lt;false-return-value&gt;)</code>	Conditional Search and Replacement	<code>np.nonzero(&lt;np-array&gt;)</code>	Index of non-zero elements	<table border="1"><thead><tr><th>Method</th><th>Description</th></tr></thead><tbody><tr><td><code>np.sort(&lt;np-array&gt;, &lt;axis&gt;)</code></td><td>Sort Array</td></tr><tr><td><code>np.argsort(&lt;np-array&gt;, &lt;axis&gt;)</code></td><td>Return the order of indices that sort the array</td></tr></tbody></table>	Method	Description	<code>np.sort(&lt;np-array&gt;, &lt;axis&gt;)</code>	Sort Array	<code>np.argsort(&lt;np-array&gt;, &lt;axis&gt;)</code>	Return the order of indices that sort the array																																		
Method	Description																																																			
<code>np.argmax(&lt;np-array&gt;, &lt;axis&gt;)</code>	Max Element Index																																																			
<code>np.argmin(&lt;np-array&gt;, &lt;axis&gt;)</code>	Min Element Index																																																			
<code>np.where(&lt;condition&gt;, &lt;true-return-value&gt;, &lt;false-return-value&gt;)</code>	Conditional Search and Replacement																																																			
<code>np.nonzero(&lt;np-array&gt;)</code>	Index of non-zero elements																																																			
Method	Description																																																			
<code>np.sort(&lt;np-array&gt;, &lt;axis&gt;)</code>	Sort Array																																																			
<code>np.argsort(&lt;np-array&gt;, &lt;axis&gt;)</code>	Return the order of indices that sort the array																																																			

NumPy holds wide applicability in industry and academia due to its unparalleled potential.

Thus, being aware of its most common methods is necessary for Data Scientists.

Yet, it is important to understand that whenever you are learning a new library, mastering/practicing each and every method is not necessary.

What's more, this may be practically infeasible and time-consuming in many cases.

Instead, put Pareto's principle to work:



*20% of your inputs contribute towards generating 80% of your outputs.*

In other words, there are always some specific methods that are most widely used.

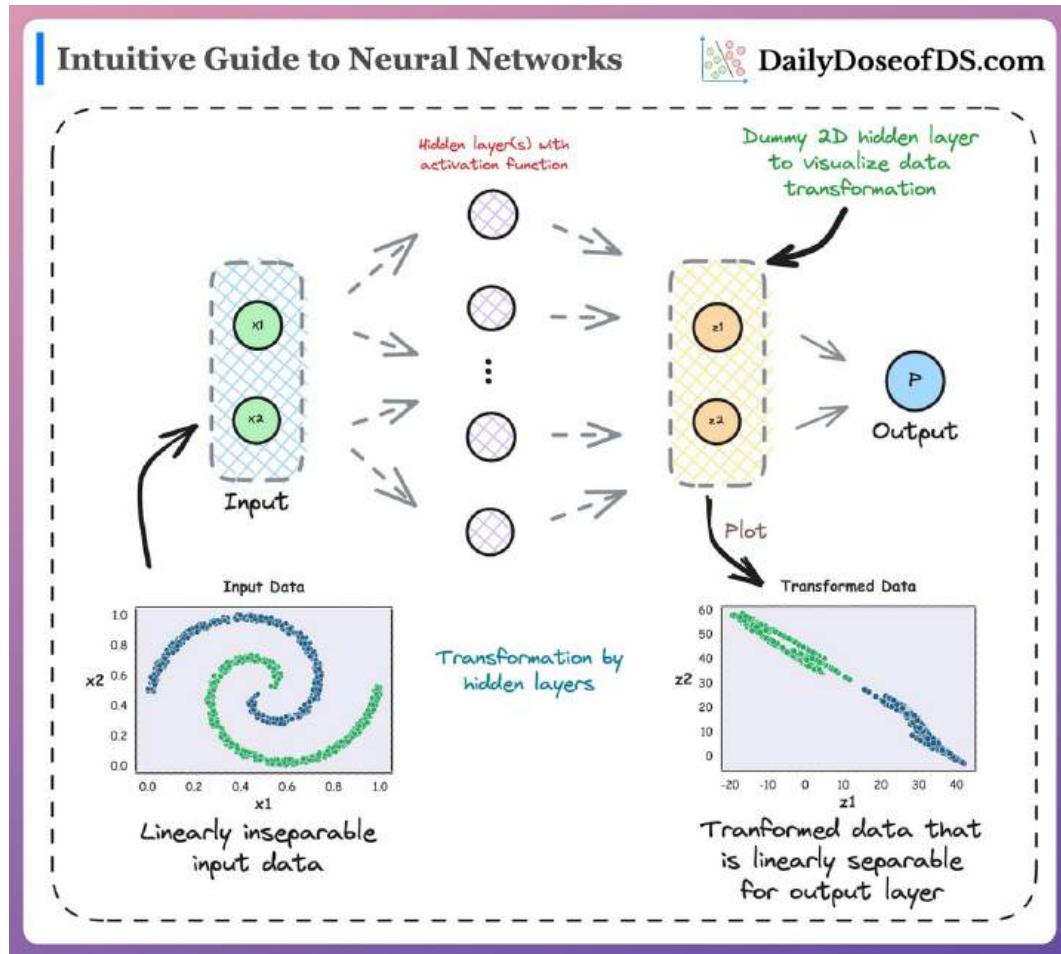
The above visual depicts the 40 most commonly used methods for NumPy.

Having used NumPy for over 4 years, I can confidently say that you will use these methods 95% of the time working with NumPy.

If you are looking for an in-depth guide, you can read my article on Medium here: [Medium NumPy article](#).



# An Overly Simplified Guide To Understanding How Neural Networks Handle Linearly Inseparable Data



Many folks struggle to truly comprehend how a neural network learns complex non-linear patterns.

Here's an intuitive explanation to understand the data transformations performed by a neural network when modeling linearly inseparable data.

---

We know that in a neural network, the data is passed through a series of transformations at every hidden layer.



This involves:

- Linear transformation of the data obtained from the previous layer
- ...followed by a non-linearity using an activation function — ReLU, Sigmoid, Tanh, etc.
- This is depicted below:

$$a_l = \sigma(W_l \cdot a_{l-1} + b_l)$$

where:

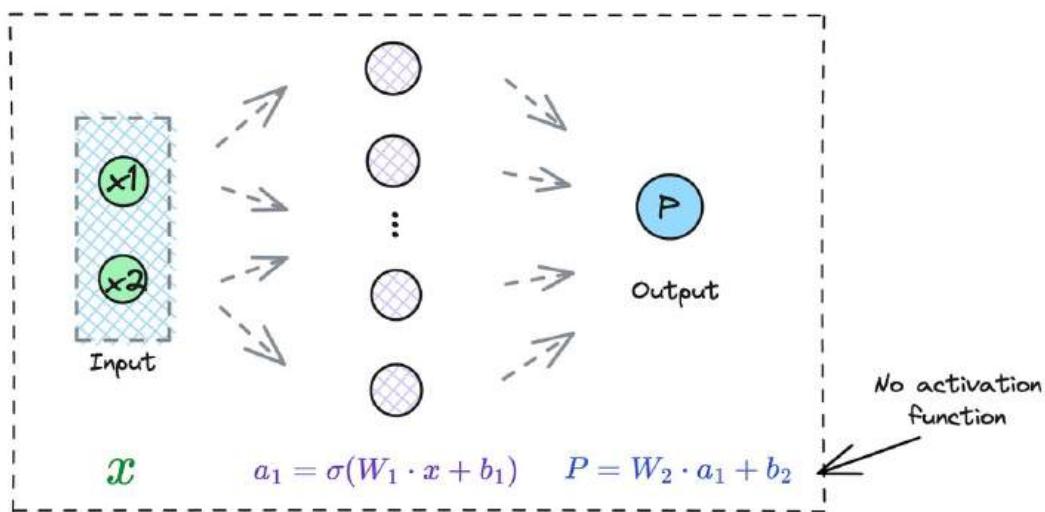
$a_l$  : output activation of current layer

$a_{l-1}$  : output activation of previous layer

$b_l$  : bias       $W_l$  : weights

$\sigma$  : activation function

For instance, consider a neural network with just one hidden layer:



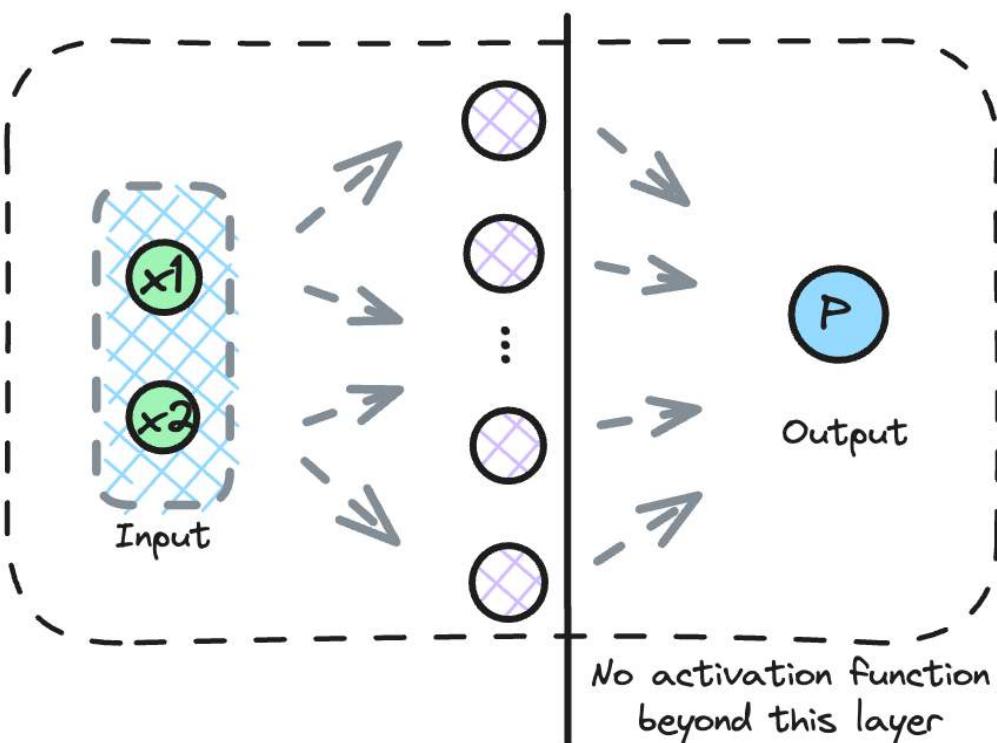


The data is transformed at the hidden layer along with an activation function.

Lastly, the output of the hidden layer is transformed to obtain the final output.

It's time to notice something here.

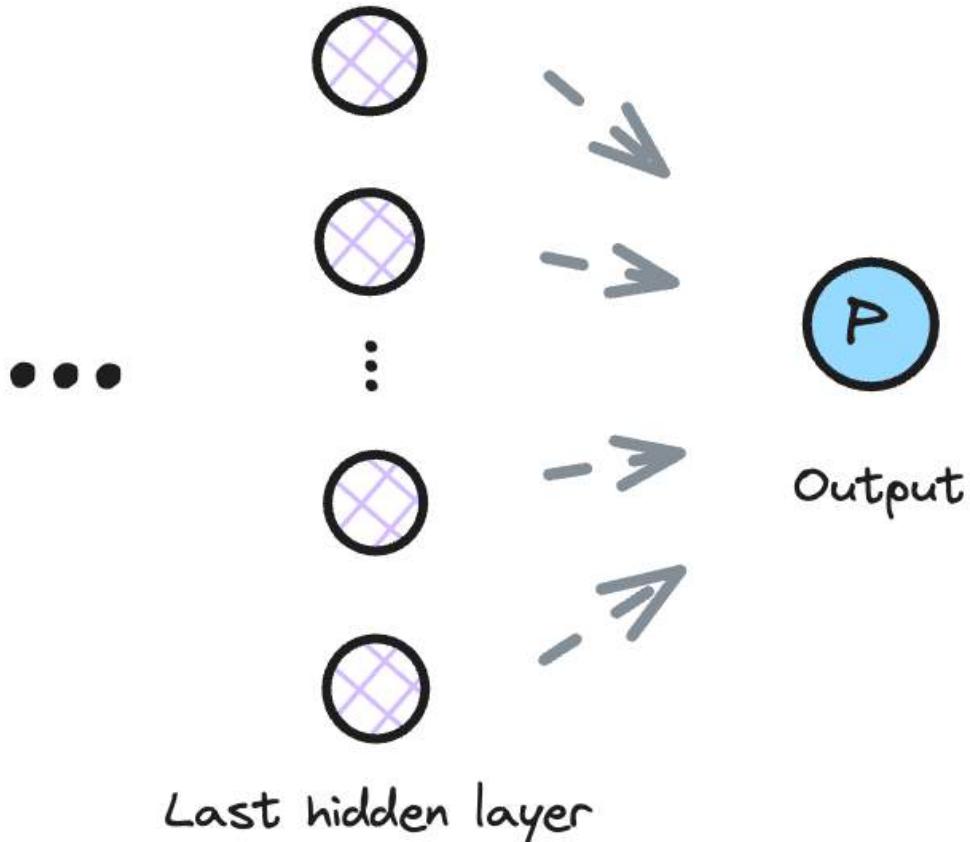
When the data comes out of the last hidden layer, and it is progressing towards the output layer for another transformation, **EVERY** activation function that ever existed in the network has already been utilized.



In other words, in any neural network, all sources of non-linearity — “activation functions”, exist on or before the last hidden layer.



And while progressing from the last hidden layer to the output layer, the data will pass through one final transformation before it spits some output.



But given that the transformation from the last hidden layer to the output layer is entirely linear (or without any activation function), there is no further scope for non-linearity in the network.



$$o = W_L \cdot a_L + b_L$$

where:

- $o$  : output of neural network
- $a_L$  : output activation of last hidden layer
- $b_L$  : bias       $W_L$  : weights

On a side note, the transformation from the last hidden layer to the output layer (assuming there is only one output neuron) can be thought of as a:

- linear regression model for regression tasks, or,
- logistic regression if you are modeling class probability with sigmoid function.

Thus, to make accurate predictions, the data received by the output layer from the last hidden layer MUST BE linearly separable.

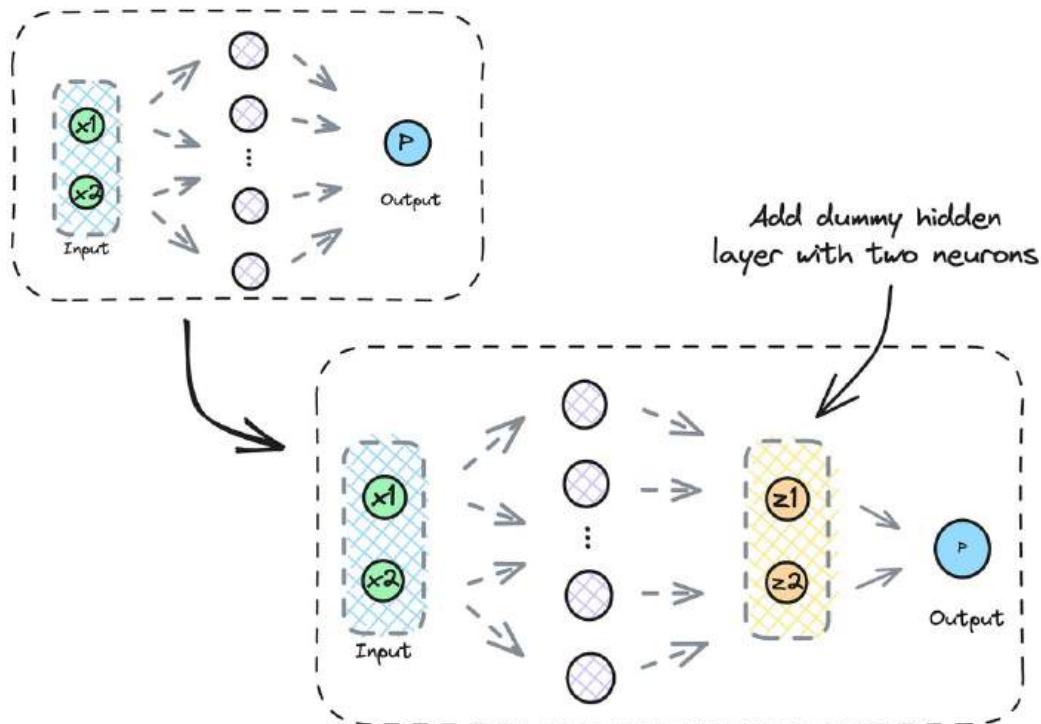
To summarize...

While transforming the data through all its hidden layers and just before reaching the output layer, a neural network is constantly hustling to project the data to a latent space where it becomes linearly separable.

Once it does, the output layer can easily handle the data.

We can also verify this experimentally.

To visualize the input transformation, add a dummy hidden layer with just two neurons **right before the output layer** and train the neural network again.



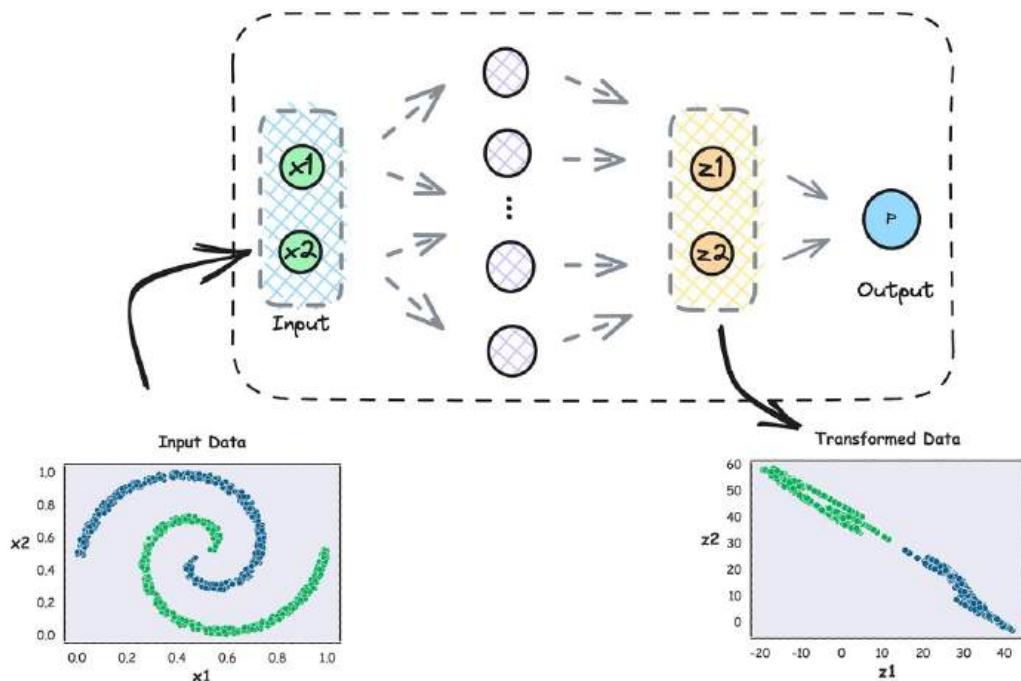
Why two neurons?

It's simple.

So that we can visualize it easily.

We expect that if we plot the activations of this 2D dummy hidden layer, they should be linearly separable.

The below visual precisely depicts this.



As we notice above, while the input data was linearly inseparable, the input received by the output layer is indeed linearly separable.

This transformed data can be easily handled by the output classification layer.

Hope that helped!

Feel free to respond with any queries that you may have.

👉 If you wish to experiment yourself, the code is available here: [Notebook](#).



## 2 Mathematical Proofs of Ordinary Least Squares

**2 Mathematical Proofs of Ordinary Least Squares**  DailyDoseofDS.com

**Objective:** Find  $\theta$ , such that :  $y = X\theta$

**Shapes:**  $y \rightarrow (n, 1)$   $X \rightarrow (n, m)$   $\theta \rightarrow (m, 1)$

Proof #1	Proof #2
1) Linear Regression Equation: $y = X\theta$	1) Minimize Squared Error: $L = \ y - X\theta\ ^2$
2) We cannot invert $X$ to get $\theta$ $\theta = X^{-1}y$ ...this is because $X$ may not be square. Hence, non-invertible.	2) Split the squared norm into 2 terms $L = (y - X\theta)^T(y - X\theta)$ $= y^T y - y^T X\theta - (X\theta)^T y + (X\theta)^T X\theta$
3) Multiple both sides of (1) by $X^T$ $X^T y = X^T X\theta$ $X^T X$ is square. Hence, invertible*.	3) Minimize by differentiating w.r.t $\theta$ and then solve for $\theta$ $\frac{dL}{d\theta} = -2X^T y + 2X^T X\theta$
4) Invert $X^T X$ $\theta = (X^T X)^{-1} X^T y$	4) Set the derivative to zero $-2X^T y + 2X^T X\theta = 0$ rearrange... $X^T X\theta = X^T y$
*In case of perfect multicollinearity, this won't be invertible.	5) Invert $X^T X$ $\theta = (X^T X)^{-1} X^T y$

Most machine learning algorithms use gradient descent to learn the optimal parameters.

However, in addition to gradient descent, linear regression can model data using another technique called ordinary least squares (OLS).

### Ordinary Least Square (OLS):

1. It is a deterministic algorithm. If run multiple times, it will always converge to the same weights.
2. It always finds the optimal solution.

The above image shows two ways to find the OLS solution of OLS.

Full issue here: <https://www.blog.dailydoseofds.com/p/2-mathematical-proofs-of-ordinary>



# A Common Misconception About Log Transformation



Log transform is commonly used to eliminate skewness in data.

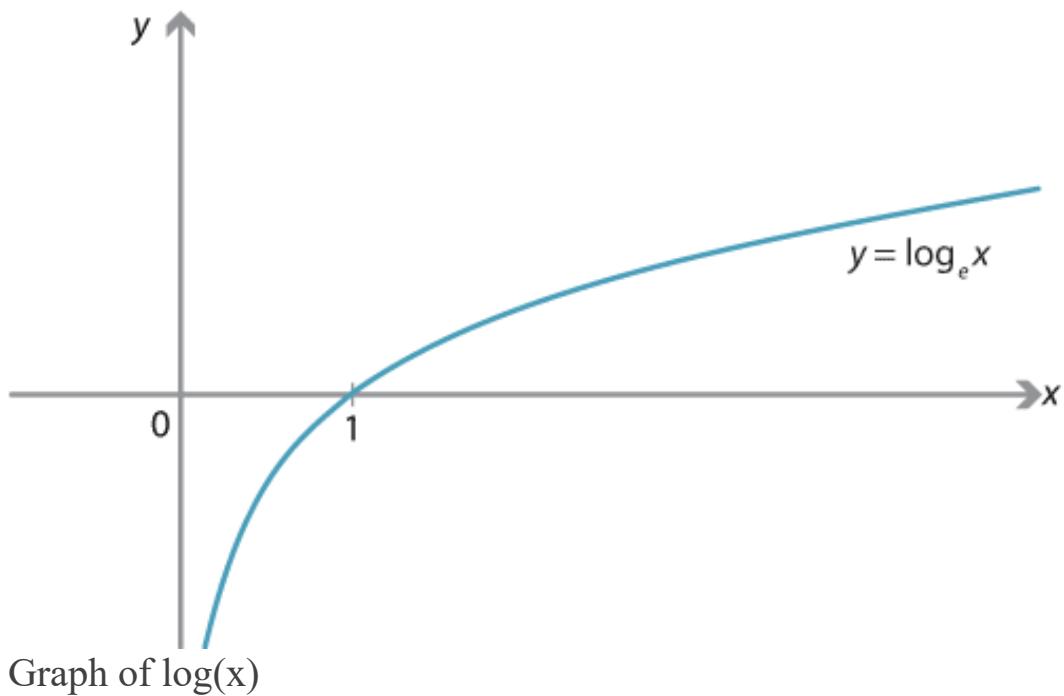
Yet, it is not always the ideal solution for eliminating skewness.

It is important to note that log transform:

- Does not eliminate left-skewness.
- Only works for right-skewness, that too when the values are small and positive.

This is also evident from the image above.

It is because the log function grows faster for lower values. Thus, it stretches out the lower values more than the higher values.



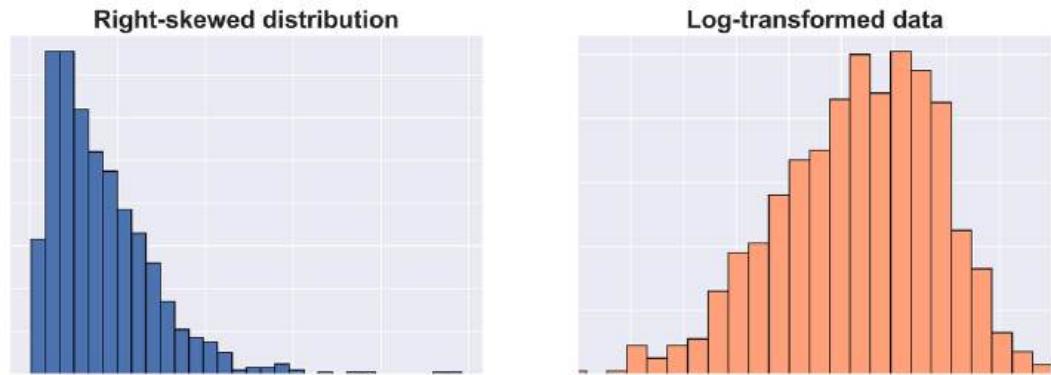
Thus,

- In case of left-skewness:



Left-skewness with log transform

- The tail exists to the left, which gets stretched out more than those to the right
  - Thus, skewness isn't affected much.
- In case of right-skewness:



### Right-skewness with log transform

- Majority of values and peak exists to the left, which get stretched out more.
- However, the log function grows slowly when the values are large. Thus, the impact of stretch is low.

There are a few things you can do:

- See if transformation can be avoided as it inhibits interpretability.
- If not, try box-cox transform. It is often quite effective, both for left-skewed and right-skewed data. You can use it using Scipy's implementation: [Scipy docs](#).

👉 Over to you: What are some other ways to eliminate skewness?



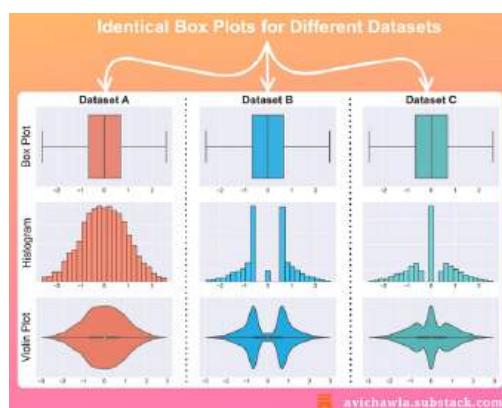
# Raincloud Plots: The Hidden Gem of Data Visualisation



Visualizing data distributions using box plots and histograms can be misleading at times.

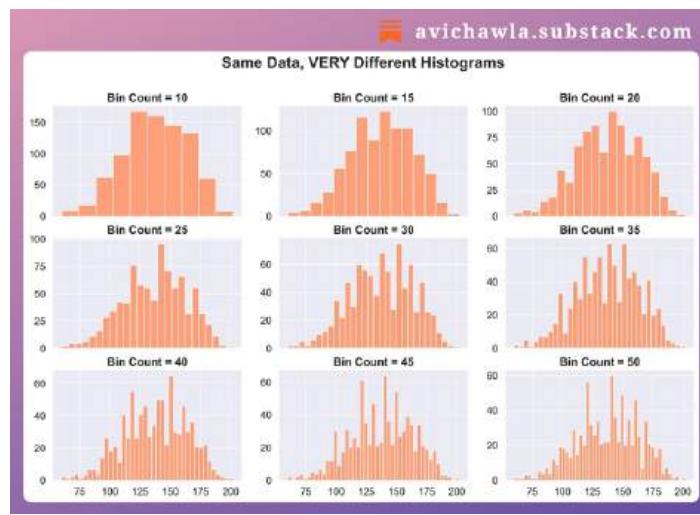
This is because:

- It is possible to get the same box plot with entirely different data.
  - For instance, consider the illustration below from one of my previous posts: [Use Box Plots With Caution! They May Be Misleading.](#)





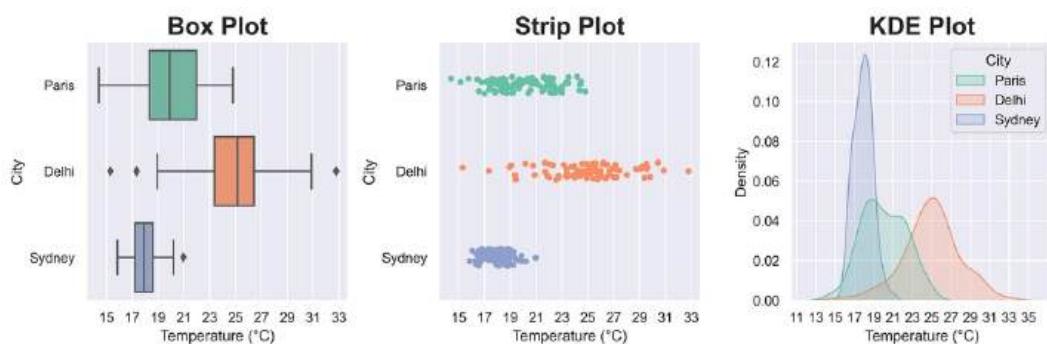
- We get the same box plot with three different datasets.
- Altering the number of bins changes the shape of a histogram.
  - [Read this post here.](#)



Thus, to avoid misleading conclusions, it is recommended to plot the data distribution.

Here, jitter (strip) plots and KDE plots are immensely helpful.

One way is to draw them separately and analyze them together, as shown below. But this is quite tedious.



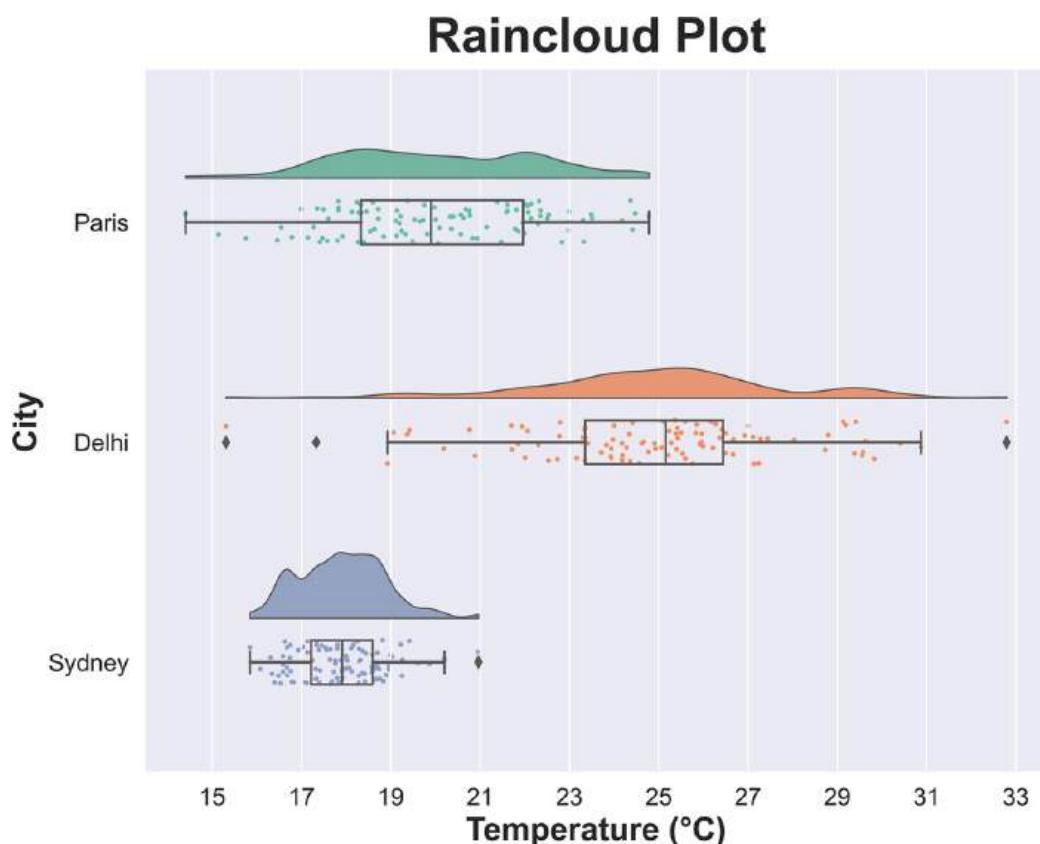


Instead, try Raincloud plots.

They provide a concise way to combine and visualize three different types of plots together.

These include:

- Box plots for data statistics.
- Strip plots for data overview.
- KDE plots for the probability distribution of data.



Raincloud plot with Box, strip and KDE plot at once

Overall, Raincloud plots are an excellent choice for data visualization.

With Raincloud plots, you can:

- Combine multiple plots to prevent incorrect/misleading conclusions

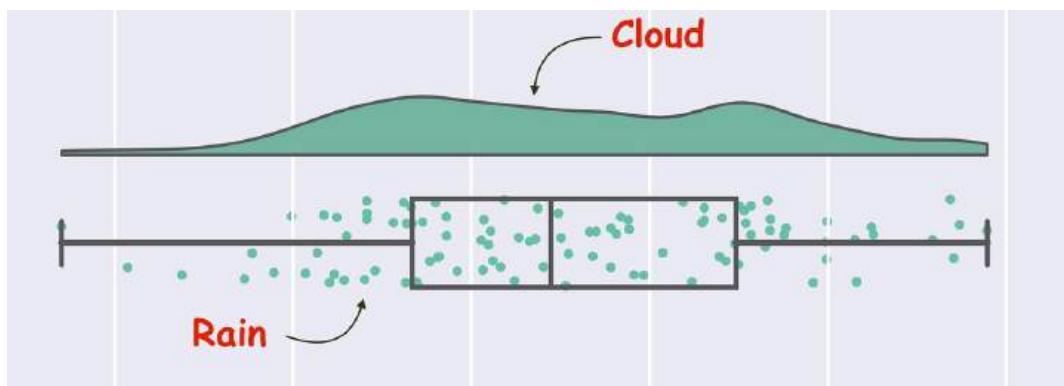


- Reduce clutter and enhance clarity
- Improve comparisons between groups
- Capture different aspects of the data through a single plot

You can use the PtitPrince library to create Raincloud plots in Python: [GitHub](#).

R users can use Raincloud Plots library: [GitHub](#).

P.S. If the name “Raincloud plot” isn’t obvious yet, it comes from the visual appearance of the plot:

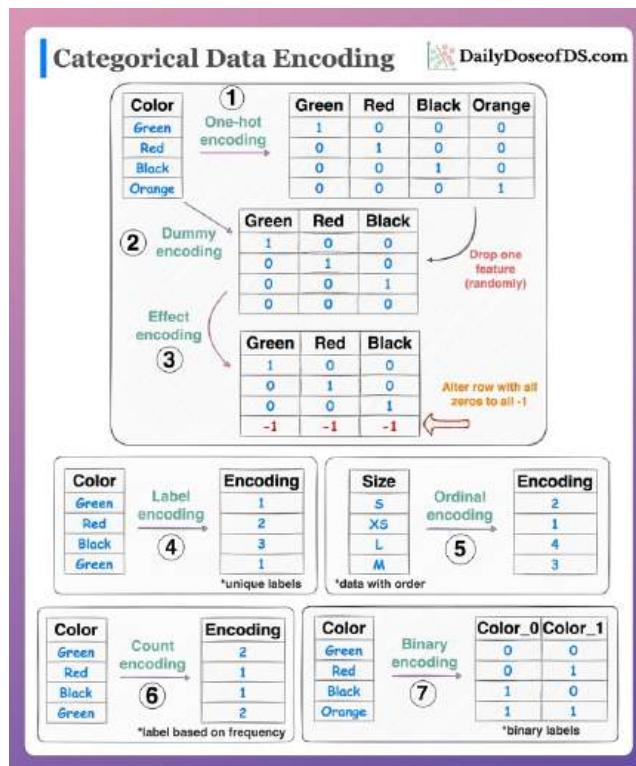


The origin of the name “Raincloud plot”

👉 Over to you: What are some other hidden gems of data visualization?



# 7 Must-know Techniques For Encoding Categorical Feature



Almost all real-world datasets come with multiple types of features.

These primarily include:

- Categorical
- Numerical

While numerical features can be directly used in most ML models without any additional preprocessing, categorical features require encoding to be represented as numerical values.

On a side note, do you know that not all ML models need categorical feature encoding? Read one of my previous guides on this here: **Is Categorical Feature Encoding Always Necessary Before Training ML Models?**

---

If categorical features do need some additional processing, being aware of the common techniques to encode them is crucial.

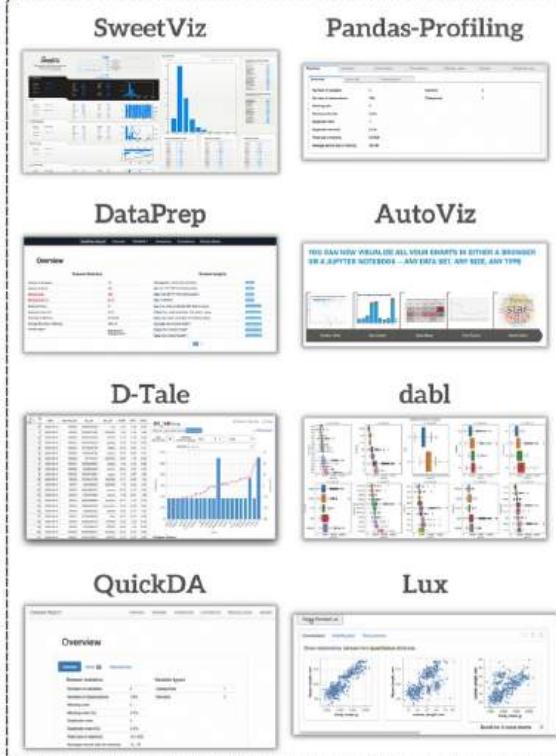
The above visual summarizes 7 most common methods for encoding categorical features.

Read the full issue here: <https://www.blog.dailydoseofds.com/p/7-must-know-techniques-for-encoding>



# Automated EDA Tools That Let You Avoid Manual EDA Tasks

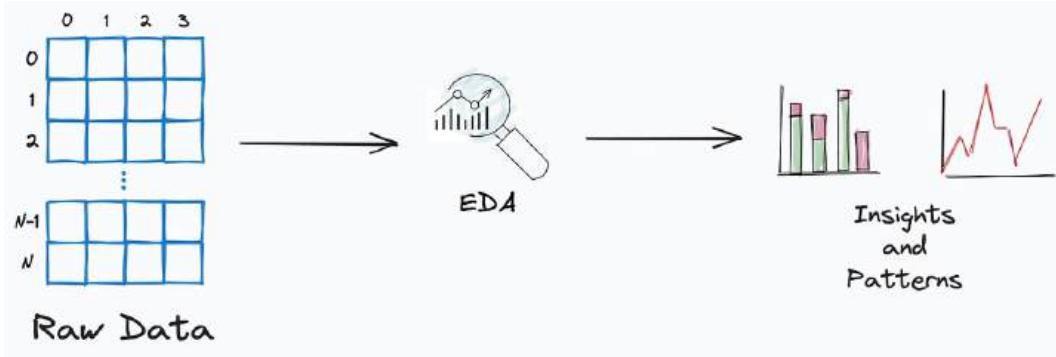
8 Automated EDA Tools  DailyDoseofDS.com



The collage displays the user interfaces of eight different automated EDA tools. SweetViz shows a dashboard with various charts and metrics. Pandas-Profiling provides a detailed report on data types and distributions. DataPrep offers a comprehensive overview of data quality and schema. AutoViz allows users to visualize data in Jupyter notebooks. D-Tale and dabl both feature extensive data exploration dashboards with many charts. QuickDA and Lux provide similar functionalities for data analysis and visualization.

EDA is a vital step in all data science projects.

It is important because examining and understanding the data directly aids the modeling stage.





By uncovering hidden insights and patterns, one can make informed decisions about subsequent steps in the project.

Despite its importance, it is often a time-consuming and tedious task.

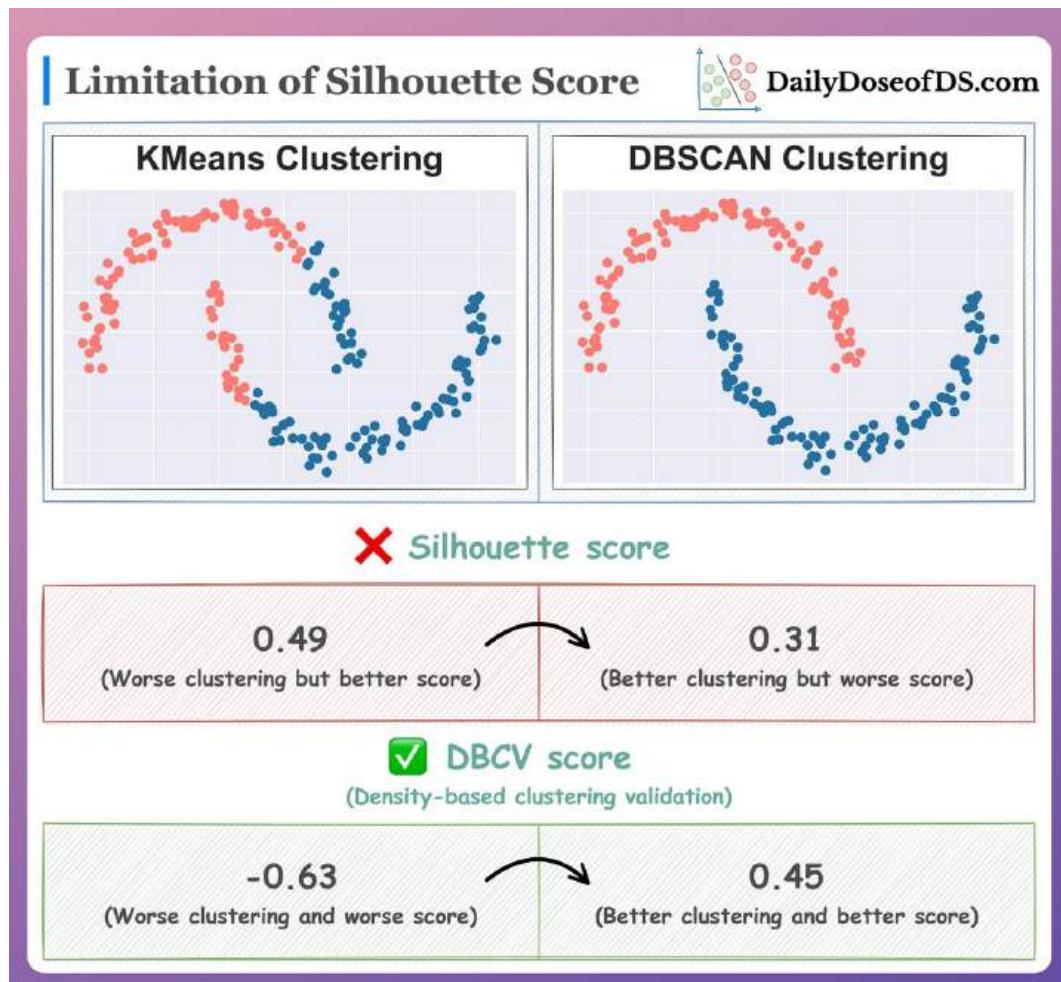
The above visual summarizes 8 powerful EDA tools, that automate many redundant steps of EDA and help you profile your data in quick time.

**Read the full issue here to learn more about each of these tools:**

<https://www.blog.dailydoseofds.com/p/automated-eda-tools-that-let-you>



# The Limitation Of Silhouette Score Which Is Often Ignored By Many



Silhouette score is commonly used for evaluating clustering results.

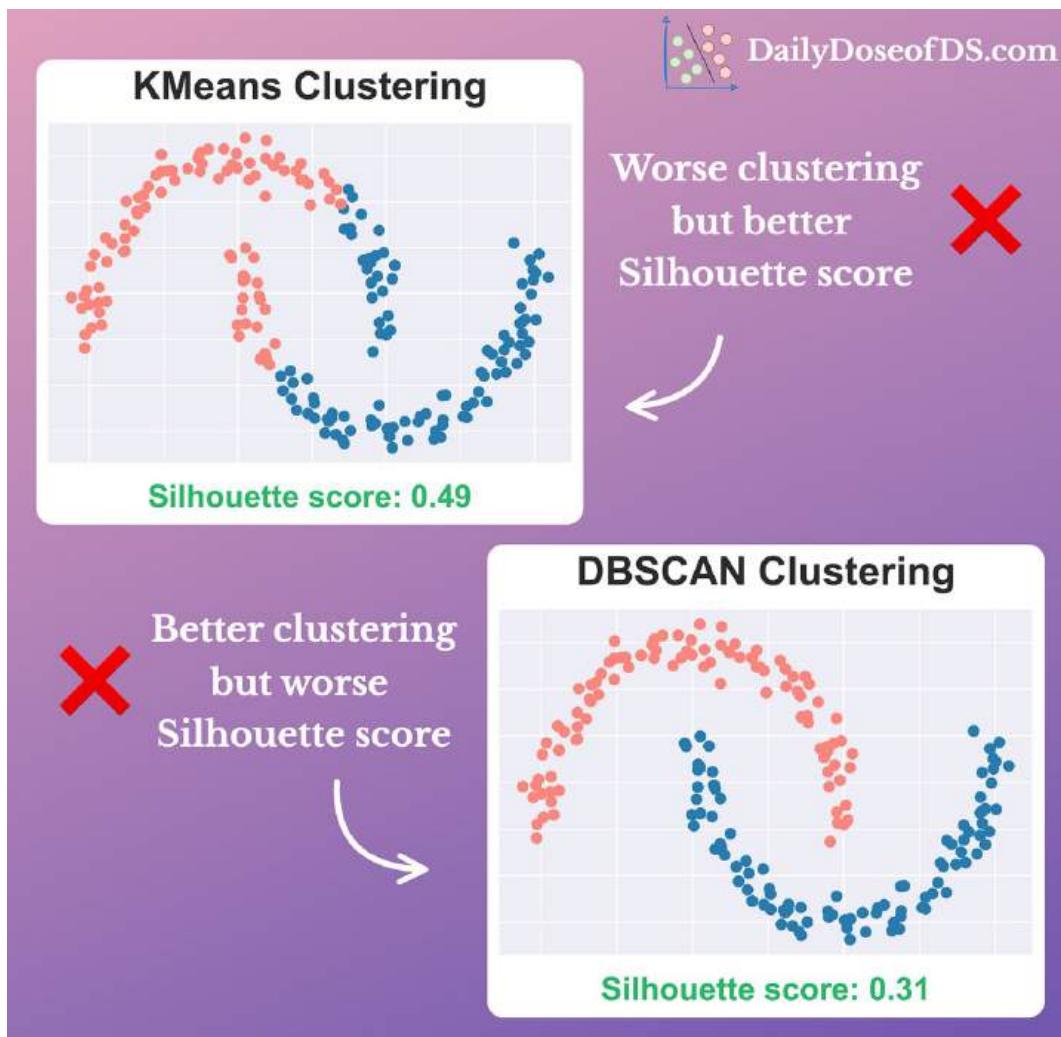
At times, it is also preferred in place of the elbow curve to determine the optimal number of clusters. ([I have covered this before if you wish to recap or learn more](#)).

However, while using the Silhouette score, it is also important to be aware of one of its major shortcomings.

The Silhouette score is typically higher for convex (or somewhat spherical) clusters.

However, using it to evaluate arbitrary-shaped clustering can produce misleading results.

This is also evident from the following image:



While the clustering output of KMeans is worse, the Silhouette score is still higher than Density-based clustering.

DBCV — density-based clustering validation is a better metric in such cases.

As the name suggests, it is specifically meant to evaluate density-based clustering.

Simply put, DBCV computes two values:

- The density **within** a cluster
- The density **between** clusters

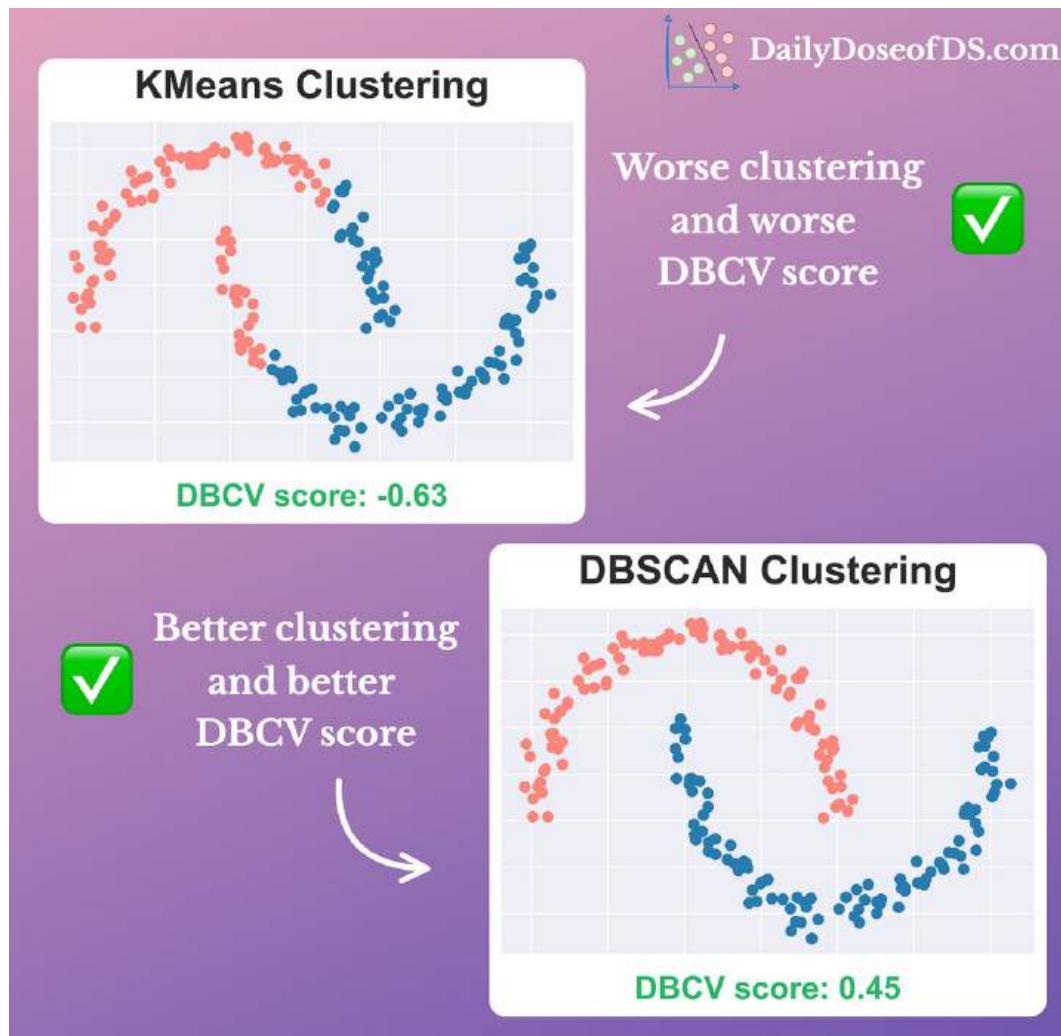
A high density within a cluster and a low density between clusters indicates good clustering results.

DBCV can also be used when you don't have ground truth labels.



This adds another metric to my recently proposed methods: [Evaluate Clustering Performance Without Ground Truth Labels](#).

The effectiveness of DBCV is also evident from the image below:



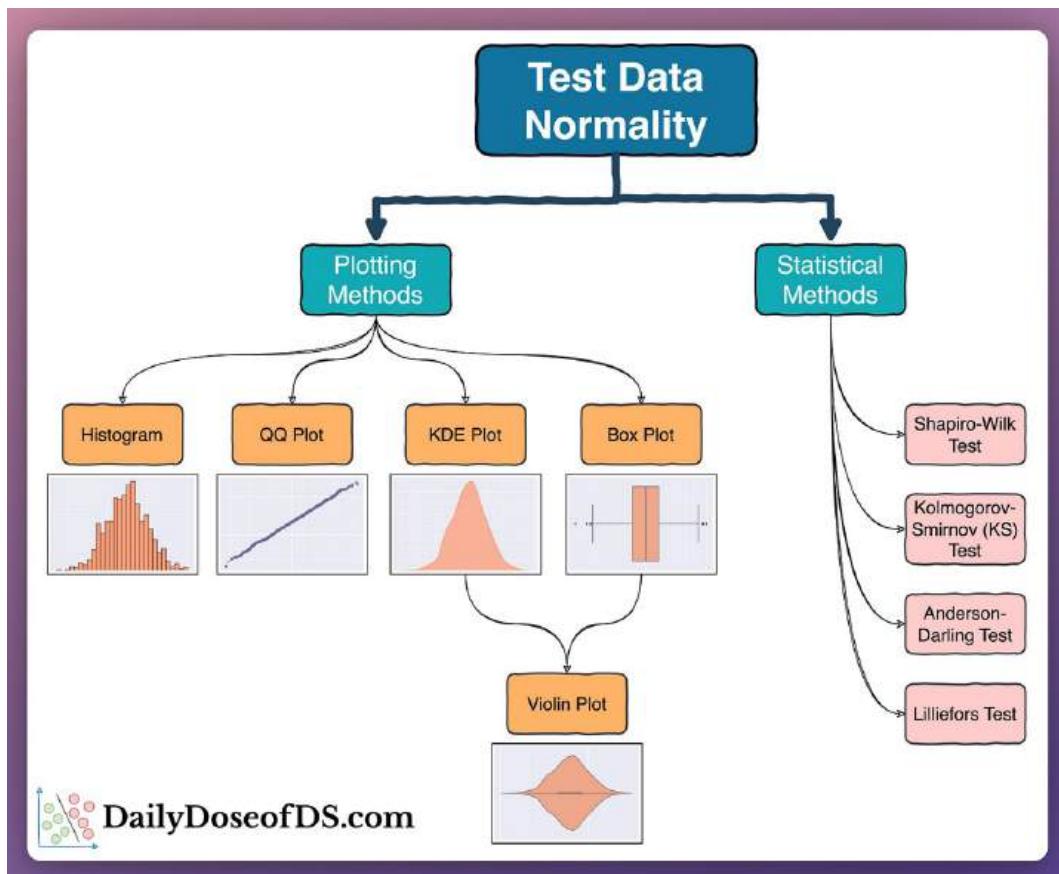
This time, the score for the clustering output of KMeans is worse, and that of density-based clustering is higher.

Get started with DBCV here: [GitHub](#).

👉 Over to you: What are some other ways to evaluate clustering where traditional metrics may not work?



# 9 Must-Know Methods To Test Data Normality



The normal distribution is the most popular distribution in data science.

Many ML models assume (or work better) under the presence of normal distribution.

For instance:

1. linear regression assumes residuals are normally distributed
2. at times, transforming the data to normal distribution can be beneficial (Read one of my previous posts on this [here](#))
3. linear discriminant analysis (LDA) is derived under the assumption of normal distribution
4. and many more.

Thus, being aware of the ways to test normality is extremely crucial for data scientists.

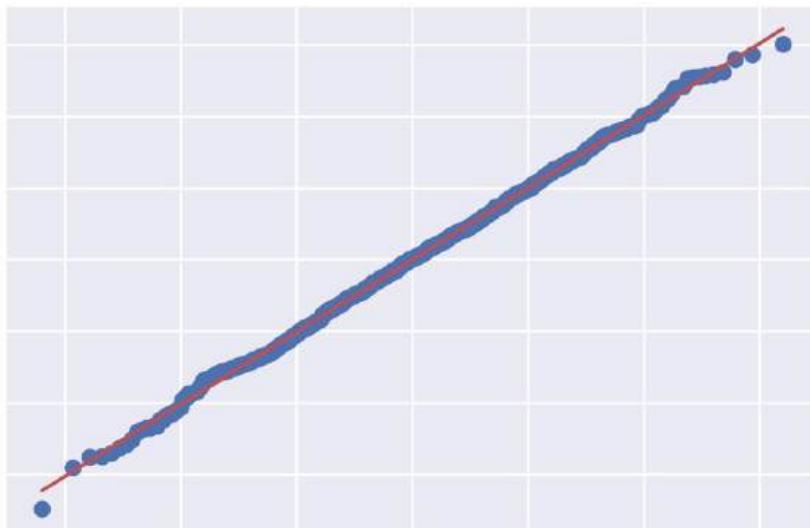
The visual above depicts the 9 most common methods to test normality.



## #1) Plotting methods:

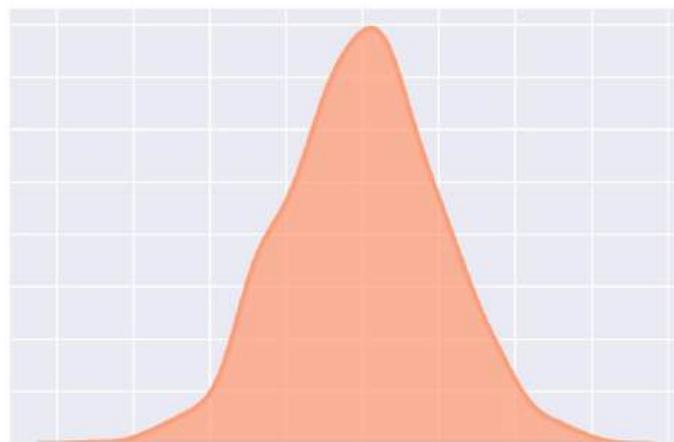
**Histogram**

**QQ Plot:**



1. It depicts the quantiles of the observed distribution (the given data in this case) against the quantiles of a reference distribution (the normal distribution in this case).
2. A good QQ plot will show minimal deviations from the reference line, indicating that the data is approximately normally distributed.
3. A bad QQ plot will exhibit significant deviations, indicating a departure from normality.

**KDE (Kernel Density Estimation) Plot:**

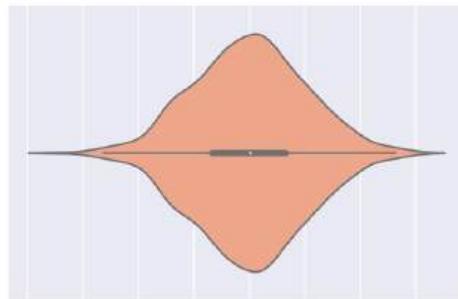




1. It provides a smoothed, continuous representation of the underlying distribution of a dataset.
2. It represents the data using a continuous probability density function.

### Box plot

### Violin plot:



1. A combination of a box plot and a KDE plot.

---

## #2) Statistical methods:

While the plotting methods discussed above are often reliable, they offer a subjective method to test normality.

In other words, the approach of visual interpretation is prone to human errors.

Thus, it is important to be aware of quantitative measures as well.

### Shapiro-Wilk test:

- a. The most common method for testing normality.
- b. It calculates a statistic based on the correlation between the data and the expected values under a normal distribution.
- c. This results in a p-value that indicates the likelihood of observing such a correlation if the data were normally distributed.
- d. A high p-value indicates the presence of samples drawn from a normal distribution.
- e. Get started: [Scipy Docs](#).



### Kolmogorov-Smirnov (KS) test:

- a. The Kolmogorov-Smirnov test is typically used to determine if a dataset follows a specific distribution—normal distribution in normality testing.
- b. The KS test compares the cumulative distribution function (CDF) of the data to the cumulative distribution function (CDF) of a normal distribution.
- c. The output statistic is based on the maximum difference between the two distributions.
- d. A high p-value indicates the presence of samples drawn from a normal distribution.
- e. Get started: [Scipy Docs](#).

### Anderson-Darling test

- a. Another method to determine if a dataset follows a specific distribution—normal distribution in normality testing.
- b. It provides critical values at different significance levels.
- c. Comparing the obtained statistic to these critical values determines whether we will reject or fail to reject the null hypothesis of normality.
- d. Get started: [Scipy Docs](#).

### Lilliefors test

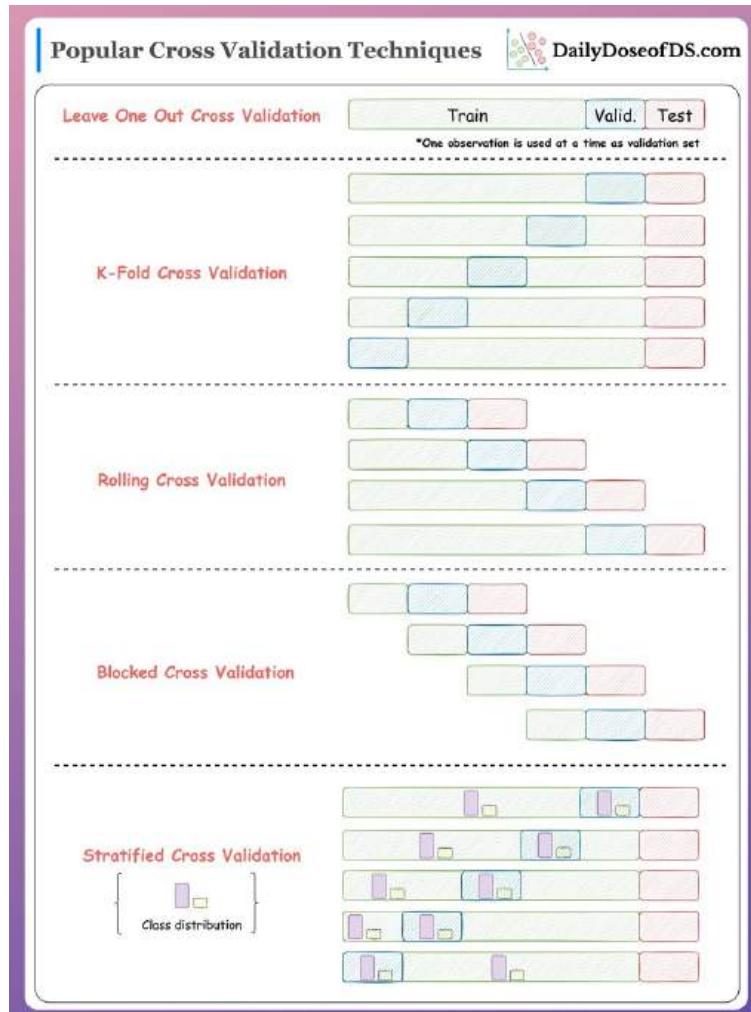
- a. It is a modification of the Kolmogorov-Smirnov test.
- b. The KS test is appropriate in situations where the parameters of the reference distribution are known.
- c. However, if the parameters are unknown, Lilliefors is recommended.
- d. Get started: [Statsmodel Docs](#).

If you are looking for an in-depth review and comparison of these tests, I highly recommend reading this research paper: [Power comparisons of Shapiro-Wilk, Kolmogorov-Smirnov, Lilliefors and Anderson-Darling tests](#).

👉 Over to you: What other common methods have I missed?



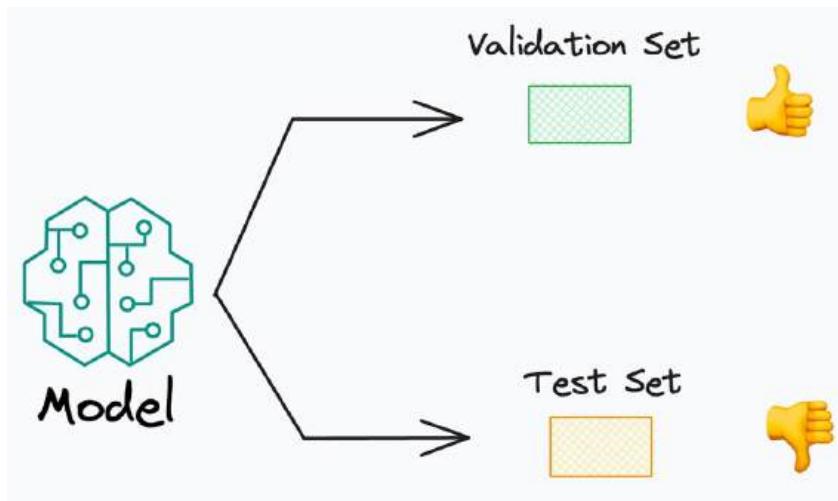
# A Visual Guide to Popular Cross Validation Techniques



Tuning and validating machine learning models on a single validation set can be misleading at times.

While traditional validation methods, such as a single train-test split, are easy to implement, they, at times, can yield overly optimistic results.

This can occur due to a lucky random split of data which results in a model that performs exceptionally well on the validation set but poorly on new, unseen data.



That is why we often use cross-validation instead of simple single-set validation.

Cross-validation involves repeatedly partitioning the available data into subsets, training the model on a few subsets, and validating on the remaining subsets.

The main advantage of cross-validation is that it provides a more robust and unbiased estimate of model performance compared to the traditional validation method.

The image above presents a visual summary of five of the most commonly used cross-validation techniques.

## Leave-One-Out Cross-Validation



1. Leave one data point for validation.
2. Train the model on the remaining data points.
3. Repeat for all points.
4. This is practically infeasible when you have tons of data points. This is because number of models is equal to number of data points.
5. We can extend this to Leave-p-Out Cross-Validation, where, in each iteration,  $p$  observations are reserved for validation and the rest are used for training.

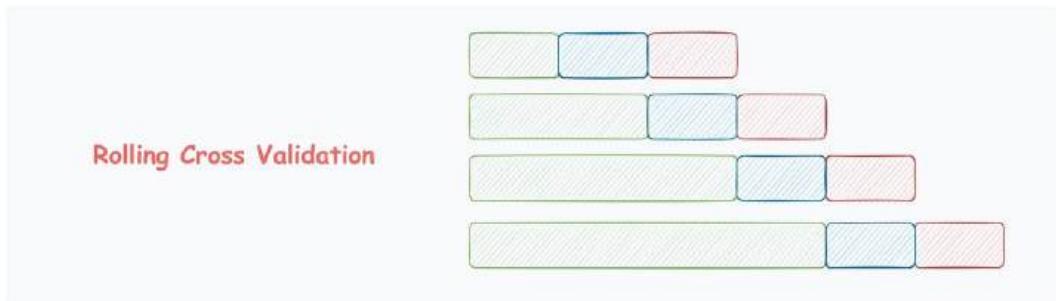


## K-Fold Cross-Validation



1. Split data into  $k$  equally-sized subsets.
2. Select one subset for validation.
3. Train the model on the remaining subsets.
4. Repeat for all subsets.

## Rolling Cross-Validation



1. Mostly used for data with temporal structure.
2. Data splitting respects the temporal order, using a fixed-size training window.
3. The model is evaluated on the subsequent window.

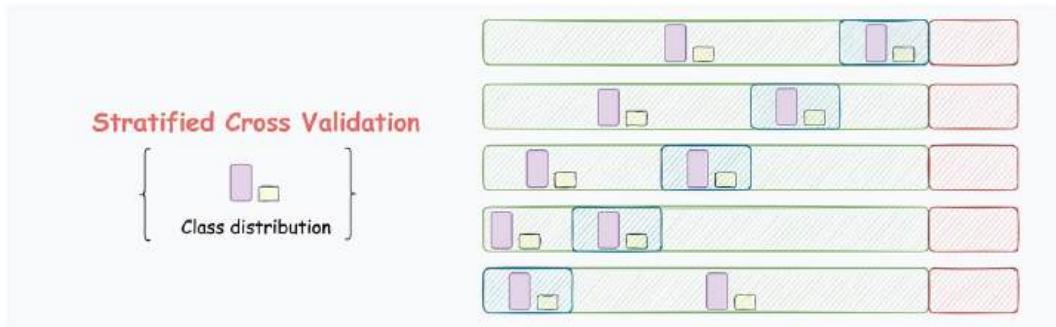


## Blocked Cross-Validation



1. Another common technique for time-series data.
2. In contrast to rolling cross-validation, the slice of data is intentionally kept short if the variance does not change appreciably from one window to the next.
3. This also saves computation over rolling cross-validation.

## Stratified Cross-Validation



1. The above techniques may not work for imbalanced datasets. Thus, this technique is mostly used for preserving the class distribution.
  2. The partitioning ensures that the class distribution is preserved.
- 👉 Over to you: What other cross-validation techniques have I missed?



# Decision Trees **ALWAYS** Overfit. Here's A Lesser-Known Technique To Prevent It.

By default, a decision tree (in sklearn's implementation, for instance), is allowed to grow until all leaves are pure.

As the model correctly classifies ALL training instances, this leads to:

1. 100% overfitting, and
2. poor generalization



Cost-complexity-pruning (CCP) is an effective technique to prevent this.

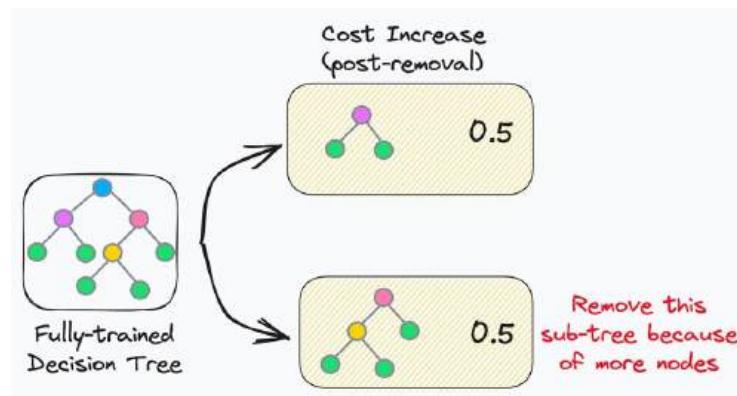
CCP considers a combination of two factors for pruning a decision tree:

1. Cost (C): Number of misclassifications
2. Complexity (C): Number of nodes

The core idea is to iteratively drop sub-trees, which, after removal, lead to:

1. a minimal increase in classification cost
2. a maximum reduction of complexity (or nodes)

In other words, if two sub-trees lead to a similar increase in classification cost, then it is wise to remove the sub-tree with more nodes.



Cost-complexity pruning at the same increase in misclassification cost.

In sklearn, you can control cost-complexity-pruning using the `ccp_alpha` parameter:

1. large value of `ccp_alpha` → results in underfitting
2. small value of `ccp_alpha` → results in overfitting

The objective is to determine the optimal value of `ccp_alpha`, which gives a better model.

The effectiveness of cost-complexity-pruning is evident from the image below:

The image compares two decision trees. The left panel shows a tree with `ccp_alpha = 0`, labeled 'Entirely Overfitted Decision Tree' with a red X. Its decision region plot is complex and jagged. The right panel shows a tree with `ccp_alpha = 0.007`, labeled 'Balanced Decision Tree' with a green checkmark. Its decision region plot is much simpler and balanced.

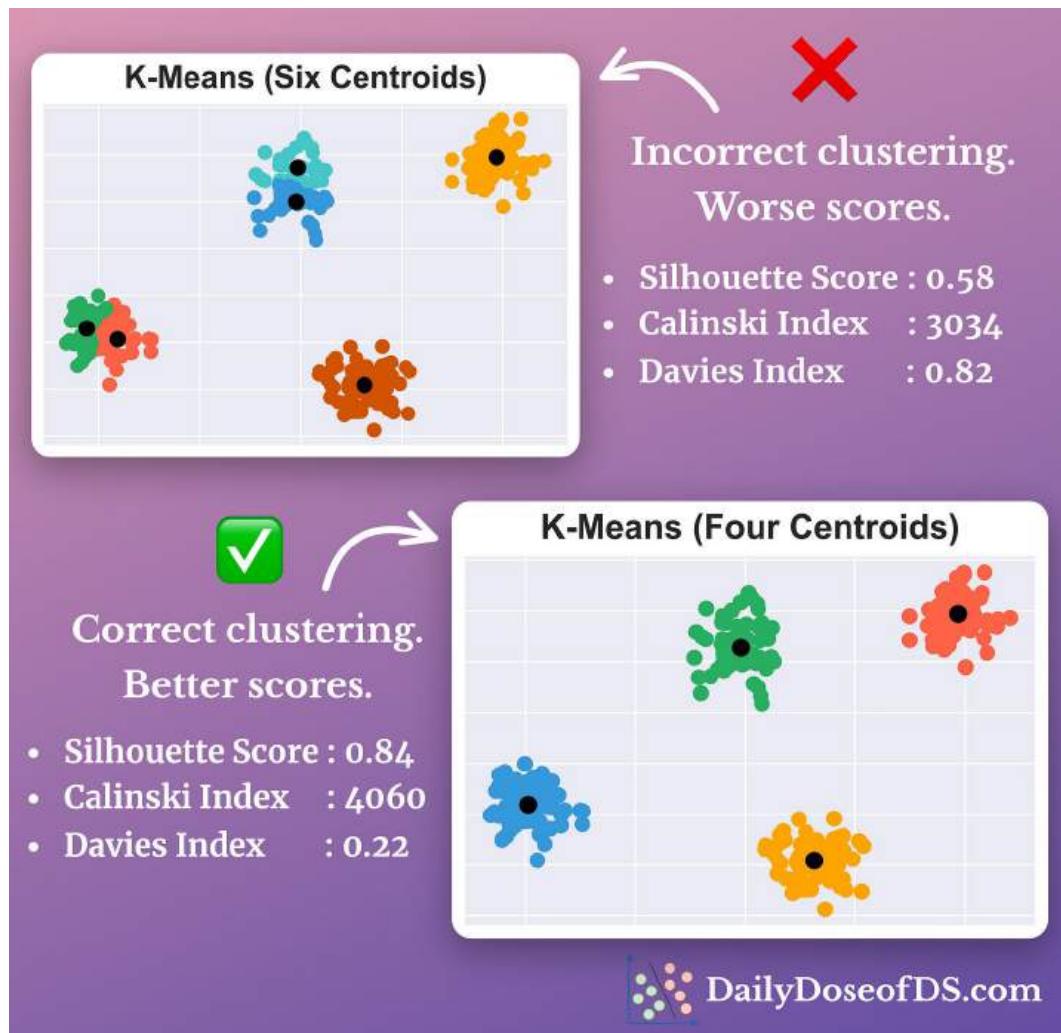
```
tree = DecisionTreeClassifier(ccp_alpha = 0).fit(X, y)
Training score: 1.0
Test score: 0.86
```

```
tree = DecisionTreeClassifier(ccp_alpha = 0.007).fit(X, y)
Training score: 0.93
Test score: 0.90
```

👉 Over to you: What are some other ways you use to prevent decision trees from overfitting?



# Evaluate Clustering Performance Without Ground Truth Labels



In the absence of ground truth labels, evaluating clustering performance is difficult.

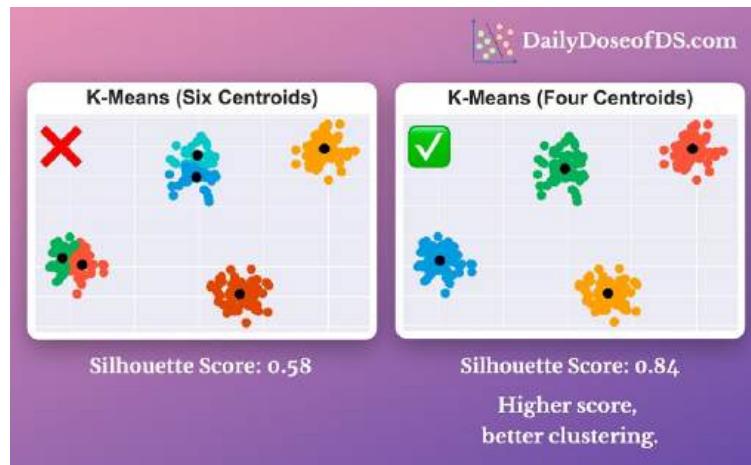
Yet, there are a few performance metrics that can help.

Using them, you can compare multiple clustering results, say, those obtained with a different number of centroids.

This is especially useful for high-dimensional datasets, as visual evaluation is difficult.



### Silhouette Coefficient:



1. for every point, find average distance to all other points within its cluster (A)
2. for every point, find average distance to all points in the nearest cluster (B)
3. score for a point is  $(B-A)/\max(B, A)$
4. compute the average of all individual scores to get the overall clustering score
5. computed on all samples, thus, it's computationally expensive
6. a higher score indicates better and well-separated clusters.

I covered this here if you wish to understand Silhouette Coefficient with diagrams: [The Limitations Of Elbow Curve And What You Should Replace It With.](#)

### Calinski-Harabasz Index:

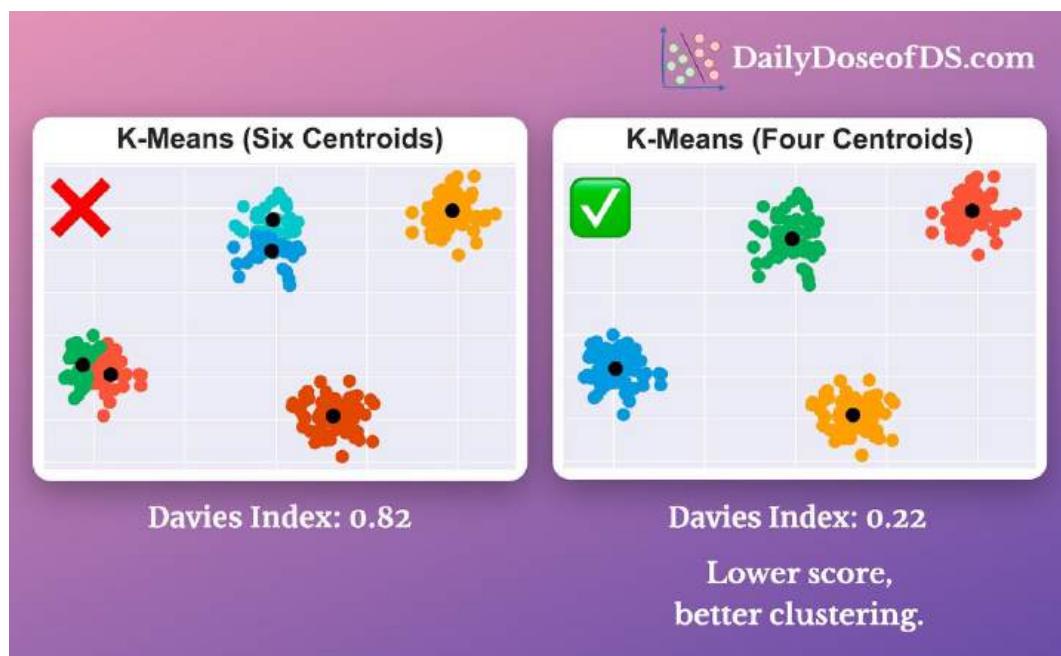


1. A: sum of squared distance between all centroids and overall dataset center



2. B: sum of squared distance between all points and their specific centroid
3. metric is computed as A/B (with an additional scaling factor)
4. relatively faster to compute
5. it is sensitive to scale
6. a higher score indicates well-separated clusters

Davies-Bouldin Index:



measures the similarity between clusters

thus, a lower score indicates dissimilarity and better clustering

Luckily, they are neatly integrated with sklearn too.

### Silhouette Coefficient

### Calinski-Harabasz Index

### Davies-Bouldin Index

👉 Over to you: What are some other ways to evaluate clustering performance in such situations?



# One-Minute Guide To Becoming a Polars-savvy Data Scientist

Pandas to Polars Guide		DailyDoseofDS.com	
Operation	Pandas	Polars	Syntax Comparison
Import	<code>import pandas as pd</code>	<code>import polars as pl</code>	-
Read CSV	<code>df = pd.read_csv(file)</code>	<code>df = pl.read_csv(file)</code>	Same
Save to CSV	<code>df.to_csv(file)</code>	<code>df.to_csv(file)</code>	Same
Print first 10 (or k) rows	<code>df.head(10)</code>	<code>df.head(10)</code>	Same
Dimensions	<code>df.shape</code>	<code>df.shape</code>	Same
Datatype	<code>df.dtypes</code>	<code>df.dtypes</code>	Same
Memory Usage	<code>df.memory_usage()</code>	<code>df.estimated_size()</code>	Different Method Name
Select column(s)	<code>df[["col1", "col2"]]</code>	<code>df[["col1", "col2"]]</code>	Same
Filter Data	<code>df[df.column &gt; 10]</code>	<code>df[df.column &gt; 10]</code>	Same
		<code>df.filter(pl.col("column") &gt; 10)</code>	Different
Sort	<code>df.sort_values("column")</code>	<code>df.sort("column")</code>	Similar
Fill NaN	<code>df.column.fillna(0)</code>	<code>df.column.fill_nan(0)</code>	Similar
Join	<code>pd.merge(df1, df2, on = "col", how = "inner")</code>	<code>df1.join(df2, on = "col", how = "inner")</code>	Similar
Concatenate	<code>pd.concat([df1, df2])</code>	<code>pl.concat([df1, df2])</code>	Same
Group	<code>df.groupby("column").agg_col.mean()</code>	<code>df.groupby("column").agg(pl.mean("agg_col"))</code>	Similar
Unique values	<code>df.column.unique()</code>	<code>df.column.unique()</code>	Same
Rename column	<code>df.rename(columns = {"old_name": "new_name"})</code>	<code>df.rename(mapping = {"old_name": "new_name"})</code>	Similar
Delete column	<code>df.drop(columns = ["column"])</code>	<code>df.drop(name = ["column"])</code>	Similar
Lazy Execution	<code>Not Supported</code>	<code>df.lazy()</code>	-

Pandas is an essential library in almost all Data Science projects.

But it has many limitations.

For instance, Pandas:

always adheres to single-core computation

offers no lazy execution

creates bulky DataFrames

is slow on large datasets, and many more

Polars is a lightning-fast DataFrame library that addresses these limitations.

It provides two APIs:

Eager: Executed instantly, like Pandas.

Lazy: Executed only when one needs the results.



The visual presents the syntax comparison of Polars and Pandas for various operations.

It is clear that Polars API is extremely similar to Pandas'.

Thus, contrary to common belief, the transition from Pandas to Polars is not that intimidating and tedious.

If you know Pandas, you (mostly) know Polars.

In most cases, the transition will require minimal code updates.

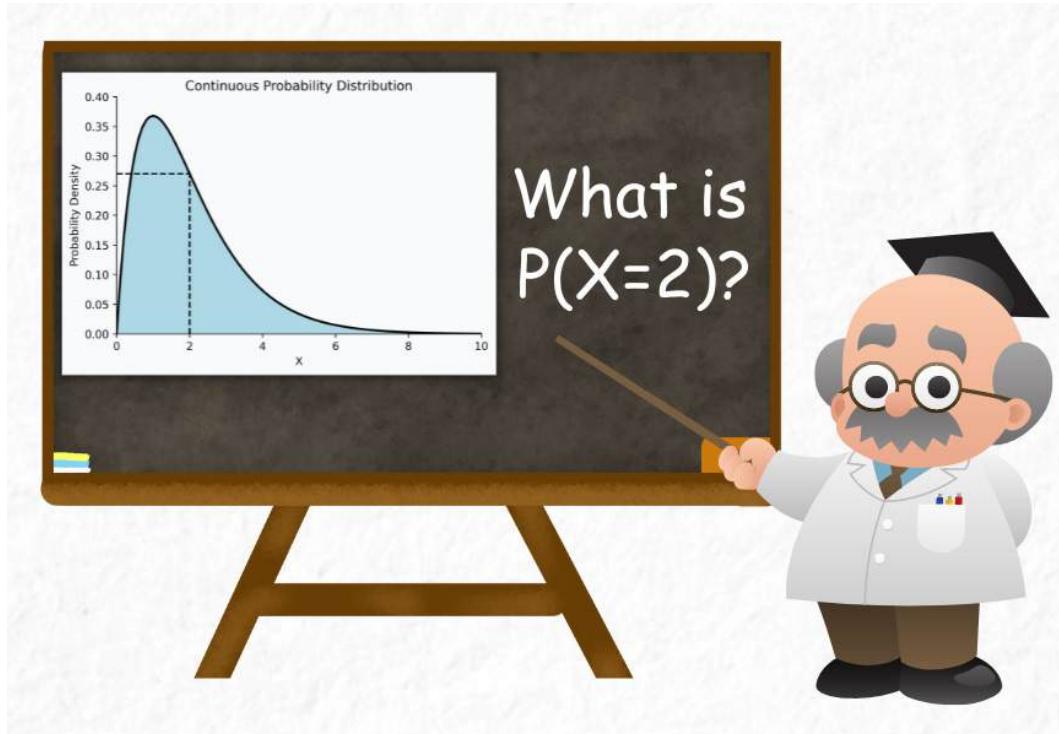
But you get to experience immense speed-ups, which you don't get with Pandas.

I recently did a comprehensive benchmarking of Pandas and Polars, which you can read here: [Pandas vs Polars — Run-time and Memory Comparison](#).

👉 Over to you: What are some other faster alternatives to Pandas that you are aware of?



# The Most Common Misconception About Continuous Probability Distributions



This issue has many mathematical formulations.

Please read it here: <https://www.blog.dailydoseofds.com/p/the-most-common-misconception-about-470>



# Don't Overuse Scatter, Line and Bar Plots. Try These Four Elegant Alternatives.

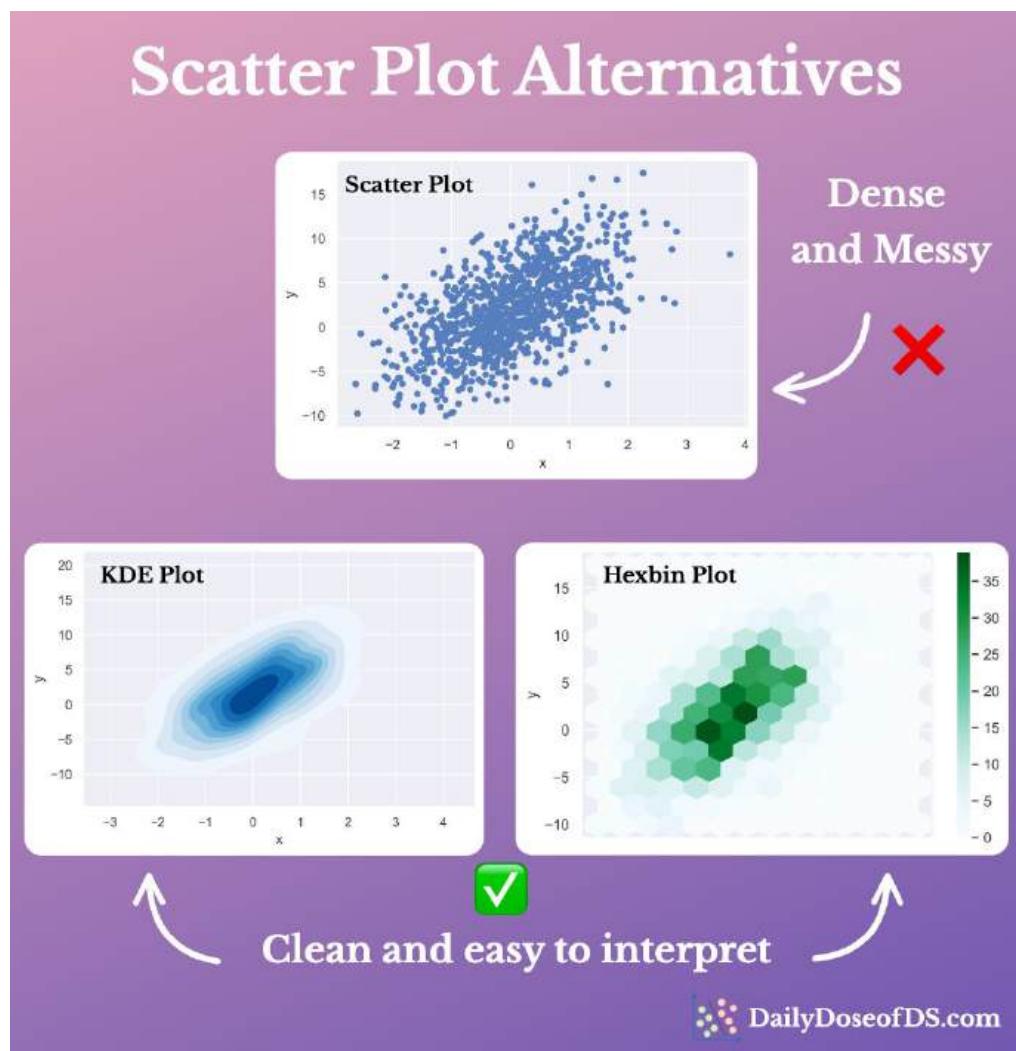
Scatter, bar, and line plots are the three most commonly used plots to visualize data.

While these plots do cover a wide variety of visualization use cases, many data scientists use them excessively in every possible place.

Here are some alternatives that can supercharge your visualizations.

## Scatter plot alternatives

When you have thousands of data points, scatter plots can get too dense to interpret.





Instead, you can replace them with Hexbin or KDE plots.

Hexbin plots bin the area of a chart into hexagonal regions. Each region is assigned a color intensity based on the method of aggregation used (the number of points, for instance).

A KDE plot illustrates the distribution of a set of points in a two-dimensional space.

A contour is created by connecting points of equal density. In other words, a single contour line depicts an equal density of data points.

## Bar plot alternative

When you have many categories to depict, the plot can easily get cluttered and messy.



Instead, you can replace them with Dot plots. They are like scatter plots but with one categorical and one continuous axis.

## Line/Bar plot alternative

When visualizing the change in value over time, it is difficult to depict incremental changes with a bar/line plot.



## Bar/Line Plot Alternative



Naive  
Plots  
✗



Elegant color-encoded changes over time

Bar/Line plot alternative — Waterfall chart

Instead, try Waterfall charts. The changes are automatically color-coded, making them easier to interpret.

👉 Over to you: What are some other elegant alternatives to commonly used plots?

I have written a Medium article on this if you are interested in learning more: [Medium Blog](#).



# CNN Explainer: Interactively Visualize a Convolutional Neural Network

The screenshot shows a web-based tool for visualizing CNN architectures. At the top, there's a navigation bar with various icons and a search bar. Below it, a large grid of small images represents the input data. To the right of the input grid, a detailed diagram of a neural network layer is shown. This diagram includes an 'Input (26, 26)' section, a 'ReLU Activation' section with a mathematical formula  $\max(0, x) = \boxed{0}$ , and an 'Output (26, 26)' section. A color scale at the bottom indicates pixel values from -0.76 to 0.76. The overall interface is clean and interactive, designed to help users understand how CNNs process and transform images.

Convolutional Neural Networks (CNNs) have been a revolutionary deep learning architecture in computer vision.

On a side note, we know that CNNs are mostly used for computer vision tasks etc. But they are also used in NLP applications too. [Further reading.](#)

The core component of a CNN is convolution, which allows them to capture local patterns, such as edges and textures, and helps in extracting relevant information from the input.

Yet, at times, understanding:

1. how CNNs internally work
2. how inputs are transformed
3. what is the representation of the image after each layer
4. how convolutions are applied
5. how pooling operation is applied
6. how the shape of the input changes, etc.



...is indeed difficult.

If you have ever struggled to understand CNN, you should use [CNN Explainer](#).

It is an incredible interactive tool to visualize the internal workings of a CNN.

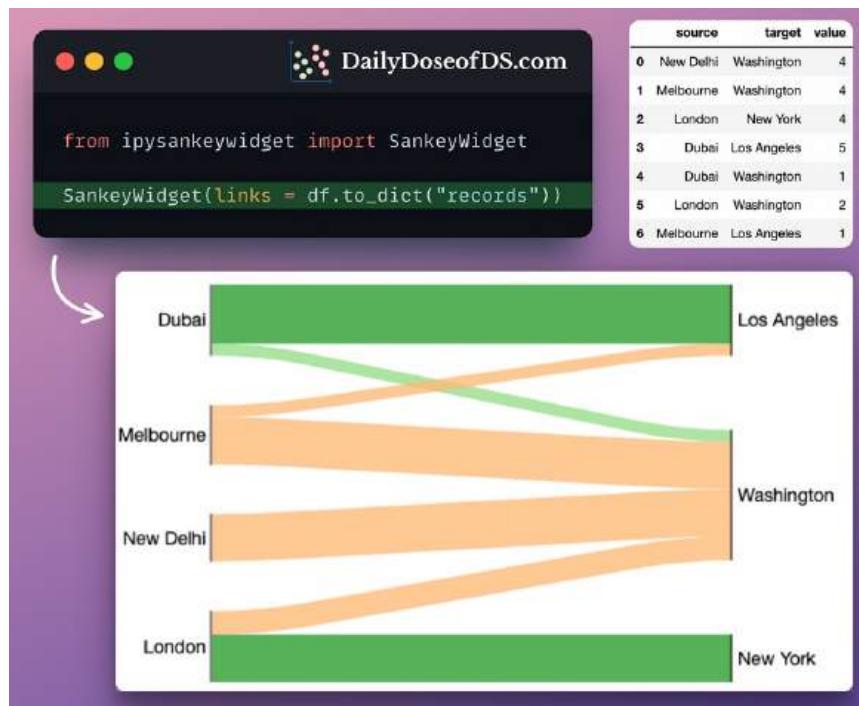
Essentially, you can play around with different layers of a CNN and visualize how a CNN applies different operations.

Try it here: [CNN Explainer](#).

👉 Over to you: What are some interactive tools to visualize different machine learning models/architectures, that you are aware of?



# Sankey Diagrams: An Underrated Gem of Data Visualization



Many tabular data analysis tasks can be interpreted as a flow between the source and a target.

Instead of manually analyzing tabular data, try to represent them as Sankey diagrams.

They immensely simplify the data analysis process.

For instance, from the diagram above, one can quickly infer that:

1. Washington hosts flights from all origins
2. New York only receives passengers from London
3. Majority of flights in Los Angeles come from Dubai
4. All flights from New Delhi go to Washington

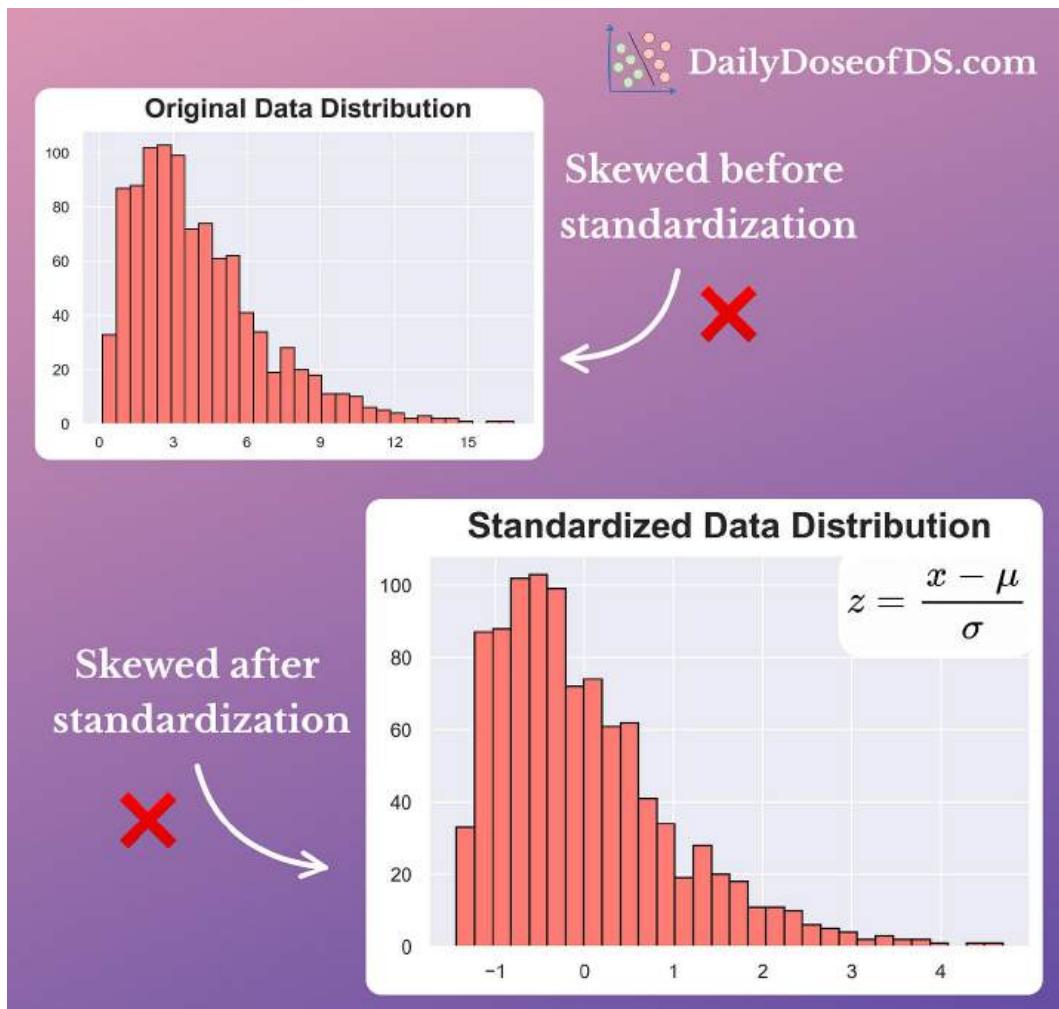
Now imagine doing that by just looking at the tabular data.

- 1) It will be time-consuming
- 2) You may miss out on a few insights

👉 Over to you: What are some other ways you use to simplify data analysis?



# A Common Misconception About Feature Scaling and Standardization



Feature scaling and standardization are common ways to alter a feature's range.

For instance:

- MinMaxScaler shrinks the range to [0,1]:

$$x' = \frac{x - x_{min}}{x_{max} - x_{min}}$$

- Standardization makes the mean zero and standard deviation one, etc.

$$z = \frac{x - \mu}{\sigma}$$

It is desired because it prevents a specific feature from strongly influencing the model's output. What's more, it ensures that the model is more robust to variations in the data.



In the image above, the scale of Income could massively impact the overall prediction. Scaling (or standardizing) the data to a similar range can mitigate this and improve the model's performance.

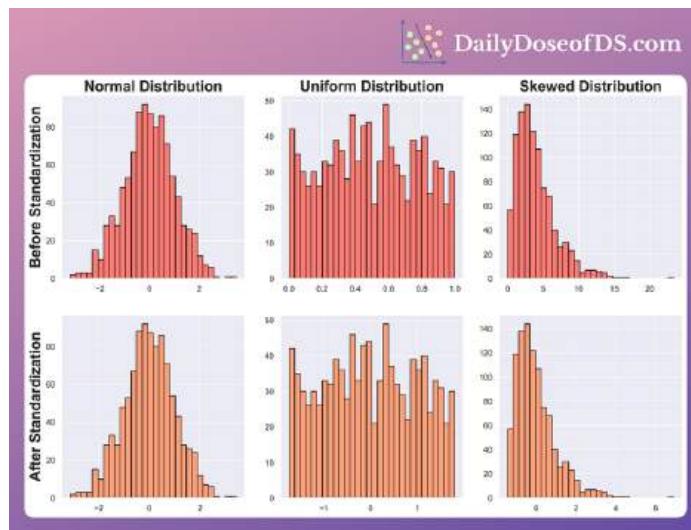
Yet, contrary to common belief, they NEVER change the underlying distribution.

Instead, they just alter the range of values.

Thus:

1. Normal distribution → stays Normal
2. Uniform distribution → stays Uniform
3. Skewed distribution → stays Skewed
4. and so on...

We can also verify this from the below illustration:





If you intend to eliminate skewness, scaling/standardization won't help.

Try feature transformations instead.

I recently published a post on various transformations, which you can read here: [Feature transformations](#).

👉 Over to you: While feature scaling is immensely helpful, some ML algorithms are unaffected by the scale. Can you name some algorithms?



# 7 Elegant Usages of Underscore in Python

## 1) Obtain Last Computed Value

```
>>> 5+10
15

>>> _ # returns last value
15
```

Get last computed value using `_`

```
>>> 5+10
15

>>> _ + 10
25
```

DailyDoseofDS.com

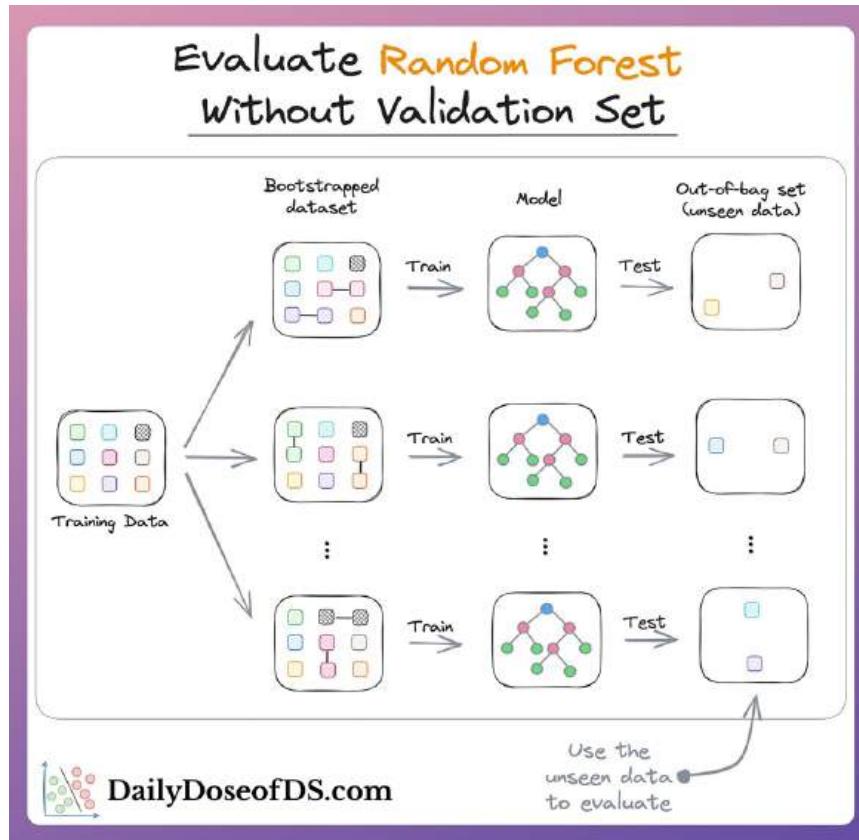
Underscore offers many functionalities in Python.

The above animation highlights 7 of the must-know usages among Python programmers.

Read the full issue here: <https://www.blog.dailydoseofds.com/p/7-elegant-usages-of-underscore-in>



# Random Forest May Not Need An Explicit Validation Set For Evaluation



We all know that ML models should not be evaluated on the training data. Thus, we should always keep a held-out validation/test set for evaluation.

But random forests are an exception to that.

In other words, you can reliably evaluate a random forest using the training set itself.

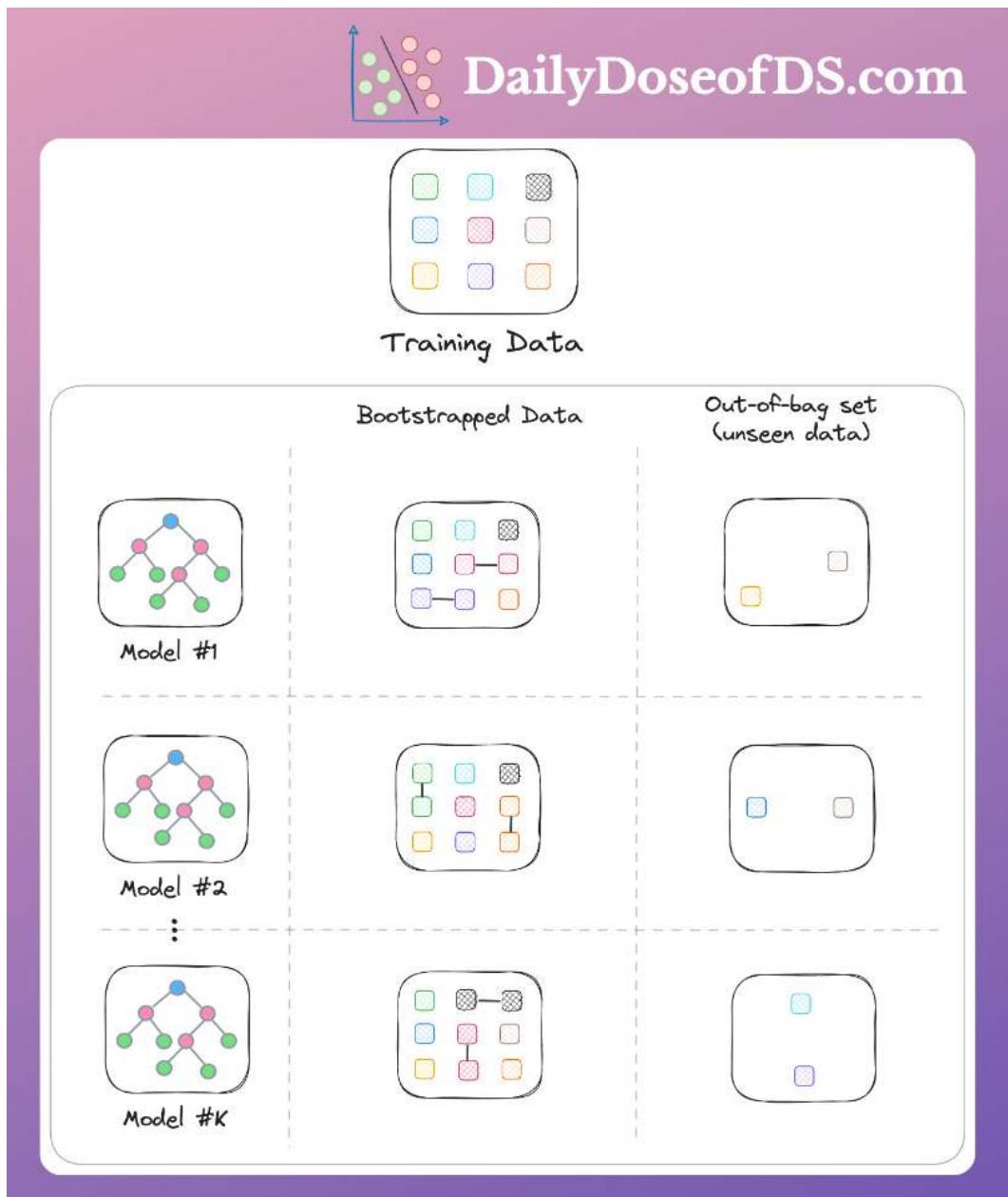
Confused?

Let me explain.

To recap, a random forest is trained as follows:

1. First, create different subsets of data with replacement.
2. Next, train one decision tree per subset.
3. Finally, aggregate all predictions to get the final prediction.

Clearly, **EVERY** decision tree has some **unseen data points** in the entire training set.



Thus, we can use them to validate that specific decision tree.

This is also called **out-of-bag validation**.

Calculating the **out-of-bag score** for the whole random forest is simple too.

1. For every data point in the entire training set:
2. Gather predictions from all decision trees that used it as an out-of-bag sample



3. Aggregate predictions to get the final prediction
4. Finally, score all the predictions to get the out-of-bag score.

Out-of-bag validation has several benefits:

1. If you have less data, you can prevent data splitting
2. It's computationally faster than using, say, cross-validation
3. It ensures that there is no data leakage, etc.

Luckily, out-of-bag validation is neatly tied in sklearn's random forest implementation too.

```
from sklearn.ensemble import RandomForestClassifier  
  
>>> RandomForestClassifier( oob_score = True )
```

out-of-bag scoring flag

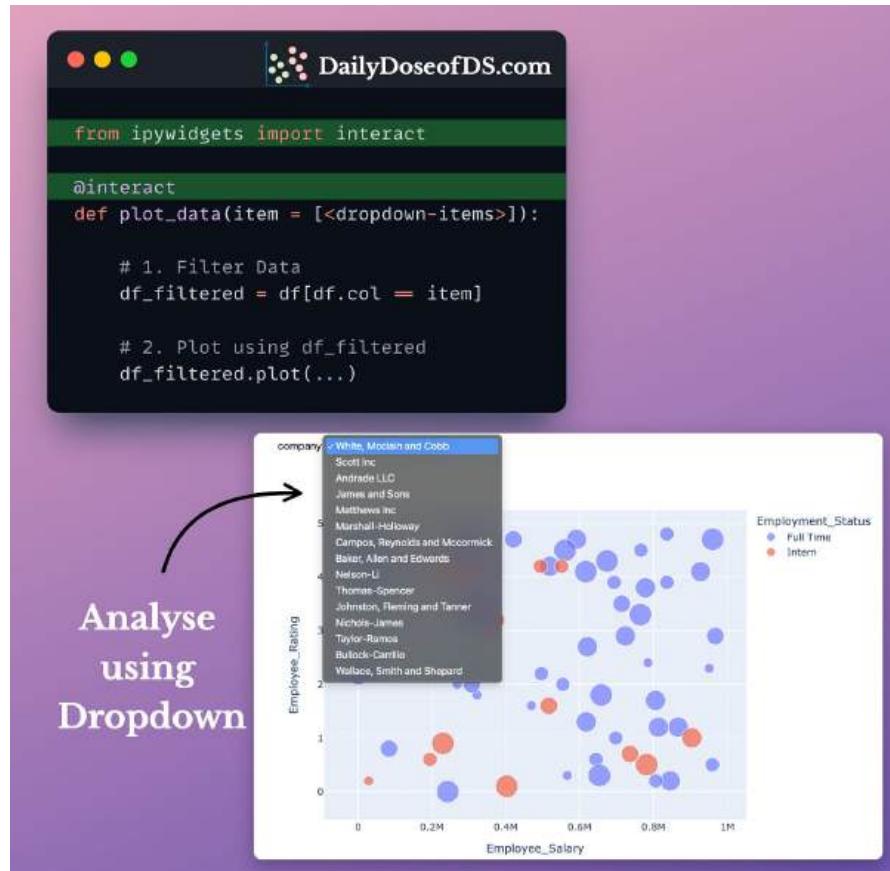
Parameter for out-of-bag scoring as specified in the [official docs](#)

👉 Over to you:

1. What are some limitations of out-of-bag validation?
2. How reliable is the out-of-bag score to tune the hyperparameters of the random forest model?



# Declutter Your Jupyter Notebook Using Interactive Controls



While using Jupyter, one often finds themselves in situations where they repeatedly modify a cell and re-run it.

This makes data exploration:

- irreproducible,
- tedious, and
- unorganized.

What's more, the notebook also gets messy and cluttered.

Instead, leverage interactive controls using IPywidgets.

A single decorator (`@interact`) allows you to add:

- sliders
- dropdowns
- text fields, and more.



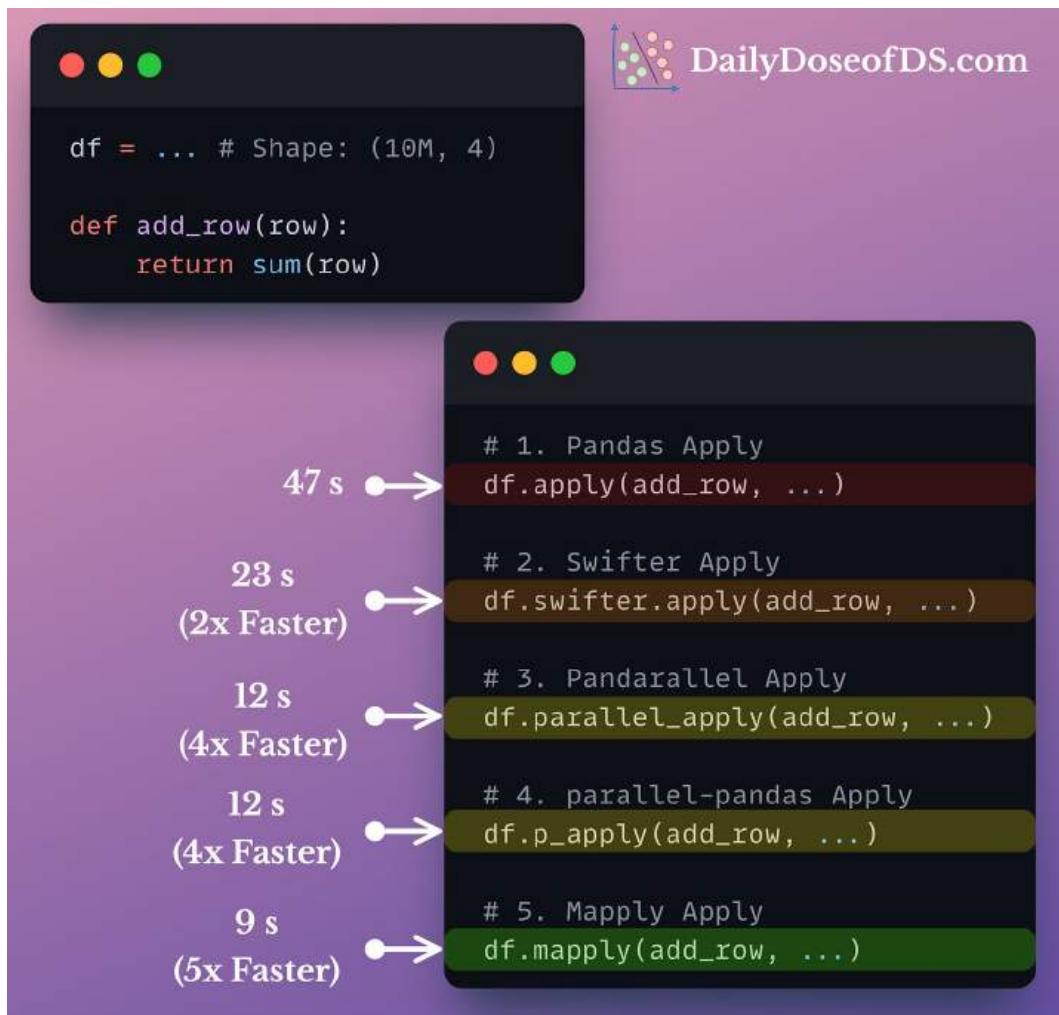
As a result, you can:

- explore your data interactively
- speed-up data exploration
- avoid repetitive cell modifications and executions
- organize your data analysis.

👉 Over to you: What are some other ways to elegantly explore data in Jupyter that you are aware of?



# Avoid Using Pandas' Apply() Method At All Times



The `apply()` method in Pandas is the most common approach to apply a function along an axis of a DataFrame/Series.

But contrary to common belief, Pandas' `apply()` method:

- is NOT vectorized
- instead, it's a glorified for-loop

Thus, it does not offer any inherent optimization and the code runs at native Python speed.

One solution is to eliminate the `apply()` method by using a vectorized approach.



But it is understandable that at times, coming up with a vectorized approach is difficult. (Here's one of my previous guides on this: [If You Are Not Able To Code A Vectorized Approach, Try This](#))

Another solution is to parallelize the `apply()` method by using external libraries.

The image above compares the run-time of alternatives that support parallelization.

It is evident that Pandas' `apply()` is not the optimal way to apply a method.

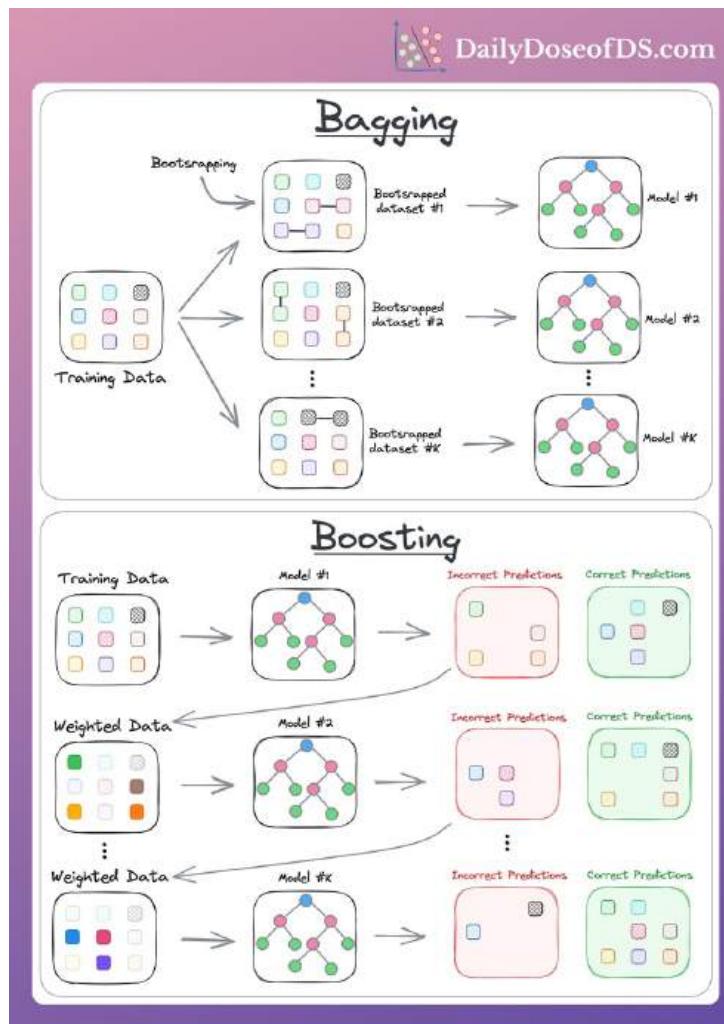
Get started with these libraries here:

- **Swifter:** <https://github.com/jmcarpenter2/swifter>
- **Pandarallel:** <https://github.com/nalepae/pandarallel>
- **Parallel Pandas:** <https://pypi.org/project/parallel-pandas/>
- **Mapply:** <https://pypi.org/project/mapply/>

👉 Over to you: What are some other techniques you commonly use to optimize Pandas' operations?



# A Visual and Overly Simplified Guide To Bagging and Boosting



Many folks often struggle to understand the core essence of Bagging and boosting. Here's a simplified visual guide depicting what goes under the hood.

In a gist, an ensemble combines multiple models to build a more powerful model.

They are fundamentally built on the idea that by aggregating the predictions of multiple models, the weaknesses of individual models can be mitigated. Combining models is expected to provide better overall performance.

Whenever I wish to intuitively illustrate their immense power, I use the following image:

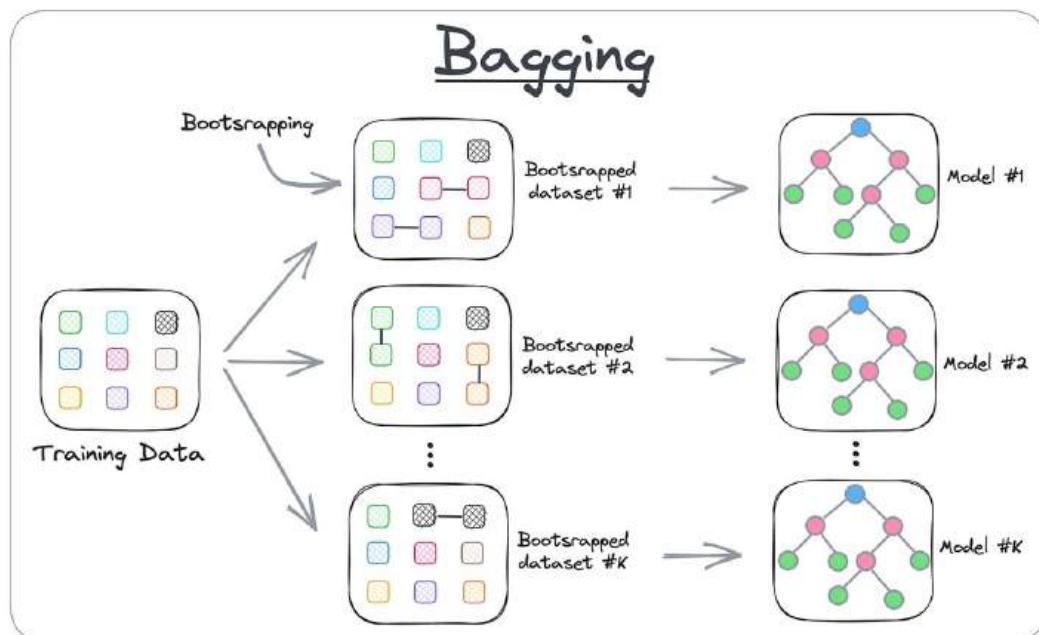


Ensembles are primarily built using two different strategies:

Bagging

Boosting

1) Bagging (short for Bootstrapped Aggregation):



Bagging illustration

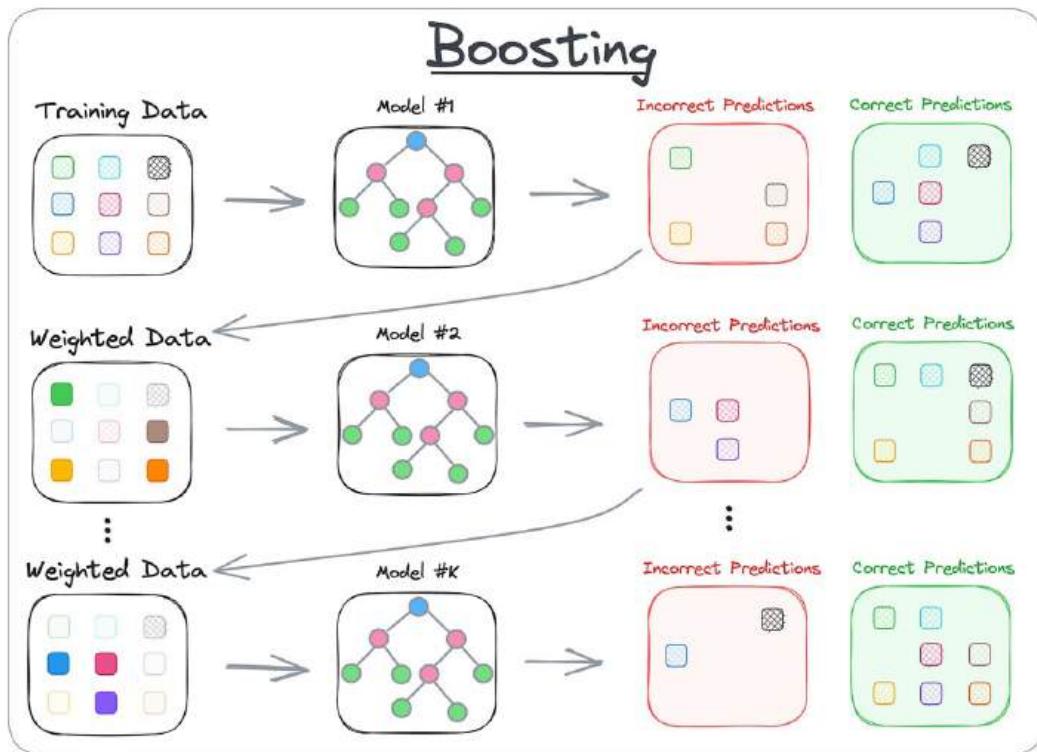
- creates different subsets of data (this is called bootstrapping)
- trains one model per subset
- aggregates all predictions to get the final prediction

Some common models that leverage Bagging are:

- Random Forests
- Extra Trees



## 2) Boosting:



Boosting illustration

- is an iterative training process
- the subsequent model puts more focus on misclassified samples from the previous model
- the final prediction is a weighted combination of all predictions

Some common models that leverage Boosting are:

- XGBoost,
- AdaBoost, etc.

Overall, ensemble models significantly boost the predictive performance compared to using a single model. They tend to be more robust, generalize better to unseen data, and are less prone to overfitting.

👉 Over to you: What are some challenges/limitations of using ensembles?



# 10 Most Common (and Must-Know) Loss Functions in ML

10 Most Common Loss Functions in Machine Learning by Avi Chawla		
Loss Function Name	Description	Function
<b>Regression Losses</b>		
Mean Bias Error	Captures average bias in prediction. But is rarely used for training.	$\mathcal{L}_{MBE} = \frac{1}{N} \sum_{i=1}^N (y_i - f(x_i))$
Mean Absolute Error	Measures absolute average bias in prediction. Also called L1 Loss.	$\mathcal{L}_{MAE} = \frac{1}{N} \sum_{i=1}^N  y_i - f(x_i) $
Mean Squared Error	Average squared distance between actual and predicted. Also called L2 Loss.	$\mathcal{L}_{MSE} = \frac{1}{N} \sum_{i=1}^N (y_i - f(x_i))^2$
Root Mean Squared Error	Square root of MSE. Loss and dependent variable have same units.	$\mathcal{L}_{RMSE} = \sqrt{\frac{1}{N} \sum_{i=1}^N (y_i - f(x_i))^2}$
Huber Loss	A combination of MSE and MAE. It is parametric loss function.	$\mathcal{L}_{Huber} = \begin{cases} \frac{1}{2}(y_i - f(x_i))^2 & :  y_i - f(x_i)  \leq \delta \\ \delta( y_i - f(x_i)  - \frac{1}{2}\delta) & : otherwise \end{cases}$
Log Cosh Loss	Similar to Huber Loss + non-parametric. But computationally expensive.	$\mathcal{L}_{LogCosh} = \frac{1}{N} \sum_{i=1}^N \log(\cosh(f(x_i) - y_i))$
<b>Classification Losses (Binary + Multi-class)</b>		
Binary Cross Entropy (BCE)	Loss function for binary classification tasks.	$\mathcal{L}_{BCE} = \frac{1}{N} \sum_{i=1}^N y_i \cdot \log(p(x_i)) + (1 - y_i) \cdot \log(1 - p(x_i))$
Hinge Loss	Penalizes wrong and right (but less confident) predictions. Commonly used in SVMs.	$\mathcal{L}_{Hinge} = \max(0, 1 - (f(x) \cdot y))$
Cross Entropy Loss	Extension of BCE loss to multi-class classification.	$\mathcal{L}_{CE} = -\frac{1}{N} \sum_{i=1}^N \sum_{j=1}^M y_{ij} \cdot \log(f(x_{ij}))$ <small>N : samples; M : classes</small>
KL Divergence	Minimizes the divergence between predicted and true probability distribution	$\mathcal{L}_{KL} = \sum_{i=1}^N y_i \cdot \log\left(\frac{y_i}{f(x_i)}\right)$

Loss functions are a key component of ML algorithms.

They specify the objective an algorithm should aim to optimize during its training. In other words, loss functions tell the algorithm what it should be trying to minimize or maximize in order to improve its performance.

Therefore, knowing about the most common loss functions in machine learning is extremely crucial.

The above visual depicts the most commonly used loss functions for regression and classification tasks.



# How To Enforce Type Hints in Python?

The diagram illustrates the difference between Python's handling of type hints and Typeguard's enforcement of them.

**No type checking (Python):** A screenshot of a terminal window shows a function definition with type hints. When called with incompatible arguments (an integer and a string), it does not raise an error. Instead, it returns a boolean value, indicating a type mismatch. A red X marks this as incorrect.

```
def foo(a:int, b:float) -> bool:  
    ...  
  
>>> foo(1, "A") # expected (int, float)  
  
>>> foo(1.2, 2.1)
```

**Type mismatch raises error (Typeguard):** A screenshot of a terminal window shows the same function definition wrapped in a `@typechecked` decorator from the `typeguard` module. When called with incompatible arguments, it raises a `TypeError` stating that the argument "b" must be either a float or an int, but got a str instead. A green checkmark marks this as correct.

```
from typeguard import typechecked  
  
@typechecked # Add decorator  
def foo(a:int, b:float) -> bool:  
    ...  
  
>>> foo(1, "A")  
TypeError: type of argument "b" must be  
either float or int; got str instead
```

When writing Python functions, type hints provide an incredible way to specify explicit information about:

- the expected types of function arguments, and
- their return type.

Yet, Python NEVER enforces them.

This means that despite having type hints, a function can still accept (or return) a conflicting type.

And Python will not raise any errors/warnings if there is a type mismatch.

To enforce type hints, use Typeguard.

It provides a decorator for type-checking of Python functions.

As a result, in case of conflicting types, an error is raised.

Enforcing type hints can provide an additional layer of type safety. One can catch type violations that could only be detected at run time.

**Get started with Typeguard: [Docs](#).**

👉 Over to you: Why Python developers didn't enforce type hints? Let me know your thoughts :)



# A Common Misconception About Deleting Objects in Python

```
class MyClass:  
    def __del__(self):  
        # Invoked when object is deleted  
        print("Object Deleted!")
```

No Output

2nd delete statement  
prints *Object Deleted!*

```
1 >>> objectA = MyClass()  
2  
3 >>> objectB = objectA  
4  
5 >>> del objectA  
6  
7 >>> del objectB  
8 "Object Deleted!"
```

Many Python programmers believe that executing `del object` always deletes an object.

But that is NOT true.

## Some background:

The `__del__` magic method is used to define the behavior when an object is about to be destroyed by the Python interpreter.

It is invoked automatically right before an object's memory is deallocated.

Thus, by defining this method in your class, you can add a custom functionality when an object is deleted.

As shown in line 5, deleting the object didn't output anything specified in `__del__`.

This means that the object was not deleted.

But it did produce an output the second time (line 7-8).



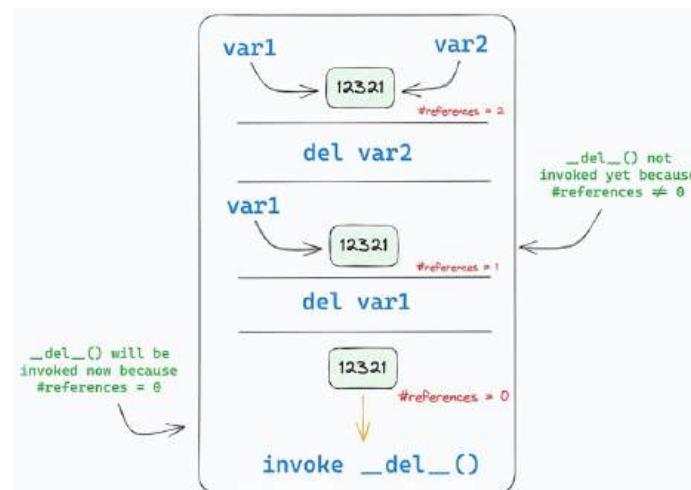
Why does it happen?

When we execute `del var`:

- Python does not always delete an object
- Instead, it deletes the name `var` from the current scope
- Finally, it reduces the number of references to that object by 1

It is ONLY when the number of references to an object becomes 0 that an object is deleted.

Consequently, `__del__` is executed.



This explains the behavior in the below code.

A screenshot of a Jupyter Notebook cell. The code defines a class `MyClass` with a `__del__` method that prints "Object Deleted!". The cell shows the following execution:

```
class MyClass:  
    def __del__(self):  
        # Invoked when object is deleted  
        print("Object Deleted!")
```

The first execution of `del objectA` (line 5) results in "No Output" (indicated by a red X). The second execution of `del objectB` (line 7) results in the output "Object Deleted!" (indicated by a green checkmark).

When we deleted the first reference (`del objectA`), the same object was still referenced by `objectB`.



Thus, at that time, the number of references to that object was non-zero.

But when we deleted the second reference (`del objectB`), Python lost all references to that object.

As a result, the `__del__` magic method was invoked.

So remember...

`del var` does not always invoke the `__del__` magic method and delete an object.

Instead, here's what it does:

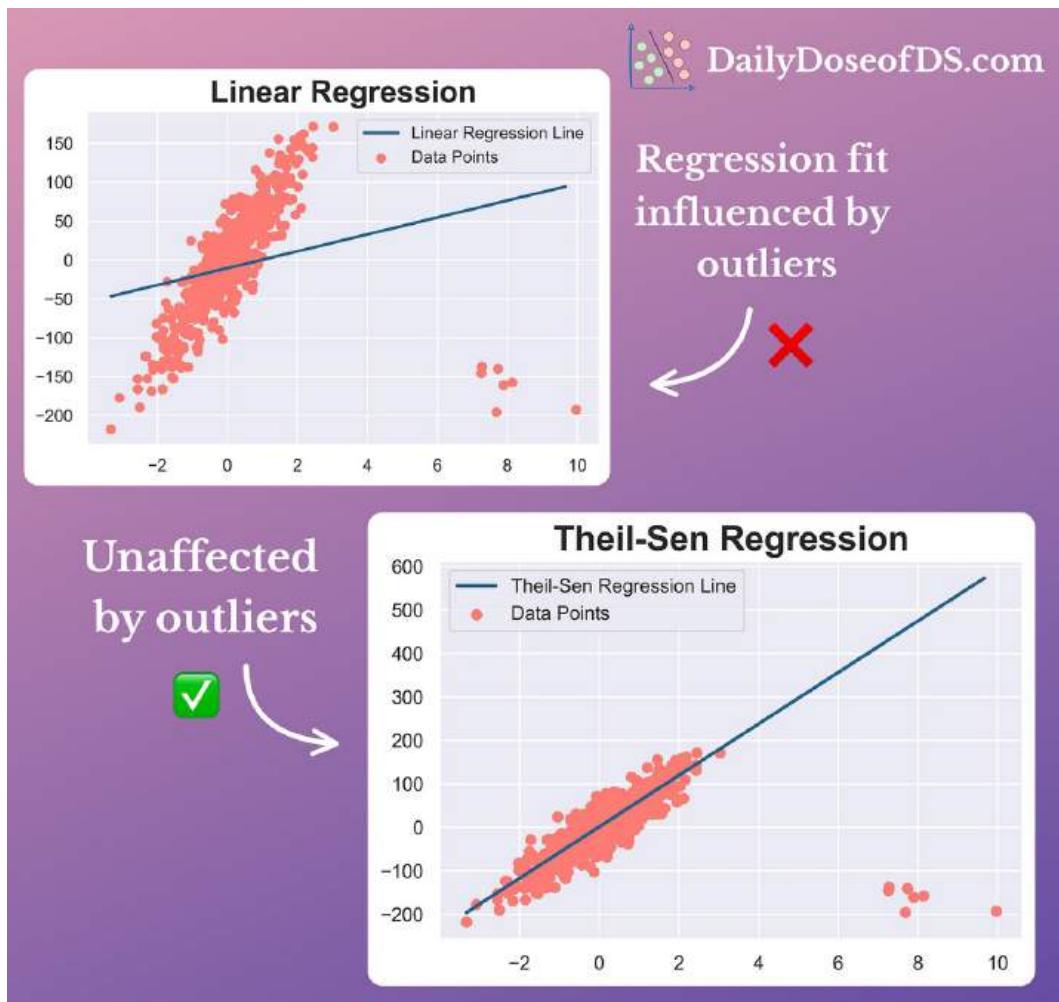
- First, it removes the variable name (`var`) from the current scope.
- Next, it reduces the number of references to that object by 1.

When the reference count becomes zero, the object is deleted.

👉 Over to you: What are some other common misconceptions in Python?



# Theil-Sen Regression: The Robust Twin of Linear Regression



Linear Regression is the most widely used ML algorithm.

But it is sensitive to outliers.

In fact, even a few outliers can significantly impact its performance.

Instead, try TheilSenRegressor. It is an outlier-robust regression algorithm.

It works as follows:

- Select a subset of data
- Fit a least squares model
- Record model weights
- Repeat



The final weights are the spatial median (or L1 Median) of all models.

The spatial median represents the “middle” or central location in a multidimensional space.

Essentially, the objective is to find a point in the same multidimensional space which minimizes the sum of the absolute differences between itself and all other points (weight vectors, in this case).

As shown above, while Linear Regression is influenced by outliers, Theil-Sen Regression isn't.

Having said that, it is always recommended to experiment with many robust methods and see which one fits your data best.

👉 Get started with Theil-Sen Estimator: [Sklearn Docs](#).

👉 Over to you: What are some other popular models that are robust to outliers? Let me know :)



# What Makes The Join() Method Blazingly Faster Than Iteration?

In [1]: words = ["Best", "way", "to", "join", "strings", "in", "Python"]

**Approach #1: Iteration**

```
In [2]: %timeit
sentence = ""
for word in words:
    sentence += word + " "
sentence = sentence[:-1]
774 ns ± 33.9 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

**Approach #2: Join()**

```
In [3]: %timeit sentence = " ".join(words)
132 ns ± 0.952 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

Join() is  
6x Faster

There are two popular ways to concatenate multiple strings:

Iterating and appending them to a single string.

Using Python's in-built `join()` method.

But as shown above, the 2nd approach is significantly faster than the 1st approach.

Here's why (or maybe stop reading here and try to guess before you read ahead).

---

When iterating, Python naively executes the instructions it comes across.

Thus, it does not know (beforehand):

number of strings it will concatenate

number of white spaces it will need

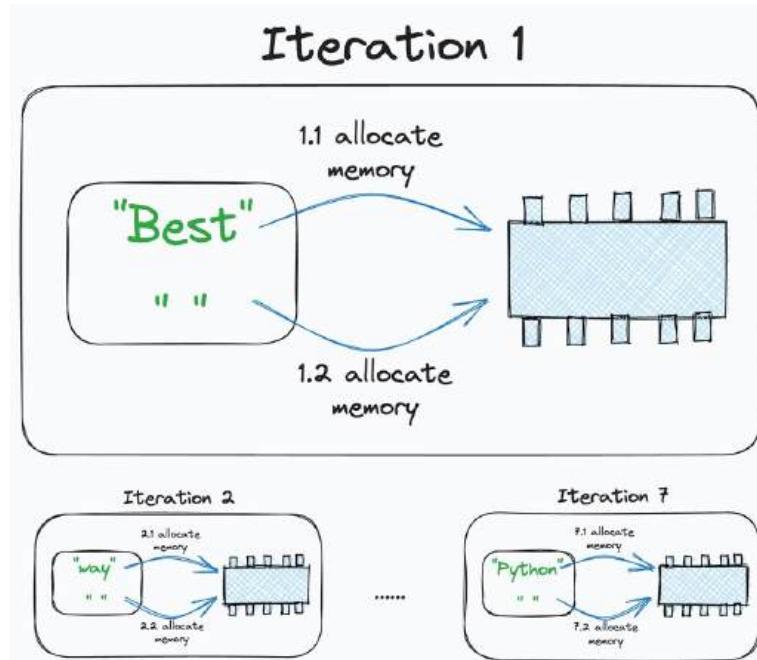
In other words, iteration inhibits the scope for optimization.

As a result, at every iteration, Python asks for a memory allocation of:



the string at the current iteration

the white space added as a separator

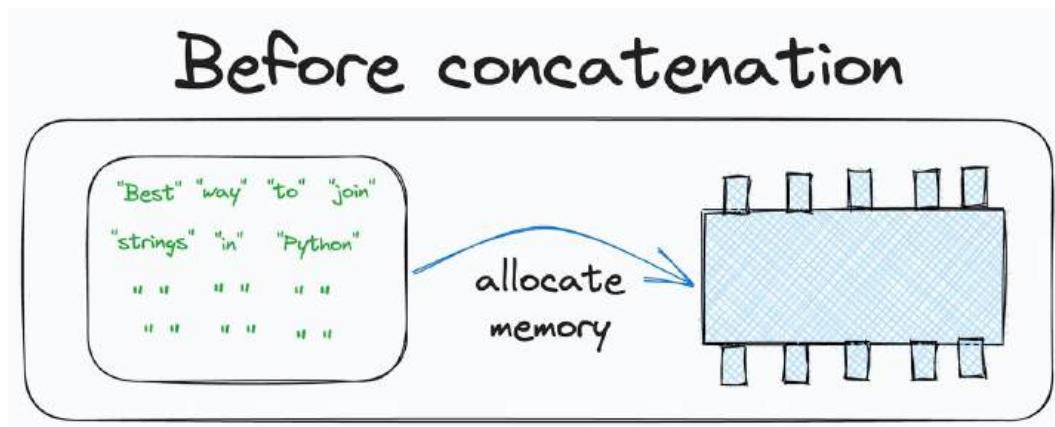


Memory allocation at each iteration

This leads to repeated calls to memory. To be precise, the number of calls, in this case, is two times the size of the list.

However, with `join()`, Python precisely knows (beforehand):

- number of strings it will be concatenating
- number of white spaces it will need



All these are applied for allocation in a single call and are available upfront before concatenation.



To summarize:

- with iteration, the number of memory allocation calls is 2x the list's size.
- with `join()`, the number of memory allocation calls is just one.

This explains the significant difference in their run-time.

This post is also a reminder to always prefer specific methods over a generalized approach. What do you think?

👉 Over to you: What other ways do you commonly use to optimize native Python code?



# A Major Limitation of NumPy Which Most Users Aren't Aware Of

The screenshot shows a Jupyter Notebook cell with the following code:

```
import numpy as np
import numexpr as ne

a = np.random.random(10**7)
b = np.random.random(10**7)
```

Below the code, there are two sections: **NumPy** and **Numexpr**.

**NumPy** output:

```
%timeit np.cos(a) + np.sin(b)
142 ms ± 257 µs per loop
```

**Numexpr** output:

```
%timeit ne.evaluate("cos(a) + sin(b)")
32.5 ms ± 229 µs per loop
```

A callout bubble points from the Numexpr time to the text **~5x Faster than NumPy**, which is accompanied by a small rocket ship icon.

NumPy undoubtedly offers

- extremely fast, and
- optimized operations.

Yet, it DOES NOT support parallelism.

This provides further scope for run-time improvement.

Numexpr is a fast evaluator for NumPy expression, which uses:

- multi-threading
- just-in-time compilation

The speedup offered by Numexpr is evident from the image above.

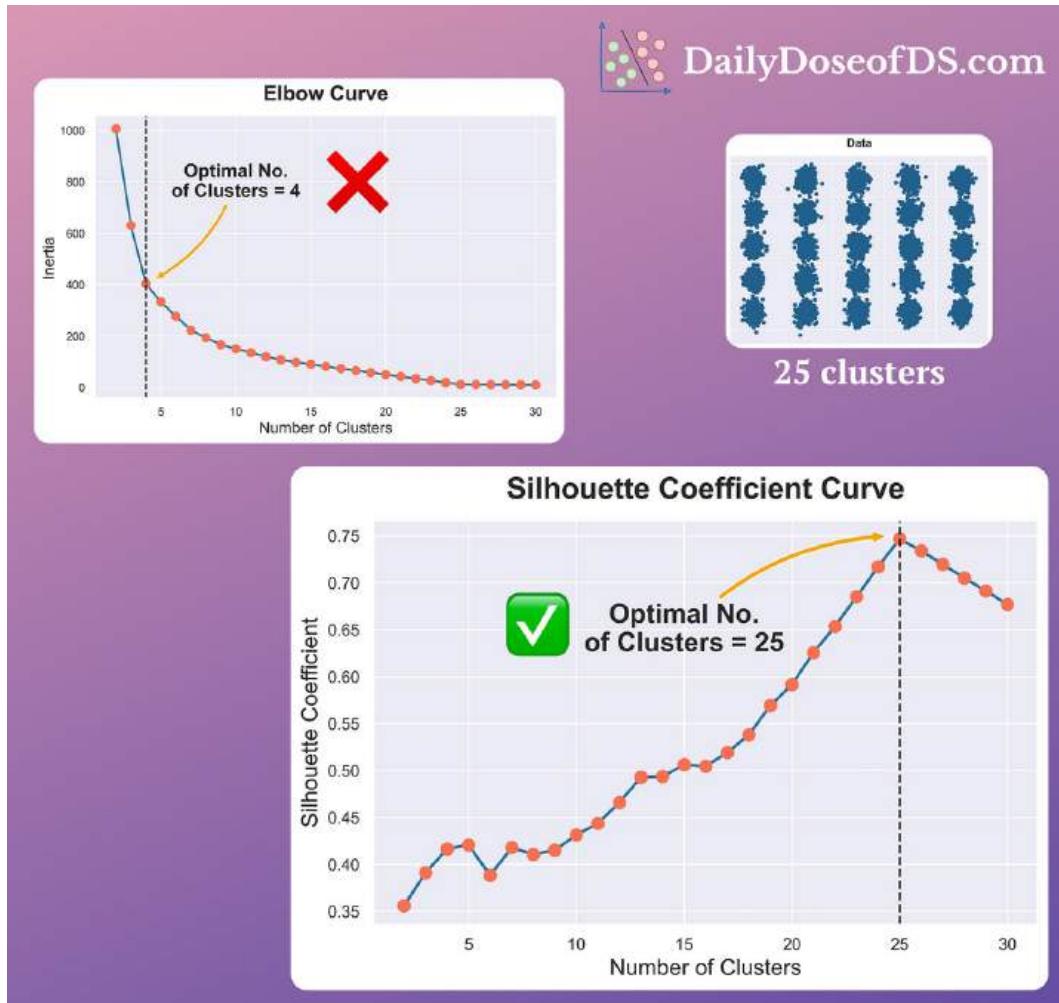
Depending upon the complexity of the expression, the speed-ups can range from 0.95x and 20x.

Read more: [Documentation](#).

👉 Over to you: What are some other ways to speedup NumPy computation?



# The Limitations Of Elbow Curve And What You Should Replace It With



We commonly use the Elbow curve to determine the number of clusters ( $k$ ) for KMeans.

However, the Elbow curve:

- has a subjective interpretation
- involves ambiguity in determining the Elbow point accurately
- only considers a within-cluster distance, and more.

Silhouette score is an alternative measure used to evaluate clustering quality.

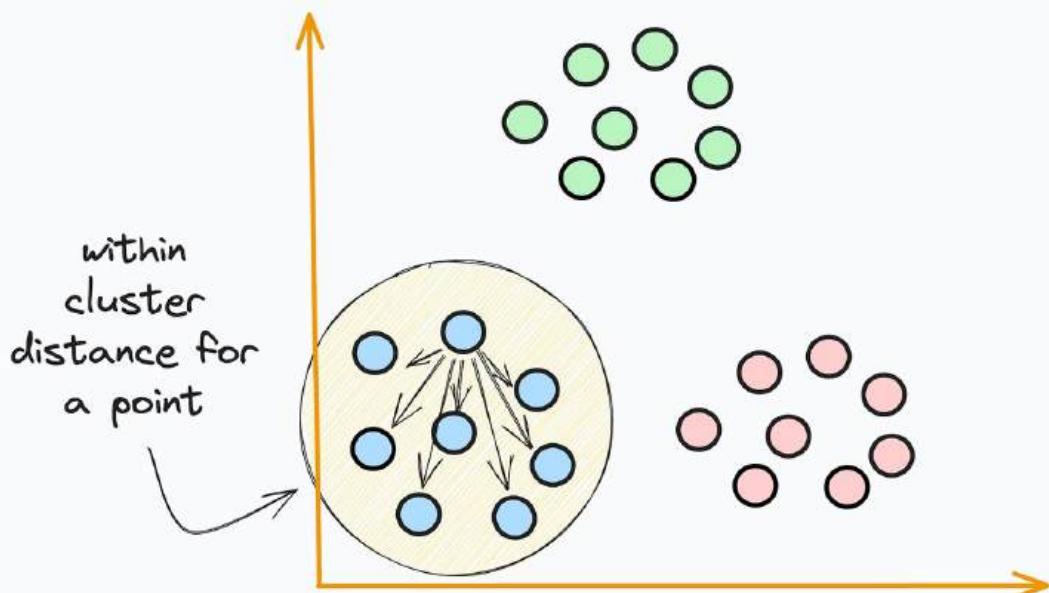
It is computed as follows:

For every data point ( $i$ ), find:

$a(i)$ : average distance to every other data point within the cluster



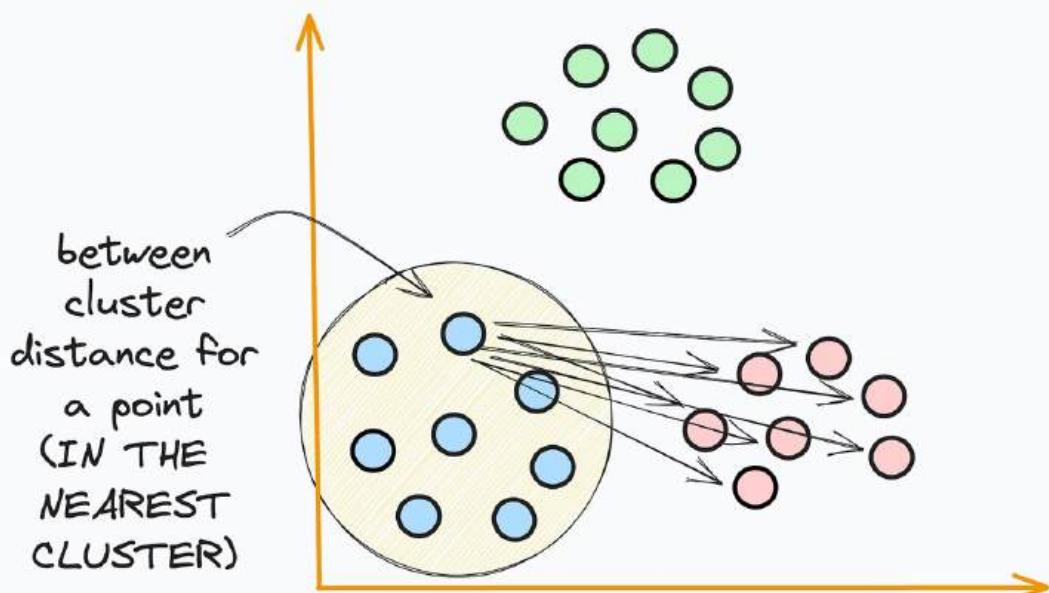
## calculate $a(i)$



Calculate  $a(i)$

$b(i)$ : average distance to every data point in the nearest cluster.

## calculate $b(i)$



Calculate  $b(i)$



- Silhouette score for a specific data point ( $i$ ) is:

$$s_i = \frac{b_i - a_i}{\max(a_i, b_i)}$$

- Silhouette score for the whole clustering is:

$$s = \frac{1}{n} \sum_i^n s_i$$

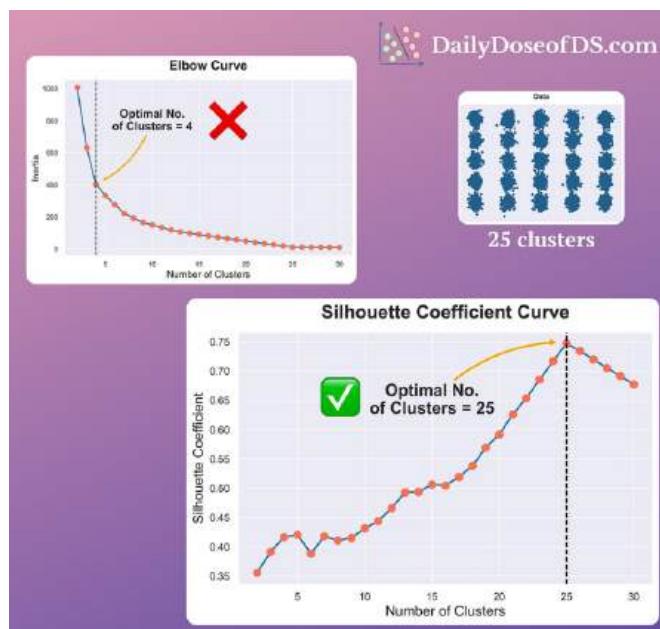
Some properties of the Silhouette score are:

- it ranges from [-1,1]
- a higher score indicates better clustering
- it can be used as an evaluation metric for clustering in the absence of ground truth labels

In contrast to the Elbow curve, the Silhouette score:

- provides a quantitative (and objective) measure
- involves no ambiguity
- considers BOTH within-cluster and between-cluster distance.

The visual below compares the Elbow curve and the Silhouette plot.





It's clear that the Elbow curve is highly misleading and inaccurate.

In a dataset with 25 clusters:

- The Elbow curve depicts **4** as the number of optimal clusters.
- The Silhouette curve depicts **25** as the number of optimal clusters.

Get started with Silhouette score here: [Sklearn Docs](#).

👉 Over to you: What are some other measures to evaluate clustering quality?



# 21 Most Important (and Must-know) Mathematical Equations in Data Science



## 21 Most Important Equations in Data Science

by Avi Chawla

### 1. Gradient Descent

$$\theta_{j+1} = \theta_j - \alpha \frac{\partial J(\theta)}{\partial \theta_j}$$

### 2. Normal Distribution

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

### 3. Sigmoid

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

### 4. Linear Regression

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n$$

### 5. Cosine Similarity

$$\text{similarity}(\mathbf{A}, \mathbf{B}) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|}$$

### 6. Naive Bayes

$$P(C_k | x_1, x_2, \dots, x_n) = \frac{P(C_k) \prod_{i=1}^n P(x_i | C_k)}{P(x_1, x_2, \dots, x_n)}$$

### 7. KMeans

$$J(c, \mu) = \sum_{i=1}^m \|x^{(i)} - \mu_{c^{(i)}}\|^2$$

### 8. Log Loss

$$\text{LogLoss}(y, \hat{y}) = -\frac{1}{m} \sum_{i=1}^m (y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}))$$

### 9. MSE

$$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

### 10. MSE

$$MSE = \text{bias}^2 + \text{variance}$$

### 11. MSE + L2 Regularization

$$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2 + \lambda \sum_{j=1}^M \theta_j^2$$

### 12. Entropy

$$H = - \sum_{i=1}^n P(c_i) \log_2 P(c_i)$$

### 13. Softmax

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^N e^{x_j}}$$

### 14. Ordinary Least Squares

$$\hat{\beta} = (X^T X)^{-1} X^T y$$

### 15. Correlation

$$r = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2} \sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}}$$

### 16. Z-score

$$z = \frac{x - \mu}{\sigma}$$

### 17. MLE

$$\hat{\theta}_{\text{MLE}} = \arg \max_{\theta} \prod_{i=1}^n f(x_i; \theta)$$

### 18. Eigen Vectors

$$Av = \lambda v$$

### 19. R2

$$R^2 = 1 - \frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{\sum_{i=1}^n (y_i - \bar{y})^2}$$

### 20. F1 Score

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

### 21. Expected Value

$$\mathbb{E}[X] = x_1 p_1 + x_2 p_2 + \dots + x_n p_n$$

Mathematics sits at the core of statistics and data science. It offers the essential framework for understanding data.

Thus, understanding the fundamental mathematical formulations in data science is highly important.



This visual depicts some of the most important equations (in no specific order) used in Data Science and Statistics.

**Gradient Descent:** An optimization algorithm used to minimize the cost function. It helps us find the optimal parameters for ML models.

**Normal Distribution:** A probability distribution that forms a bell curve and is often used to model and analyze data in statistics.

**Sigmoid:** A function that maps input values to a range between 0 and 1. It is commonly used in logistic regression to make predictions.

**Linear Regression:** A statistical model used to model a linear relationship between independent and dependent variables.

**Cosine Similarity:** A measure that calculates the cosine of the angle between two vectors. It is typically used to determine the similarity between data points.

**Naive Bayes:** A probabilistic classifier based on the Bayes theorem. It assumes independence between features and is often used in classification tasks.

**KMeans:** The most popular clustering algorithm that is used to partition data points into distinct groups.

**Log Loss:** A loss function used to evaluate the performance of classification models using output probabilities.

**MSE (Mean Squared Error):** A metric that measures the average squared difference between predicted and actual values. It is commonly used to assess regression models.

**MSE + L2 Regularization:** An extension of MSE that includes L2 regularization. It is used to prevent overfitting.

**Entropy:** A measure of the uncertainty or randomness of a random variable. It is often utilized in decision trees.

**Softmax:** A function that normalizes a set of values into probabilities. It is commonly used in multiclass classification problems.

**Ordinary Least Squares:** A method for estimating the parameters in linear regression models by minimizing the sum of squared residuals.

**Correlation:** A statistical measure that quantifies the strength and direction of the **linear relationship** between two variables.



**Z-score:** A standardized value that indicates how many standard deviations a data point is from the mean.

**MLE (Maximum Likelihood Estimation):** A method for estimating the parameters of a statistical model by maximizing the likelihood of the observed data.

**Eigen Vectors:** The non-zero vectors that do not change their direction when a linear transformation is applied. It is widely used in dimensionality reduction techniques.

**R2 (R-squared):** A statistical measure that represents the proportion of variance explained by a regression model, indicating its predictive power.

**F1 Score:** A metric that combines precision and recall to evaluate the performance of binary classification models.

**Expected Value:** The weighted average value of a random variable, calculated by multiplying each possible outcome by its probability.

👉 Over to you: Of course, this is not an all-encompassing list. What other equations will you include here?



# Beware of This Unexpected Behaviour of NumPy Methods

The screenshot shows two side-by-side Jupyter Notebook environments. Both environments have a header with three colored dots (red, yellow, green) and the URL [DailyDoseofDS.com](http://DailyDoseofDS.com).

**Left Environment (Array with NaN):**

- `>>> arr`  
array([ 1., 2., 3., nan])
- `>>> np.sum(arr)`  
nan
- `>>> np.min(arr)`  
nan
- `>>> np.max(arr)`  
nan
- `>>> np.cumsum(arr)`  
array([ 1., 3., 6., nan])

**Right Environment (Non-NaN outputs):**

- `>>> arr`  
array([ 1., 2., 3., nan])
- `>>> np.nansum(arr)`  
6.0
- `>>> np.nanmin(arr)`  
1.0
- `>>> np.nanmax(arr)`  
3.0
- `>>> np.nancumsum(arr)`  
array([1., 3., 6., 6.])

Below the environments, there are two labels with arrows pointing to them:

- Nan outputs ✗** (Red X)
- Non-NaN outputs ✓** (Green Checkmark)

If a NumPy array contains NaNs, NumPy's aggregate functions (`np.mean`, `np.min`, `np.max`, etc.) return NaN.

But this may not be desired at times.

One solution is to replace the NaN entries with a default value (0).

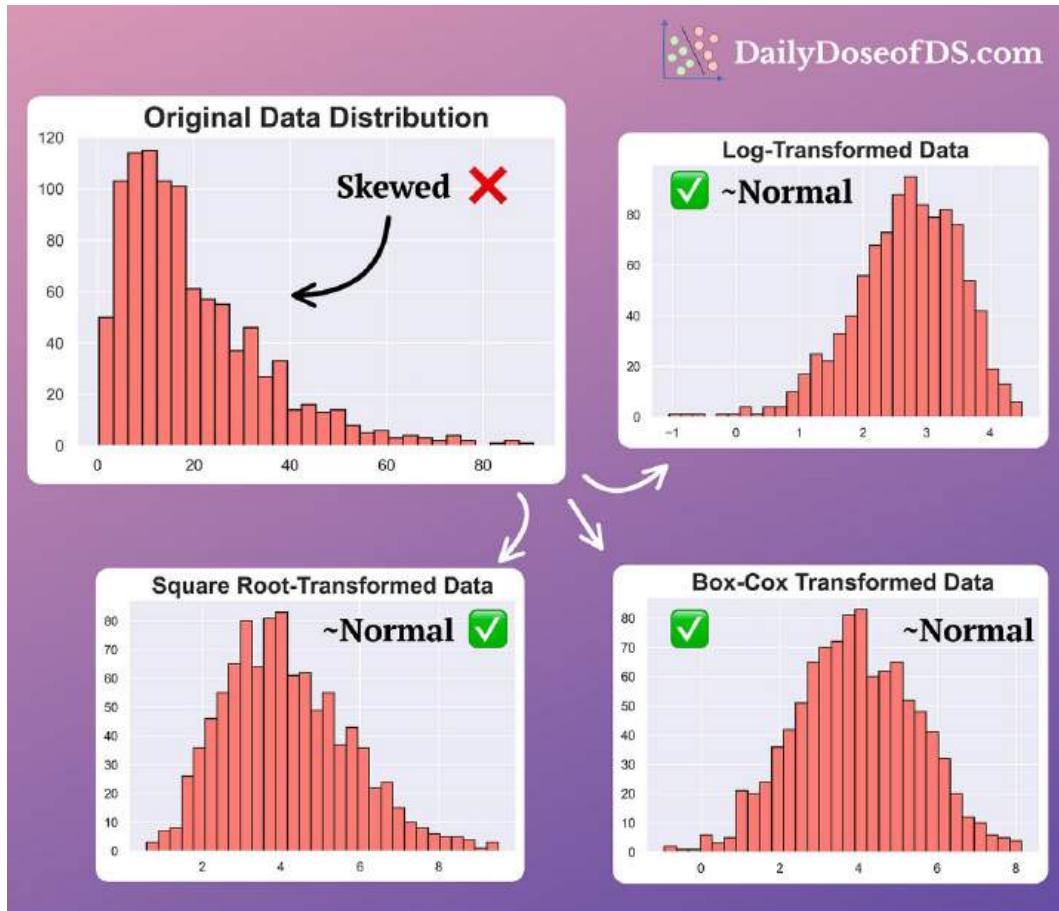
However, NumPy also provides nan-insensitive methods, such as `np.nansum`, `np.nanmin`, etc.

As a result, the output isn't influenced by the presence of NaNs.

👉 Over to you: What are some other unexpected behaviors of NumPy that you are aware of?



# Try This If Your Linear Regression Model is Underperforming



At times, skewness in data can negatively influence the predictive power of your linear regression model.

If the model is underperforming, it **may be** due to skewness.

There are many ways to handle this:

Try a different model

Transform the skewed variable

The most commonly used transformations are:

- **Log transform:** Apply the logarithmic function.
- **Sqrt transform:** Apply the square root function.
- **Box-cox transform:** You can use it using Scipy's implementation: [Scipy docs](#).



The effectiveness of the three transformations is clear from the image above.

Applying them transforms the skewed data into a (somewhat) normally distributed variable.

👉 Over to you: What are some other methods to handle skewness that you are aware of?



# Pandas vs Polars — Run-time and Memory Comparison

The chart compares Pandas and Polars across various operations. Polars consistently shows superior performance, particularly in memory usage and column selection.

Method	Pandas	Polars	*Records: 4M
Memory Usage	1.7 GBs	0.63 GBs	Polars consumes 3x Less memory
Read CSV	4.52 s	0.72 s	Polars is 6x Faster
Save to CSV	15.3 s	3.25 s	Polars is 5x Faster
Selecting Columns	36.5 ms	2.2 $\mu$ s	Polars is 16500x Faster
Filtering	130 ms	90 ms	Polars is 1.5x Faster
Grouping	217 ms	56 ms	Polars is 4x Faster
Sorting	1.1 s	1.04s	Similar Performance

Pandas is an essential library in almost all Data Science projects.

But it has many limitations.

For instance, Pandas:

- always adheres to single-core computation
- offers no lazy execution
- creates bulky DataFrames
- is slow on large datasets, and many more.

Polars is a lightning-fast DataFrame library that addresses these limitations.

It provides two APIs:



- Eager: Executed instantly, like Pandas.
- Lazy: Executed only when one needs the results.

The visual above presents a comparison of Polars and Pandas on various parameters.

It's clear that Polars is much more efficient than Pandas.

👉 Over to you: What are some other better alternatives to Pandas that you are aware of?

**Find my notebook for this post here: [GitHub](#).**

Get started with Polars: [Polars Docs](#).



# A Hidden Feature of a Popular String Method in Python

```
names = ["John", "Peter", "Mike", "Hana"]

>>> for name in names:
...     if name.startswith("Pe") or name.startswith("Mi"):
...         print(name)
```

Peter  
Mike

Multiple conditions ✗

```
names = ["John", "Peter", "Mike", "Hana"]

>>> for name in names:
...     if name.startswith(("Pe", "Mi")):
...         print(name)
```

Peter  
Mike

Single condition ✓

Here's something new and cool I recently learned about a string method in Python.

In Python, the `startswith` and `endswith` string methods are commonly used to match a substring at the start and end of the string.

However, did you know you can also pass multiple substrings as a tuple to these methods?

This prevents you from writing multiple conditions while preserving the same functionality.



# The Limitation of KMeans Which Is Often Overlooked by Many

The screenshot shows two code cells side-by-side. The top cell, titled 'sklearn.py', contains Python code for training a KMeans model on a dataset 'x\_train' with shape (500000, 1024). The bottom cell, titled 'faiss.py', contains Python code for training a KMeans model using the Faiss library on the same dataset. A callout arrow from the 'faiss.py' cell points to the 'sklearn.py' cell with the text '~20x Faster'.

```
sklearn.py:
```

```
from sklearn.cluster import KMeans  
  
kmeans = KMeans(8).fit(x_train)  
# Training Time: 162s
```

```
faiss.py:
```

```
import faiss  
  
kmeans = faiss.Kmeans(d=1024, k=8)  
kmeans.train(x_train)  
# Training Time: 7.8s
```

The KMeans algorithm has a major limitation.

To recall, KMeans is trained as follows:

- Initialize centroids
- Find the nearest centroid for each point
- Reassign centroids
- Repeat until convergence

But the second step:

- involves a brute-force and exhaustive search
- finds the distance of every point from every centroid

As a result, this implementation:

- isn't optimized



- takes plenty of time to train and predict

This is especially challenging with large datasets.

To speed up KMeans, use Faiss by Facebook AI Research.

Faiss:

- provides a much faster nearest-neighbor search.
- finds an approximate best solution
- uses "Inverted Index", which is an optimized data structure to store and index the data point

This makes performing clustering extremely efficient.

As shown above, Faiss is roughly 20x as fast as running the ideal KMeans algorithm from Sklearn.

Get started with Faiss: [GitHub](#).

👉 Over to you: What are some other limitations of the KMeans algorithm?



# 🚀 Jupyter Notebook + Spreadsheet + AI — All in One Place With Mito

The screenshot shows a Jupyter Notebook cell with the following code:

```
In [1]: import mitosheet  
mitosheet.sheet()
```

Below the code is a spreadsheet interface with the following data:

	Name	Company	City	Salary	Status	Rating
0	Johnny Maynard	White, McClain and C	New Cindychester	8,804.92	1	3.30
1	Michael Williams	Scott Inc	Ricardomouth	11,117.19	1	4.60
2	Laura Flynn	Andrade LLC	North Melissafurt	3,698.55	0	0.30
3	Stefanie Archer	James and Sons	Ricardomouth	8,192.68	1	2.80
4	Sierra Garcia	Matthews Im	Whitakerbury	10,710.60	1	4.50
5	Donna Miller	Andrade LLC	West Jamesview	3,715.78	0	1.10
6	Linda Rodriguez	Marshall-Holloway	Whitakerbury	12,505.38	1	4.70
7	Jonathan Gibson	Scott Inc	Whiteside	7,953.93	0	3.40
8	Karl Henry	Campos, Reynolds ar	Whiteside	10,558.20	1	3.70
9	Katherine Sims	Baker, Allen and Edw	Kristaburgh	9,980.62	1	3.00
10	Lisa Chambers	Nelson-Li	Kristaburgh	9,797.61	0	3.40
11	William Ingram	Baker, Allen and Edw	North Melissafurt	4,574.28	1	0.30
12	Jessica Thomas	Thomas-Spencer	New Cindychester	8,093.12	1	2.90
13	Lisa Pugh	Baker, Allen and Edw	New Cindychester	10,631.22	1	4.80

The interface includes a toolbar with various data manipulation tools like Undo, Redo, Clear, Import, Export, Add Col, Del Col, Dtype, Less, More, Number, Pivot, Graph, and AI. A red circle highlights the "AI" button, which is also labeled "Mito AI" with an arrow pointing to it.

Personally, I am a big fan of no-code data analysis tools. They are extremely useful in eliminating repetitive code across projects—thereby boosting productivity.

Yet, most no-code tools are often limited in terms of the functionality they support. Thus, flexibility is usually a big challenge while using them.

**Mito** is an incredible open-source tool that allows you to analyze your data within a spreadsheet interface in Jupyter without writing any code.

What's more, Mito recently supercharged its spreadsheet interface with AI. As a result, you can now analyze data in a notebook with text prompts.

One of the coolest things about using Mito is that each edit in the spreadsheet automatically generates an equivalent Python code. This makes it convenient to reproduce the analysis later.



## Automatic code generation

```
from mitosheet.public.v3 import *; register_analysis("id-utbdzhmhvd");
import pandas as pd

# Imported employee_dataset.csv
employee_dataset = pd.read_csv(r'employee_dataset.csv')

# group on city and find avg salary and rating
df2 = employee_dataset.groupby('City').agg({'Salary': 'mean', 'Rating': 'mean'})

# top 5 employees with highest salary
top_employees = employee_dataset.nlargest(5, 'Salary')
```

You can install Mito using pip as follows:

```
python -m pip install mitosheet
```

Next, to activate it in Jupyter, run the following two commands:

```
python -m jupyter nbextension install --py --user
mitosheet
```

```
python -m jupyter nbextension enable --py --user mitosheet
```

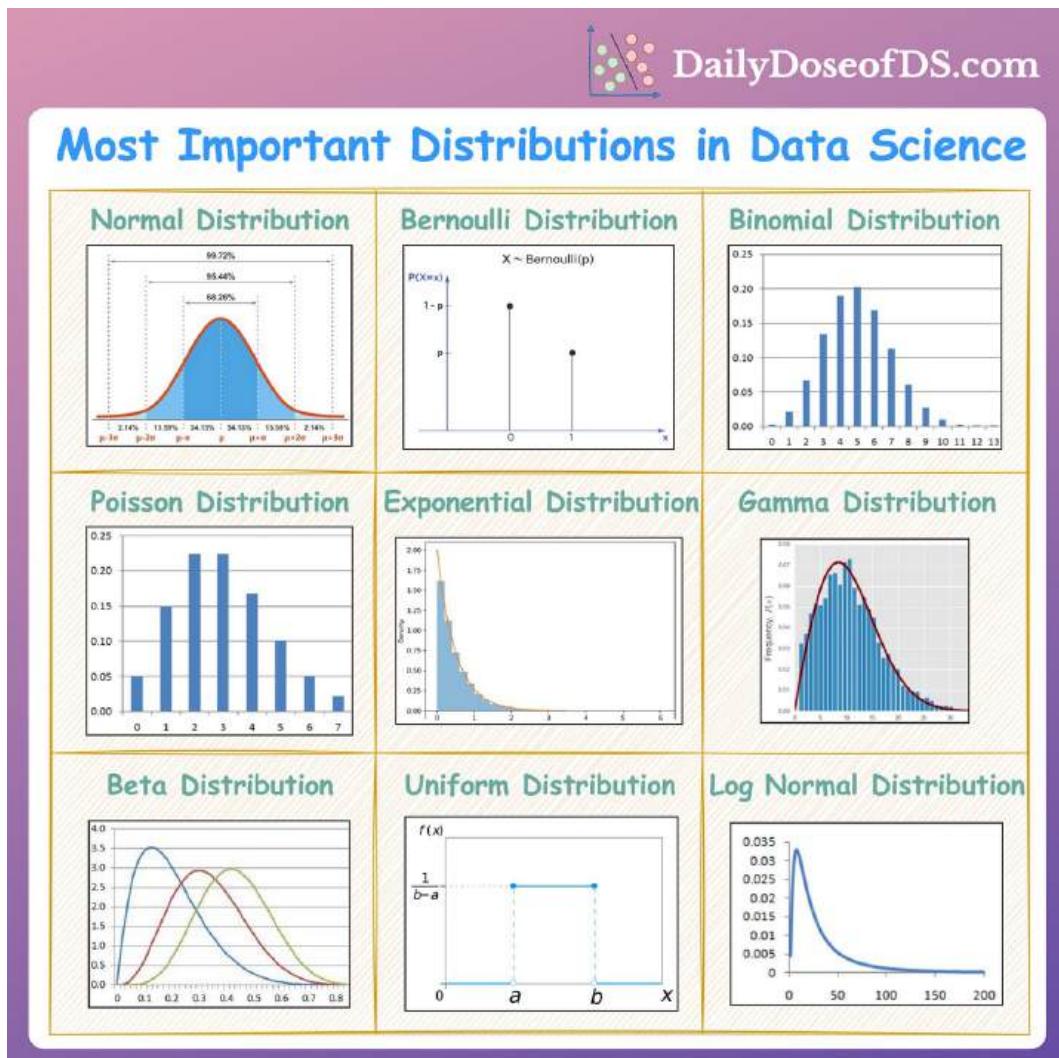
I'm always curious to read your comments. What do you think about this cool feature addition to Mito? Let me know :)

👉 **Read more about the Mito AI's release: <https://bit.ly/mito-ph>.**

👉 **Get started with Mito: <https://bit.ly/mito-qs>.**



# Nine Most Important Distributions in Data Science



Analyzing and modeling data sits at the core of data science.

A fundamental aspect of this process is understanding the underlying distributions that govern the data.

Distributions offer a concise way to:

model and analyze data,

understand the underlying characteristics of data

make informed decisions and draw insights and much more.

Thus, it is crucial to be aware of some of the most important distributions in data science.

## Normal Distribution



$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}$$

- The most widely used distribution in data science.
- It is a continuous probability distribution characterized by a symmetric bell-shaped curve.
- It is parameterized by two parameters—mean and standard deviation.
- Example: Height of individuals.

## Bernoulli Distribution

$$P(X=x) = \begin{cases} p & \text{for } x=1 \\ 1-p & \text{for } x=0 \end{cases}$$

- A discrete probability distribution that models the outcome of a binary event.
- It is parameterized by one parameter—the probability of success.
- Example: Modeling the outcome of a single coin flip.



## Binomial Distribution

$$P(X = x) = \frac{n!}{x!(n-x)!} p^x (1-p)^{(n-x)}$$

- It is Bernoulli distribution repeated multiple times.
- A discrete probability distribution that represents the number of successes in a fixed number of independent Bernoulli trials.
- It is parameterized by two parameters—the number of trials and the probability of success.

## Poisson Distribution

$$P(X) = \frac{\lambda^x e^{-\lambda}}{X!}$$

- A discrete probability distribution that models the number of events occurring in a fixed interval of time or space.
- It is parameterized by one parameter—lambda, the rate of occurrence.
- Example: Analyzing the number of goals a team will score during a specific time period.



## Exponential Distribution

$$f(x, \lambda) = \begin{cases} \lambda \cdot e^{-(\lambda x)} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

- A continuous probability distribution that models the time between events occurring in a Poisson process.
- It is parameterized by one parameter—lambda, the average rate of events.
- Example: Analyzing the time between goals scored by a team.

## Gamma Distribution

$$f(x; \alpha, \beta) = \begin{cases} \frac{1}{\beta^\alpha \Gamma(\alpha)} x^{\alpha-1} e^{-x/\beta} & x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

- It is a variation of the exponential distribution.
- A continuous probability distribution that models the waiting time for a specified number of events in a Poisson process.
- It is parameterized by two parameters—alpha (shape) and beta (rate).
- Example: Analysing the time it would take for a team to score, say, three goals.

## Beta Distribution

<b>PDF</b>	$\frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha, \beta)}$
	where $B(\alpha, \beta) = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha + \beta)}$ and $\Gamma$ is the Gamma function.



- It is used to model probabilities, thus, it is bounded between  $[0,1]$ .
- Very similar to the binomial distribution. The difference is that binomial distribution models the number of successes, while beta distribution models the probability of success ( $p$ ).
- In other words, the probability is a parameter in the binomial distribution. But in the Beta distribution, the probability is a random variable.

## Uniform Distribution

$$f(x) = \begin{cases} \frac{1}{b - a}, & x \in [a, b] \\ 0, & \text{otherwise} \end{cases}$$

- A continuous probability distribution where all outcomes within a given range are equally likely.
- It is parameterized by two parameters:  $a$  (minimum value) and  $b$  (maximum value).
- Example: Simulating the roll of a fair six-sided die, where each outcome (1, 2, 3, 4, 5, 6) has an equal probability.

## Log-Normal Distribution

$$X \sim N(\mu, \sigma^2) \quad Y = e^X$$
$$Y \sim Lognormal(\mu, \sigma^2)$$

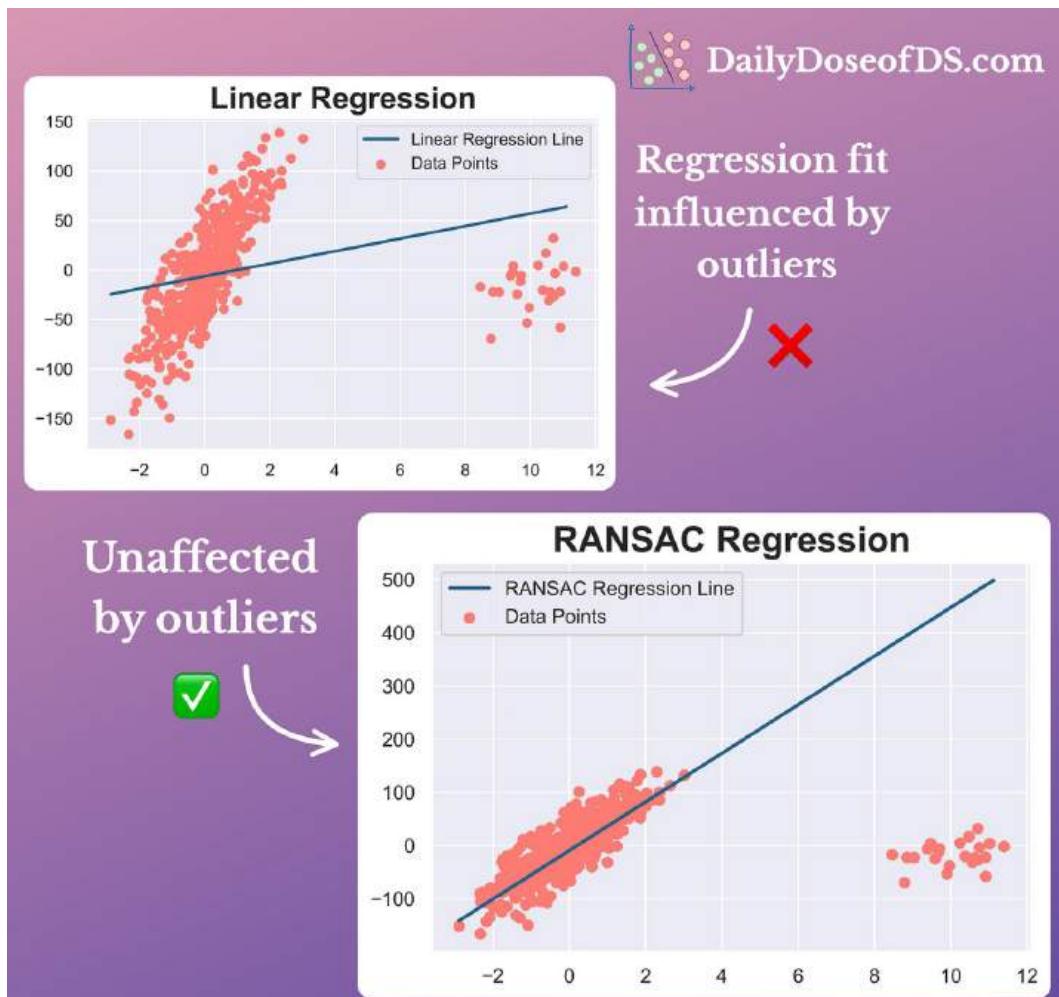
- A continuous probability distribution where the logarithm of the variable follows a normal distribution.



- It is parameterized by two parameters—mean and standard deviation.
- Example: Typically, in stock returns, the natural logarithm follows a normal distribution.
- Over to you: What more distributions will you include here?



# The Limitation of Linear Regression Which is Often Overlooked By Many



Linear Regression is the most widely used ML algorithm.

But it is sensitive to outliers.

In fact, even a few outliers can significantly impact Linear Regression performance.

Instead, try RANSAC Regression. It is

- non-deterministic,
- iterative, and
- robust to outliers.

It works as follows:

- Select a subset of data
- Fit a model



- Calculate residuals

Classify points as outliers/inliers based on thresholds applied to residuals

Repeat (until max iterations or when a condition is met)

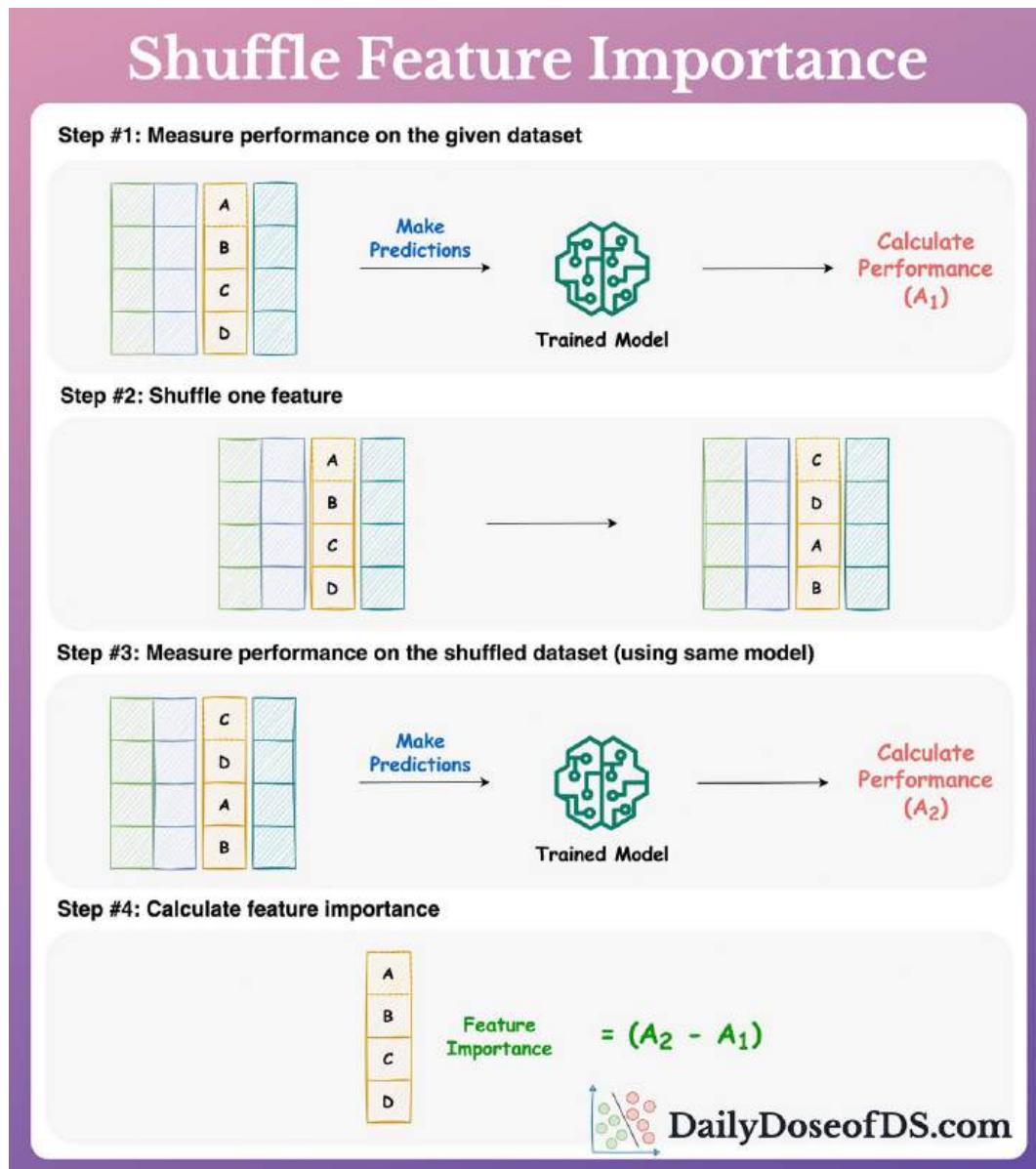
As shown above, while Linear Regression is influenced by outliers, RANSAC Regression isn't.

Nonetheless, it is always recommended to experiment with many robust methods and see which one fits your data best.

👉 Get started with RANSAC: [Sklearn Docs](#).



# A Reliable and Efficient Technique To Measure Feature Importance



Here's a neat technique to quickly and reliably measure feature importance in any ML model.

Permutation feature importance observes how randomly shuffling a feature influences model performance.

Essentially, after training a model, we do the following:

Measure model performance ( $A_1$ ) on the given dataset (test/validation/train).

Shuffle one feature randomly.



Measure performance ( $A_2$ ) again.

Feature importance =  $(A_1 - A_2)$ .

Repeat for all features.

To eliminate any potential effects of randomness during feature shuffling, it is also recommended to shuffle the same feature multiple times.

### **Benefits of permutation feature importance:**

No repetitive model training.

The technique is pretty reliable.

It can be applied to any model, as long as you can evaluate the performance.

It is efficient

Of course, there is one caveat to this approach.

Say two features are highly correlated and one of them is permuted/shuffled. In this case, the model will still have access to the feature through its correlated feature.

This will result in a lower importance value for both features.

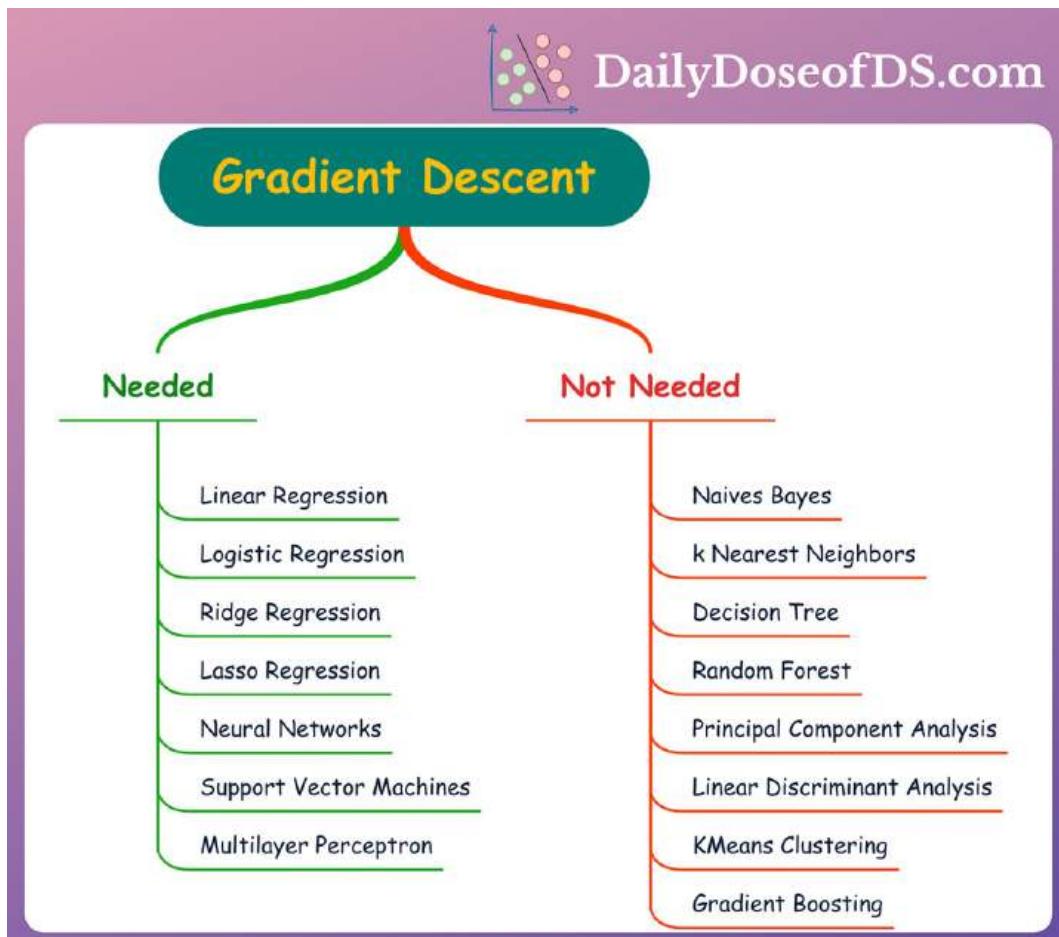
One way to handle this is to cluster features that are highly correlated and only keep one feature from each cluster.

Here's one of my previous guides on making this task easier: [The Limitations Of Heatmap That Are Slowing Down Your Data Analysis.](#)

👉 Over to you: What other reliable feature importance techniques do you use frequently?



# Does Every ML Algorithm Rely on Gradient Descent?



Gradient descent is the most common optimization technique in ML. Essentially, the core idea is to iteratively update the model's parameters by calculating the gradients of the cost function with respect to those parameters.

Why gradient descent is a critical technique, it is important to know that not all algorithms rely on gradient descent.

The visual above depicts this.

## Algorithms that rely on gradient descent:

- Linear Regression
- Logistic Regression
- Ridge Regression
- Lasso Regression



- Neural Networks (ANNs, RNNs, CNNs, LSTMs, etc.)
- Support Vector Machines
- Multilayer Perceptrons

### Algorithms that DON'T rely on gradient descent:

- Naive Bayes
- kNN
- Decision Tree
- Random Forest
- Principal Component Analysis
- Linear Discriminant Analysis
- KMeans Clustering
- Gradient Boosting



# Why Sklearn's Linear Regression Has No Hyperparameters?

DailyDoseofDS.com

## sklearn.linear\_model.LinearRegression

```
class sklearn.linear_model.LinearRegression(*, fit_intercept=True, copy_X=True, n_jobs=None, positive=False)
```

[\[source\]](#)

**No Hyperparameters!**

Parameters:

- fit\_intercept : bool, default=True**  
Whether to calculate the intercept for this model. If set to False, no intercept will be used in calculations (i.e. data is expected to be centered).
- copy\_X : bool, default=True**  
If True, X will be copied; else, it may be overwritten.
- n\_jobs : int, default=None**  
The number of jobs to use for the computation. This will only provide speedup in case of sufficiently large problems, that is if firstly `n_targets > 1` and secondly X is sparse or if `positive` is set to `True`. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See [Glossary](#) for more details.
- positive : bool, default=False**  
When set to `True`, forces the coefficients to be positive. This option is only supported for dense arrays.

New in version 0.24.

All ML models we work with have some hyperparameters, such as:

- Learning rate
- Regularization
- Layer size (for neural network), etc.

But as shown in the image above, why don't we see one in Sklearn's Linear Regression implementation?

To understand the reason, we first need to realize that the Linear Regression algorithm can model data in two different ways:

**Gradient Descent** (which we use with almost all other ML algorithms):

- It is a stochastic algorithm, i.e., involves some randomness.
- It finds an approximate solution using optimization.
- It has hyperparameters.



## Ordinary Least Square (OLS):

- It is a deterministic algorithm. If run multiple times, it will always converge to the same weights.
- It always finds the optimal solution.
- It has no hyperparameters.

Instead of gradient descent, Sklearn's Linear Regression class implements the OLS method.

**sklearn.linear\_model.LinearRegression**

```
class sklearn.linear_model.LinearRegression(*, fit_intercept=True, copy_X=True, n_jobs=None,
positive=False)
```

[\[source\]](#)

Ordinary least squares Linear Regression.

OLS

That is why it has no hyperparameters.

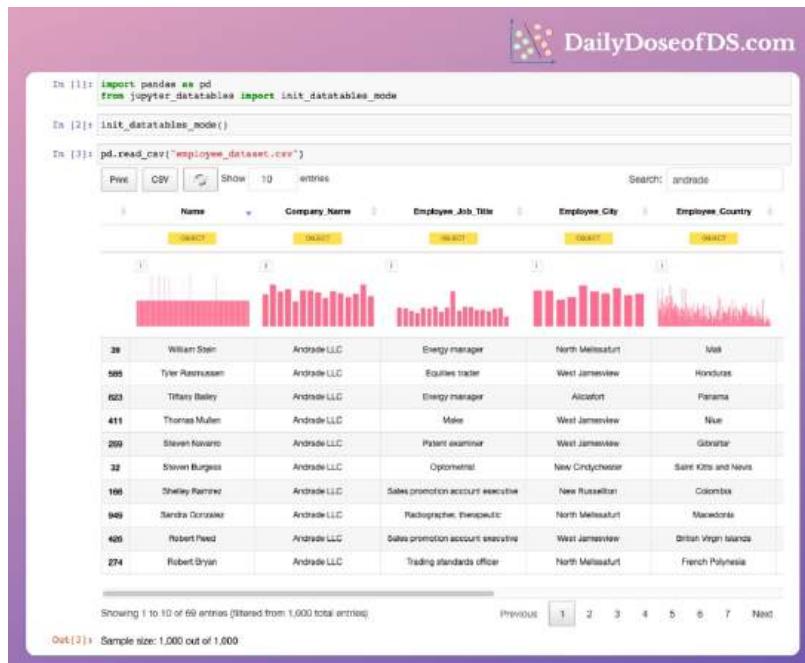
## How does OLS work?

Read the full post here with equations:

<https://www.blog.dailydoseofds.com/p/why-sklearns-linear-regression-has>.



# Enrich The Default Preview of Pandas DataFrame with Jupyter DataTables



After loading any dataframe in Jupyter, we preview it. But it hardly tells anything about the data.

One has to dig deeper by analyzing it, which involves simple yet repetitive code.

Instead, use [Jupyter-DataTables](#).

It supercharges the default preview of a DataFrame with many common operations.

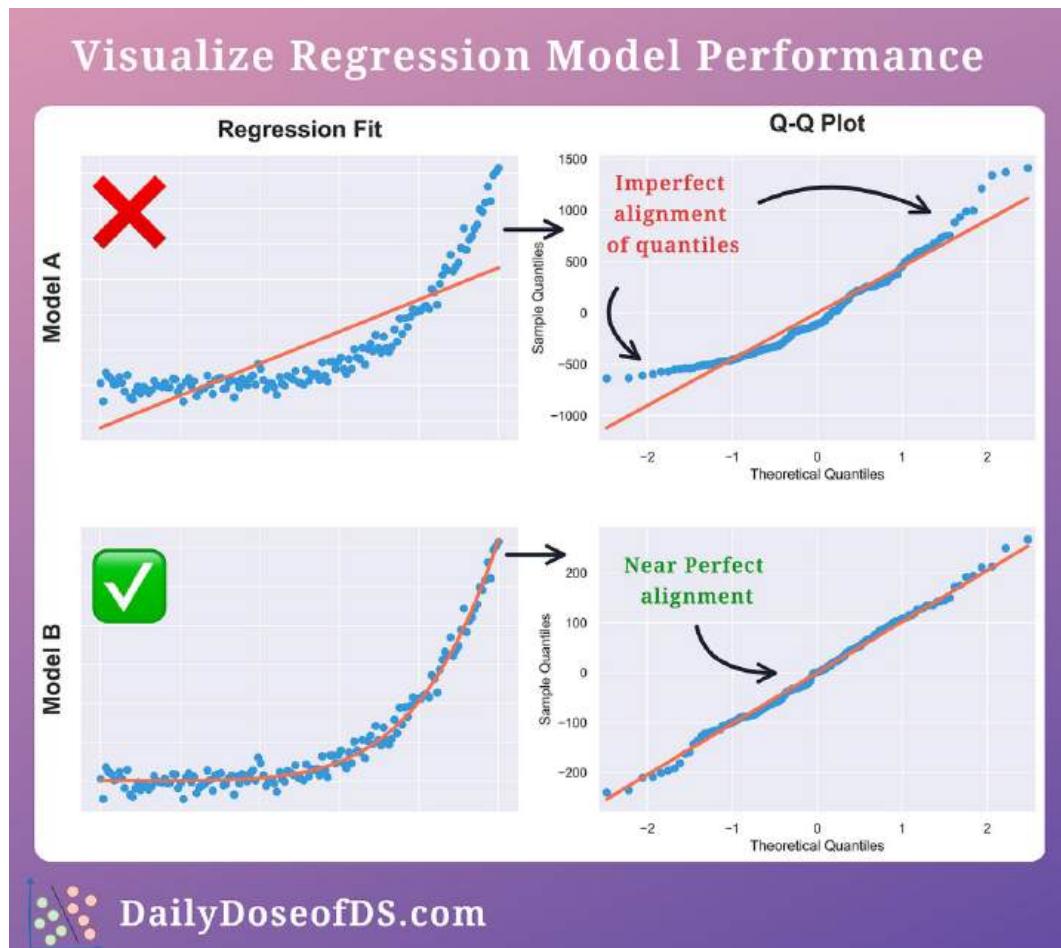
This includes:

- sorting
- filtering
- exporting
- plotting column distribution
- printing data types,
- pagination, and more.

Check it out here: [GitHub](#).



# Visualize The Performance Of Linear Regression With This Simple Plot



Linear regression assumes that the model residuals (=actual-predicted) are normally distributed.

If the model is underperforming, it may be due to a violation of this assumption.

A QQ plot (short for Quantile-Quantile) is a great way to verify this and also determine the model's performance.

As the name suggests, it depicts the quantiles of the observed distribution (residuals in this case) against the quantiles of a reference distribution, typically the standard normal distribution.

A good QQ plot will:

- Show minimal deviations from the reference line, indicating that the residuals are approximately normally distributed.



A bad QQ plot will:

- Exhibit significant deviations, indicating a departure from the normality of residuals.
- Display patterns of skewness with its diverging ends, etc.

Thus, the more aligned the QQ plot looks, the more confident you can be about your model.

This is especially useful when the regression line is difficult to visualize, i.e., in a high-dimensional dataset.

So remember...

After running a linear model, always check the distribution of the residuals.

This will help you:

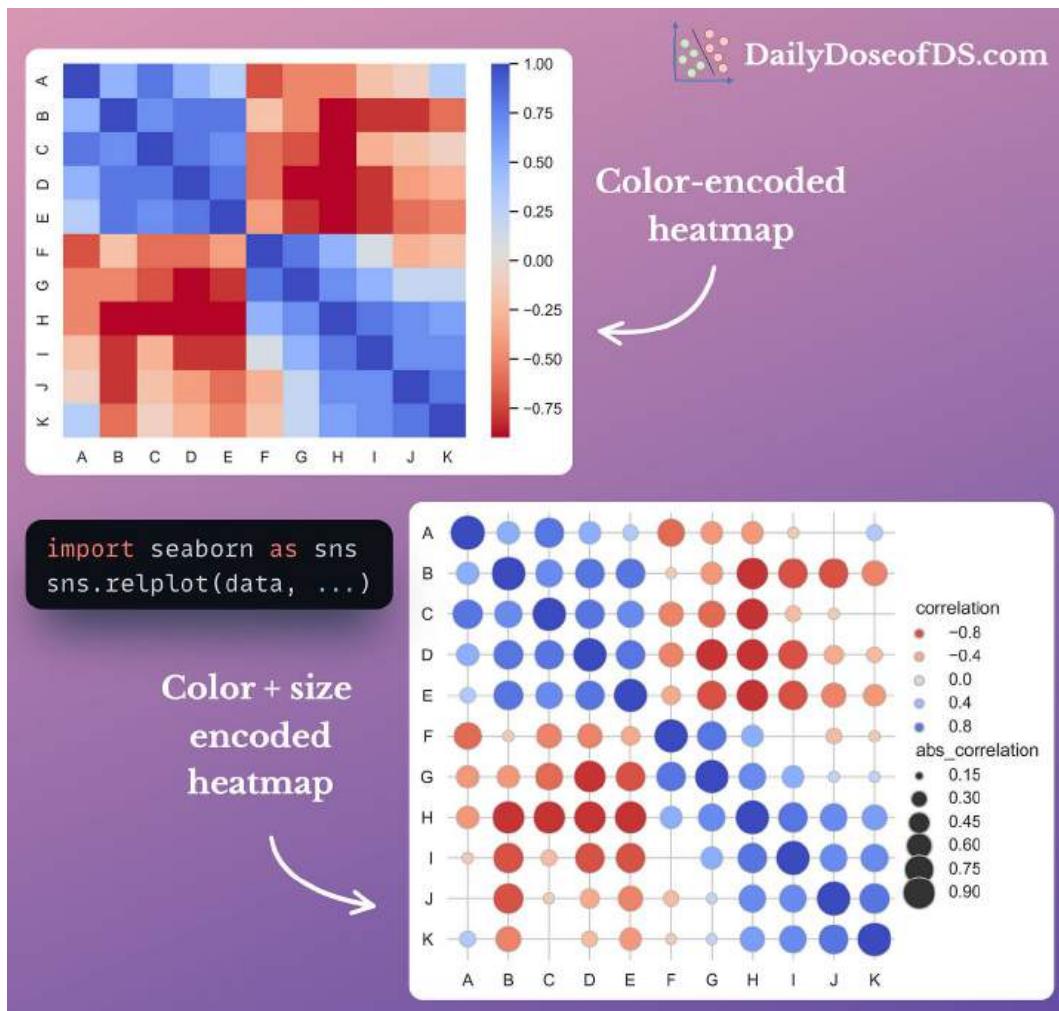
- Validate the model's assumptions
- Determine how good your model is
- Find ways to improve it (if needed)

👉 Over to you: What are some other ways/plots to determine the linear model's performance?

I covered another way in one of my previous posts: [Visualize The Performance Of Any Linear Regression Model With This Simple Plot.](#)



# Enrich Your Heatmaps With This Simple Trick



Heatmaps often make data analysis much easier. Yet, they can be further enriched with a simple modification.

A traditional heatmap represents the values using a color scale. Yet, mapping the cell color to numbers is still challenging.

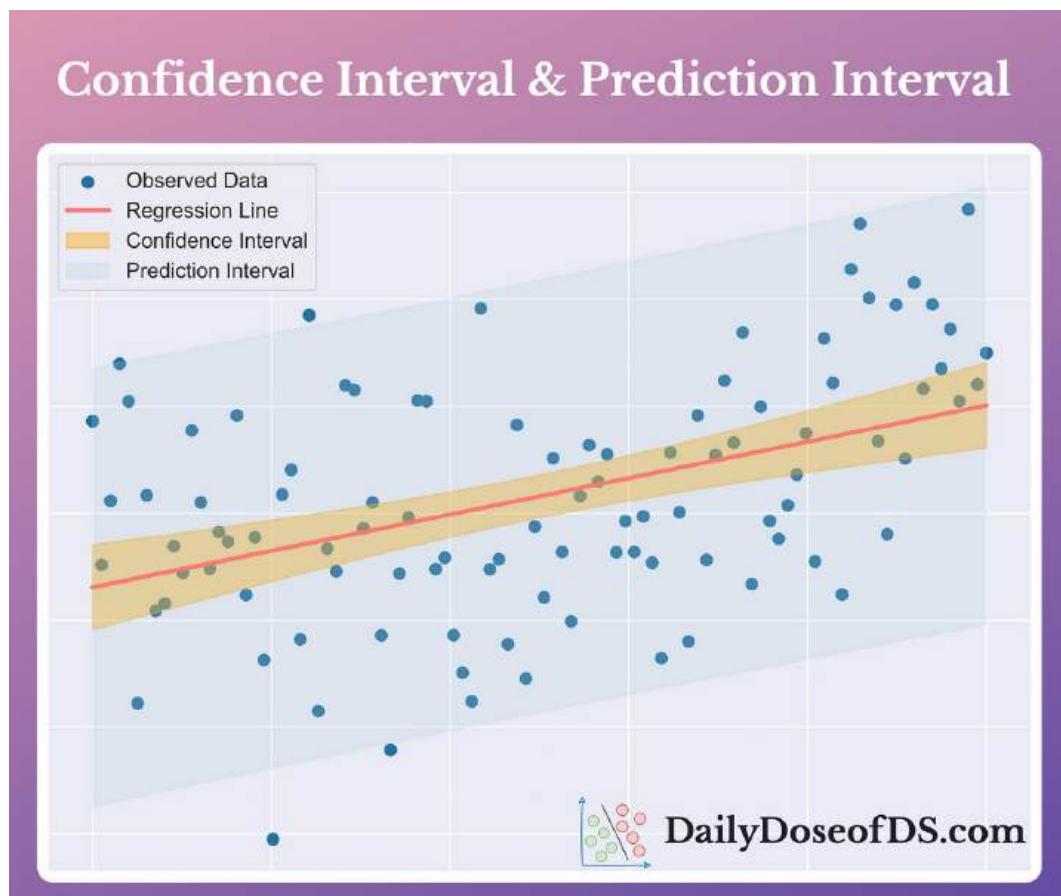
Embedding a size component can be extremely helpful in such cases. In essence, the bigger the size, the higher the absolute value.

This is especially useful to make heatmaps cleaner, as many values nearer to zero will immediately shrink.

Find more details here: [Seaborn Docs](#).



# Confidence Interval and Prediction Interval Are Not The Same



Contrary to common belief, linear regression NEVER predicts an actual value.

Instead, it models the relationship between the input and an average related to the outcome.

Thus, there's always some uncertainty involved, and it is important to communicate it.

Confidence interval and prediction interval help us capture this uncertainty.

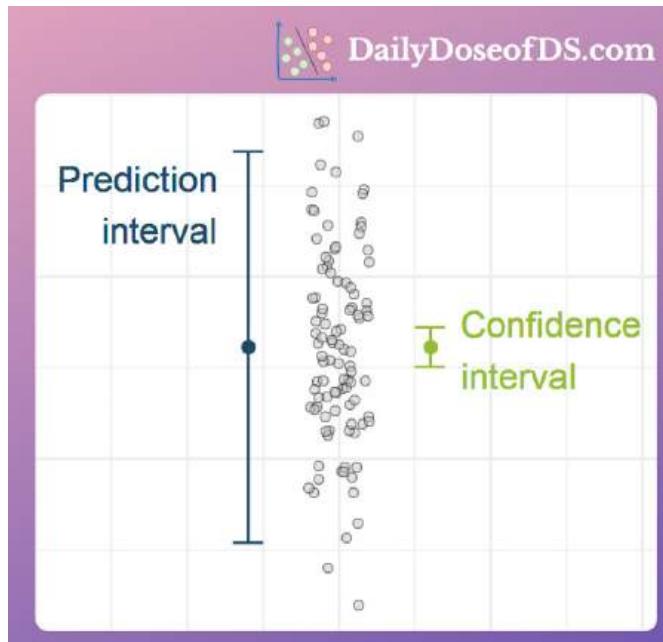
### Confidence interval:

- tells the range of mean outcome at a given input.
- answers the question: "If we know the input, what is the uncertainty around the average value of the outcome."

Thus, a 95% confidence interval says:



- given an input, we are 95% confident that the actual mean will lie in that region.



### Prediction interval, however:

- tells the range of possible values the outcome variable may take.
- answers the question: "If we know the input, what is the actual range of the outcome variable that we may observe."

For instance, a 95% prediction interval tells us that:

- given an input, 95% of observed values will lie in that region.

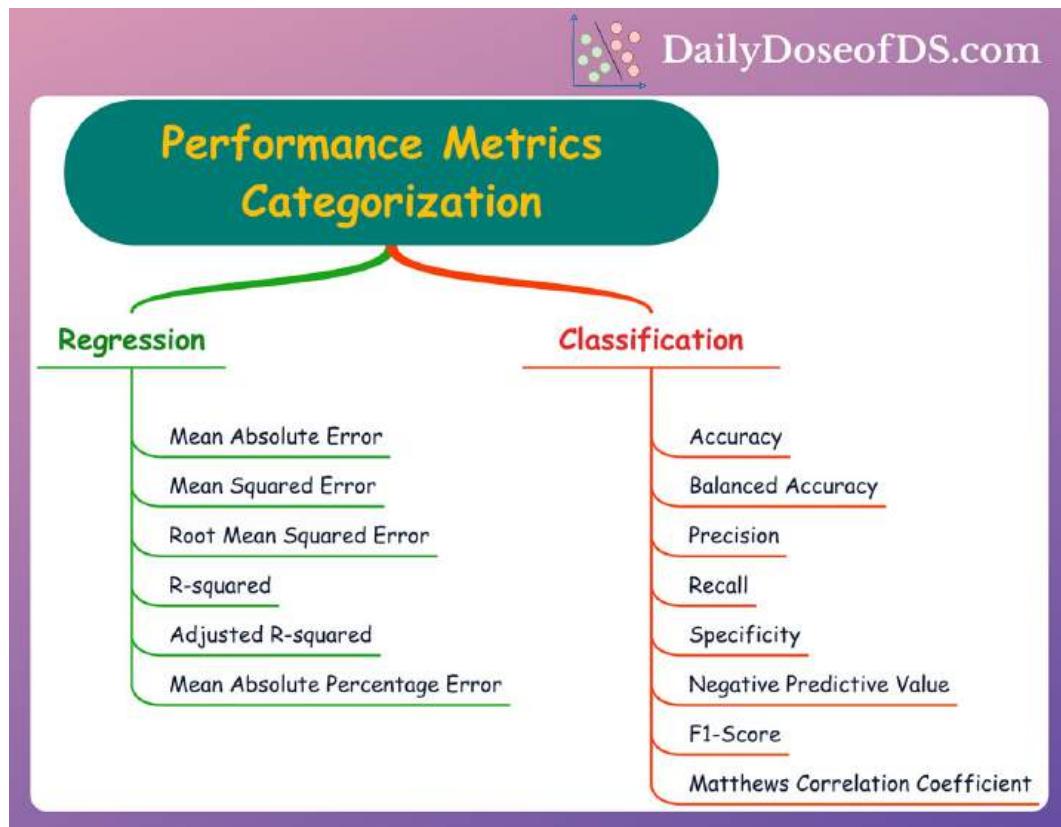
So remember...

- Confidence interval and prediction interval are NOT the same.
- They depict different uncertainties of the outcome variable.
- Confidence interval captures the range of the mean outcome around an input.
- Prediction interval captures the range of the actual values of the outcome around an input.
- Prediction interval is typically wider than confidence interval.

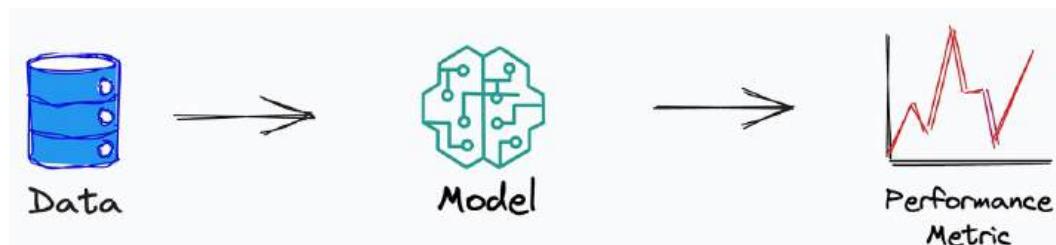
👉 Over to you: What does a 100% confidence interval and prediction interval will look like?



# The Ultimate Categorization of Performance Metrics in ML



**Performance metrics** are used to assess the performance of a model on a given dataset.



They provide quantitative ways to test the effectiveness of a model. They also help in identifying the areas for improvement and optimization.



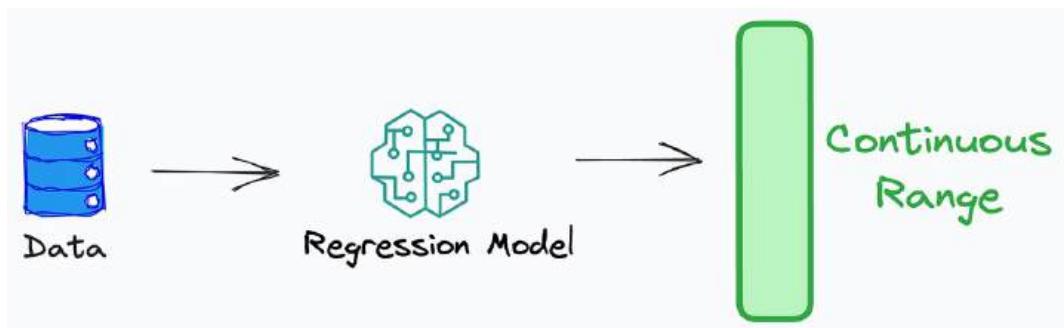
## Why Performance metrics?

Typically, it is difficult to interpret ML models (especially deep learning models). It is difficult to understand the specific patterns identified by the model from the given data.

Performance metrics allow us to determine their performance by providing unseen samples by evaluating the predictions.

This makes them a must-know in ML.

## Performance Metrics for Regression



### Mean Absolute Error (MAE):

Measures the average absolute difference between the predicted and actual value.

Provides a straightforward assessment of prediction accuracy.

### Mean Squared Error (MSE):

Measures the average squared difference between the predicted and actual values.

Larger errors inflate the overall metric.

### Root Mean Squared Error:

It is the square root of MSE.

### R-squared:

Represents the proportion of the variance in the target variable explained by the regression model.

### Adjusted R-squared:

Similar to R-squared.

But accounts for the number of predictors (features) in the model.



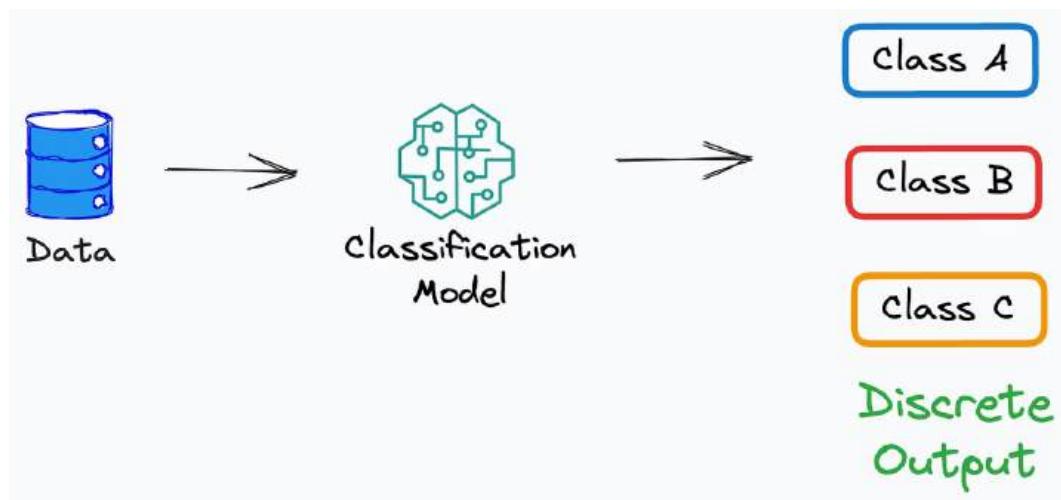
Penalizes model complexity.

### Mean Absolute Percentage Error (MAPE):

Measures the average percentage difference between the predicted and actual values.

Typically used when the scale of the target variable is significant.

## Performance Metrics for Classification



### Accuracy:

Measures the proportion of correct predictions, irrespective of the class.

Provides an overall assessment of classification performance.

### Precision:

Measures the proportion of positive predictions that were correct.

It is also called the accuracy of the model only on the positive predictions.

### Recall (Sensitivity):

Measures the proportion of correctly classified positive samples in the dataset.

It is also called the accuracy of the model on positive instances in the dataset.

---

Precision → Accuracy on positive predictions.



Recall → Accuracy on positive instances in the dataset.

Read by Medium blog to understand more: [Precision-Recall Blog](#).

---

## Specificity:

Opposite of Recall.

Measures the proportion of correctly classified negative samples in the dataset.

It is also called the accuracy of the model on negative instances in the dataset.

## Negative Predictive Value:

Opposite of Precision.

Measures the proportion of negative predictions that were correct.

It is also called the accuracy of the model only on the negative predictions.

## Balanced Accuracy:

Computes the average of Recall (accuracy on positive predictions) and specificity (accuracy on negative predictions)

A better and less-misleading measure than Accuracy in the case of an imbalanced dataset.

## F1-score:

The harmonic mean of precision and recall.

Provides a balanced measure between the two.

## Matthews Correlation Coefficient (MCC):

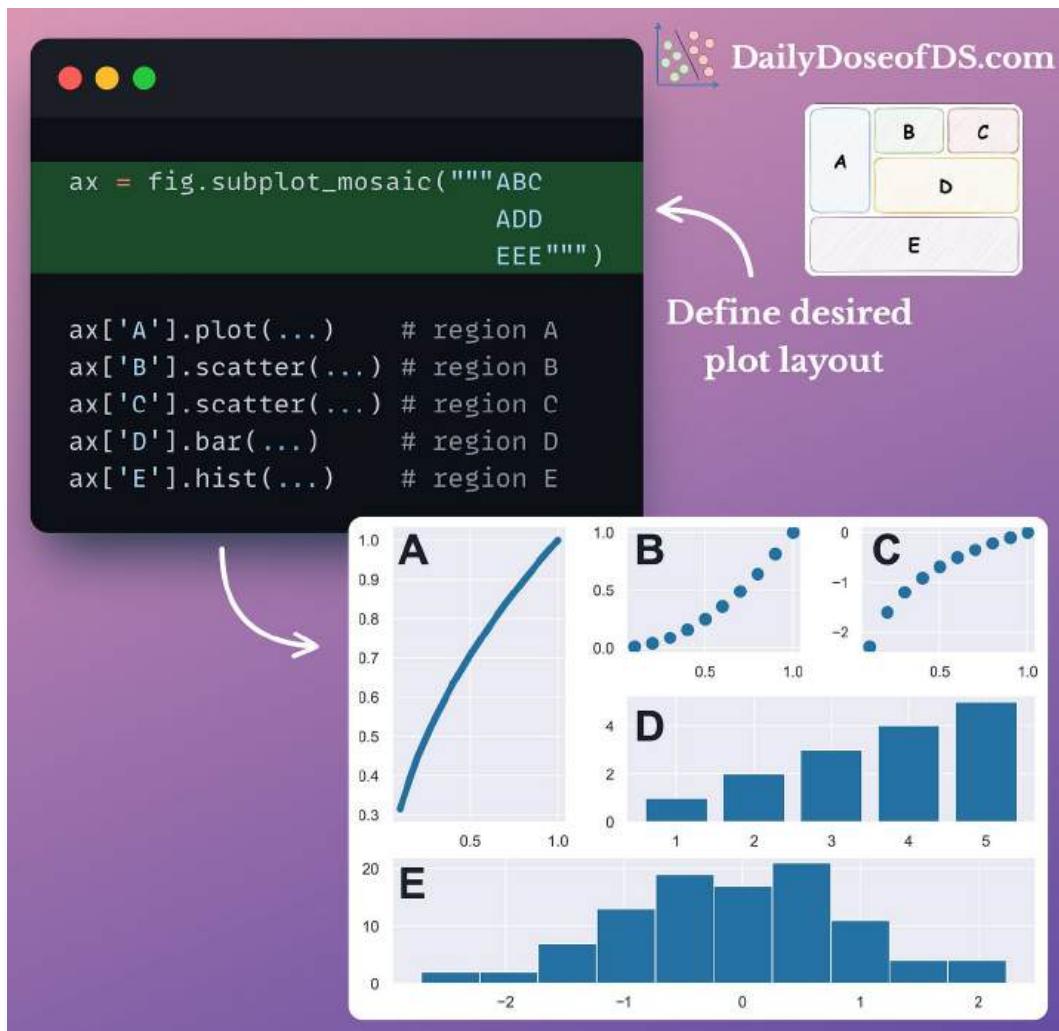
Takes into account true positive (TP), true negative (TN), false positive (FP), and false negative (FN) predictions to measure the performance of the binary classifier.

Provides a balanced performance measure unaffected by class imbalance.

If you struggle to understand TP, TN, FP and FN, read my previous post: [Use This Simple Technique To Never Struggle With TP, TN, FP and FN Again](#)



# The Coolest Matplotlib Hack to Create Subplots Intuitively



This has to be the coolest thing I have ever learned about Matplotlib.

We mostly use `plt.subplots()` method to create subplots using Matplotlib.

But this, at times, gets pretty tedious and cumbersome. For instance: it offers limited flexibility to create a custom layout.

it is prone to indexing errors, and more.

Instead, use the `plt.subplot_mosaic()` method.

Here, you can create a plot of any desired layout by defining the plot structure as a string.

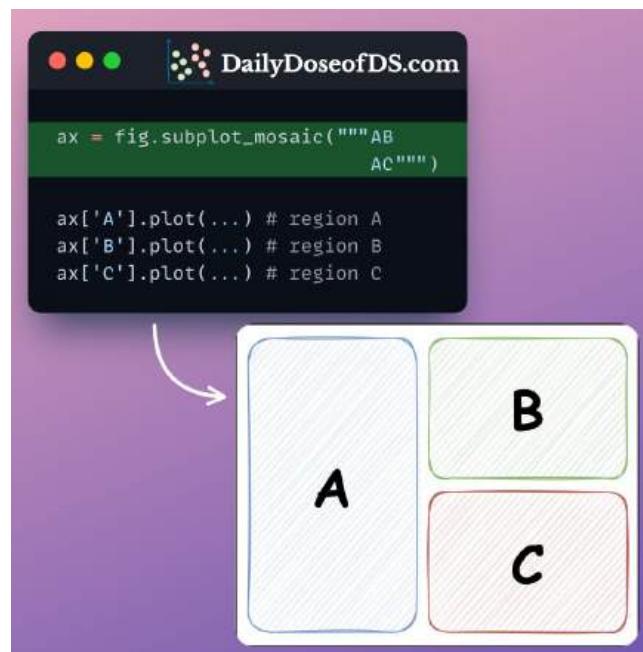
For instance, the string layout:



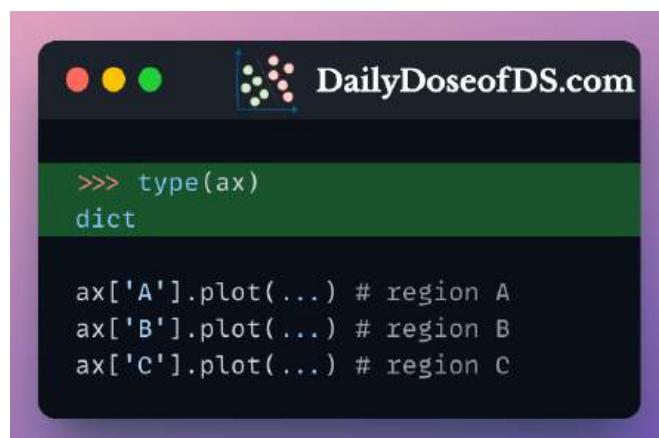
- AB
- AC

will create three subplots, wherein:

- subplot "A" will span the full first column
- subplot "B" will span the top half of the second column
- subplot "C" will span the bottom half of the second column



Next, create a subplot in a specific region by indexing the axes dictionary with its subplot key ("A", "B", or "C").

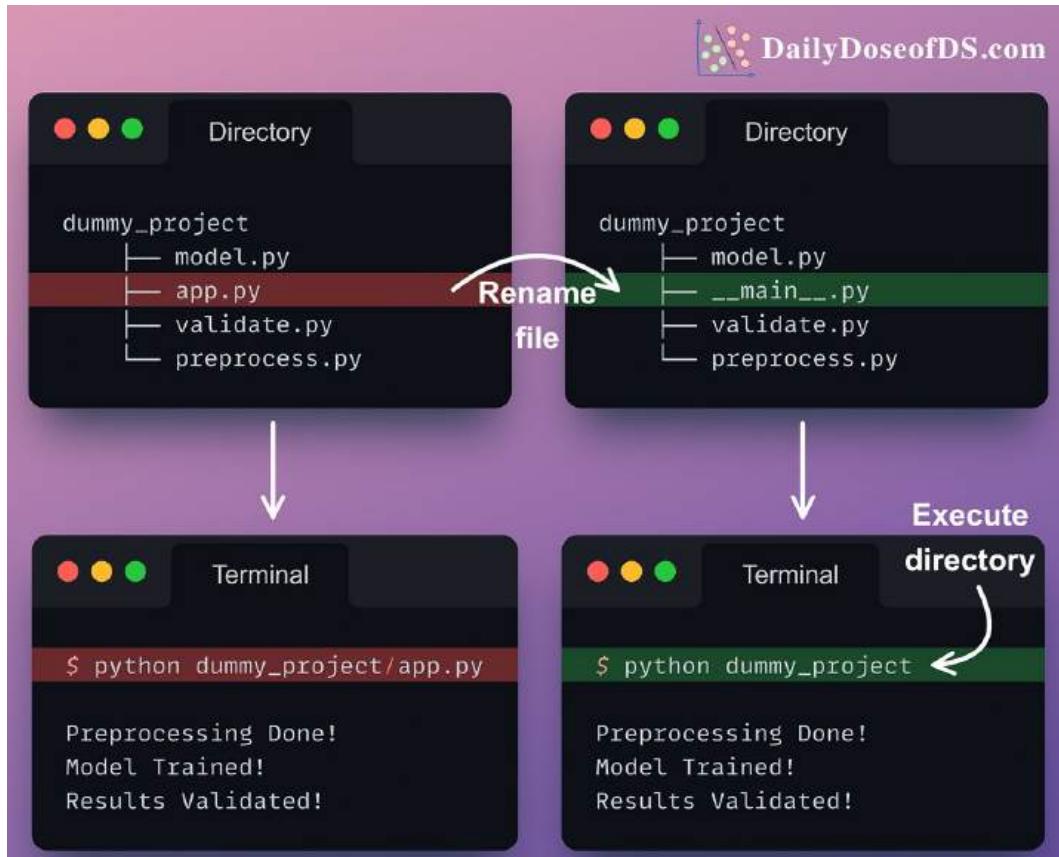


Isn't that super convenient and cool?

Read more: [Matplotlib Docs](https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.subplot_mosaic.html).



# Execute Python Project Directory as a Script



We mostly run a Python pipeline by invoking a script (**.py** file).

But did you know you can also execute the Python project directory as a script?

To do this, rename the base file of your project to **\_\_main\_\_.py**.

As a result, you can execute the whole pipeline by running the parent directory itself.

This is concise and elegant.

It also makes it easier for other users to use your project, as they are not required to dig into the directory and locate the base file.



# The Most Overlooked Problem With One-Hot Encoding

## The correct way to one-hot encode data

DailyDoseofDS.com

**Categorical data**

size	
0	small
1	medium
2	large

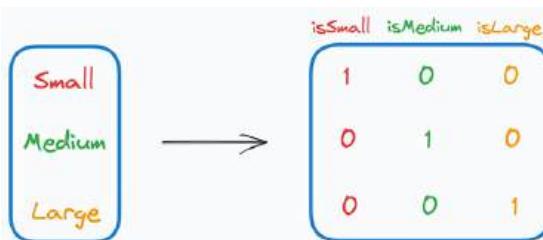
**Don't keep all features**

	size_large	size_medium	size_small
0	0	0	1
1	0	1	0
2	1	0	0

**Instead, drop one feature**

	size_medium	size_small
0	0	1
1	1	0
2	0	0

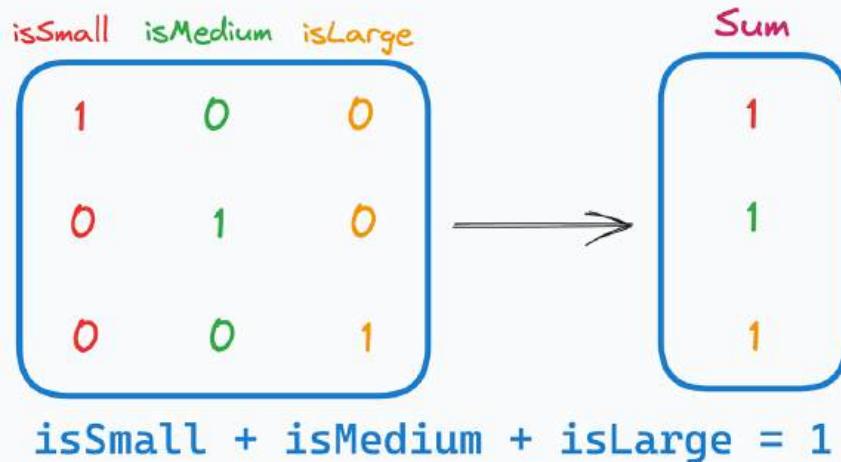
With one-hot encoding, we introduce a big problem in the data.



When we one-hot encode categorical data, we unknowingly introduce perfect multicollinearity.

Multicollinearity arises when two or more features can predict another feature.

As the sum of one-hot encoded features is always 1, it leads to perfect multicollinearity.



This is often called the Dummy Variable Trap.

It is bad because:

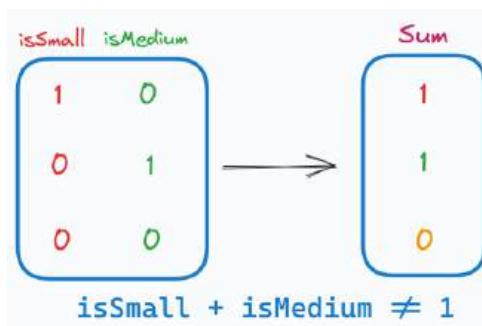
- The model has redundant features
- Regressions coefficients aren't reliable in the presence of multicollinearity, etc.

So how to resolve this?

The solution is simple.

Drop any arbitrary feature from the one-hot encoded features.

This instantly mitigates multicollinearity and breaks the linear relationship which existed before.



So remember...

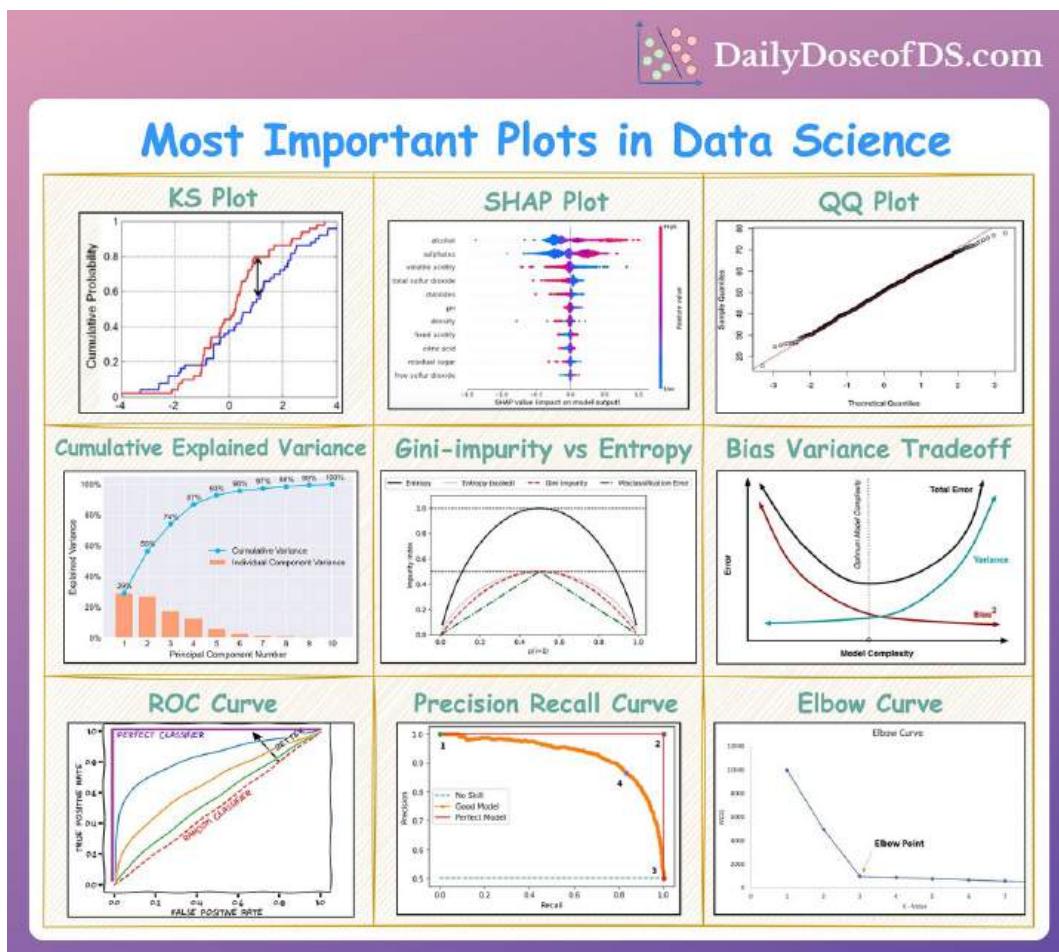
Whenever we one-hot encode categorical data, it introduces multicollinearity.

To avoid this, drop one column and proceed ahead.

👉 Over to you: What are some other problems with one-hot encoding?



# 9 Most Important Plots in Data Science



Exploring and analyzing data is a fundamental aspect of data science.

Here, visualizations play a crucial role in understanding complex patterns and relationships.

They offer a concise way to:

- understand the intricacies of statistical models,
- validate model assumptions,
- evaluate model performance, and much more.

The visual above depicts 9 of the most important and must-know plots in data science.

- **KS Plot:** It compares the cumulative distribution functions (CDFs) of a dataset to a theoretical distribution or between two datasets to assess the distributional differences.

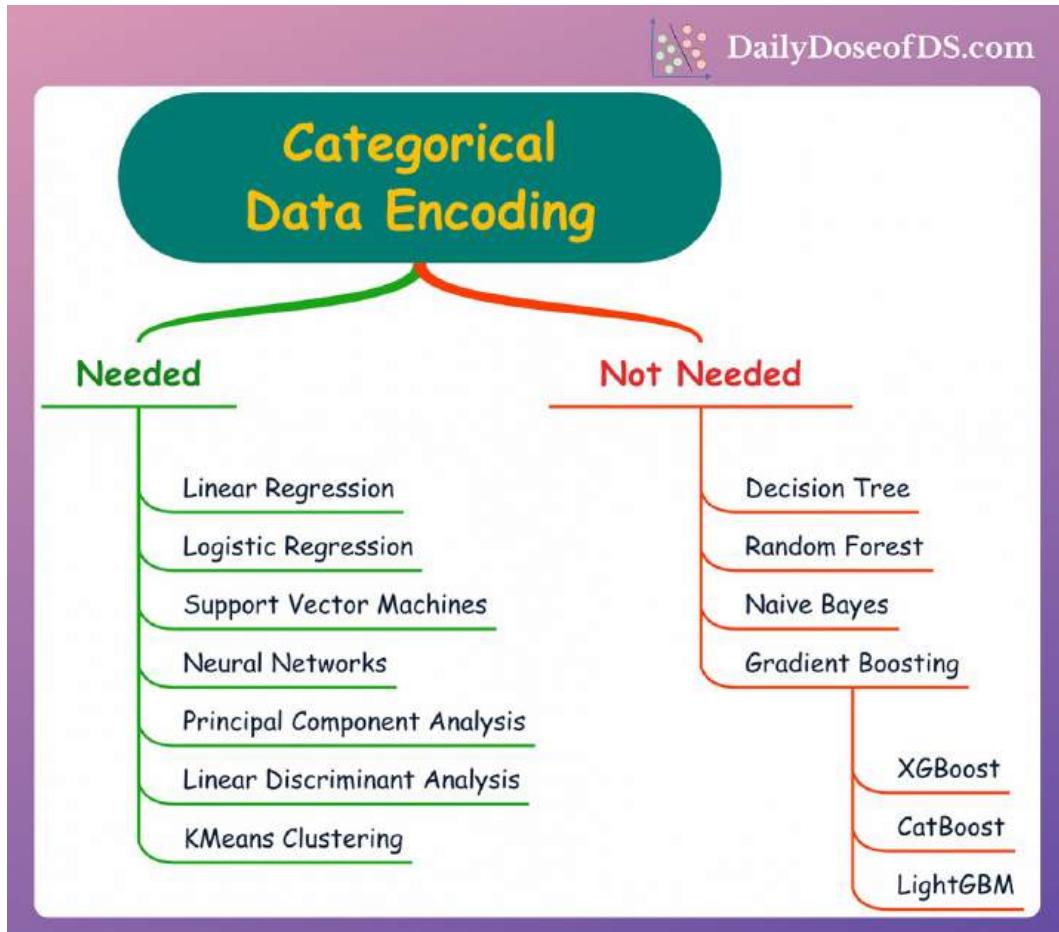


- **SHAP Plot:** It provides a summary of feature importance to a model's predictions, by considering interactions/dependencies between them.
- **QQ Plot:** It is used to assess the distributional similarity between observed data and theoretical distribution.
  - Here, we plot the quantiles of the two distributions against each other.
  - Deviations from the straight line indicate a departure from the assumed distribution.
- **Cumulative Explained Variance Plot:** I covered this in a detailed post before: [How Many Dimensions Should You Reduce Your Data To When Using PCA?](#)
- **Gini-Impurity vs. Entropy:** They are used to measure the impurity or disorder of a node or split in a decision tree.
  - The plot compares Gini impurity and Entropy across different splits. This provides insights into the tradeoff between these measures.
- **Bias-Variance Tradeoff:** It is used to find the right balance between the bias and the variance of a model.
- **ROC Curve:** It depicts the trade-off between the true positive rate (TPR) and the false positive rate (FPR) across different classification thresholds.
- **Precision-Recall Curve:** It depicts the trade-off between Precision and Recall across different classification thresholds.
- **Elbow Curve:** The plot helps identify the optimal number of clusters for k-means algorithm.

Over to you: What more plots will you include here?



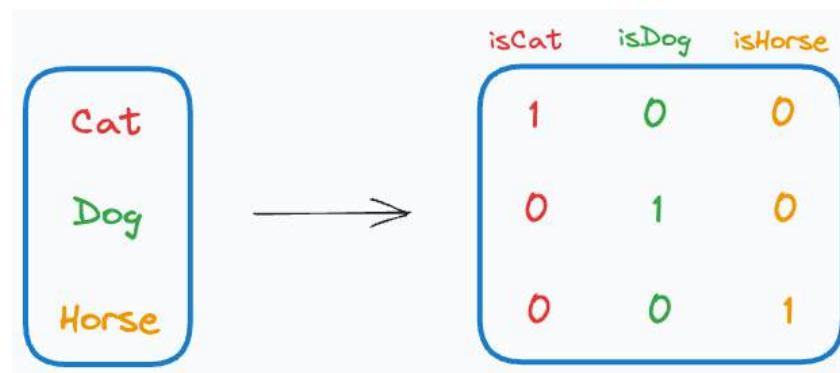
# Is Categorical Feature Encoding Always Necessary Before Training ML Models?



When data contains categorical features, they may need special attention at times. This is because many algorithms require numerical data to work with.

Thus, when dealing with such datasets, it becomes crucial to handle these features appropriately to ensure accurate and meaningful analysis.

For instance, one common approach is to use one-hot encoding, as shown below:

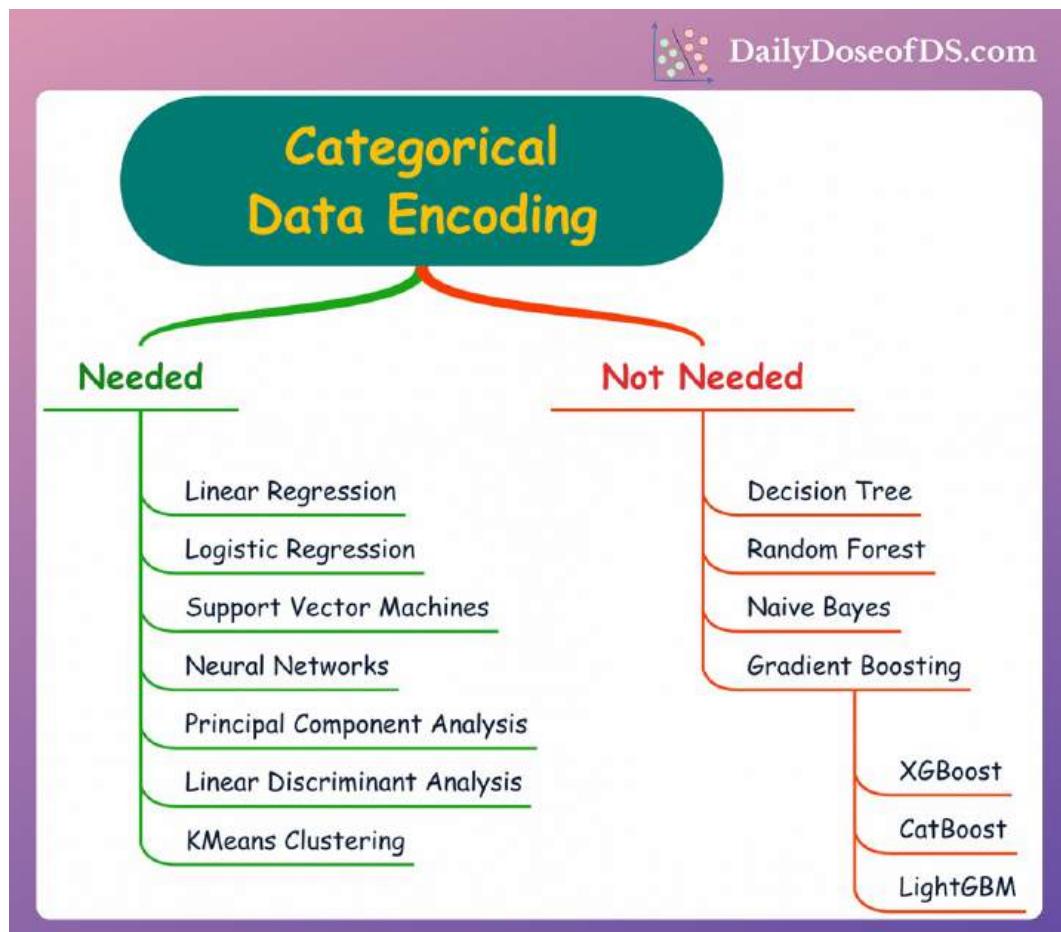


Encoding categorical data allows algorithms to process them effectively.

But is it always necessary?

While encoding categorical data is often crucial, knowing when to do it is also equally important.

The following visual depicts which algorithms need categorical data encoding and which don't.

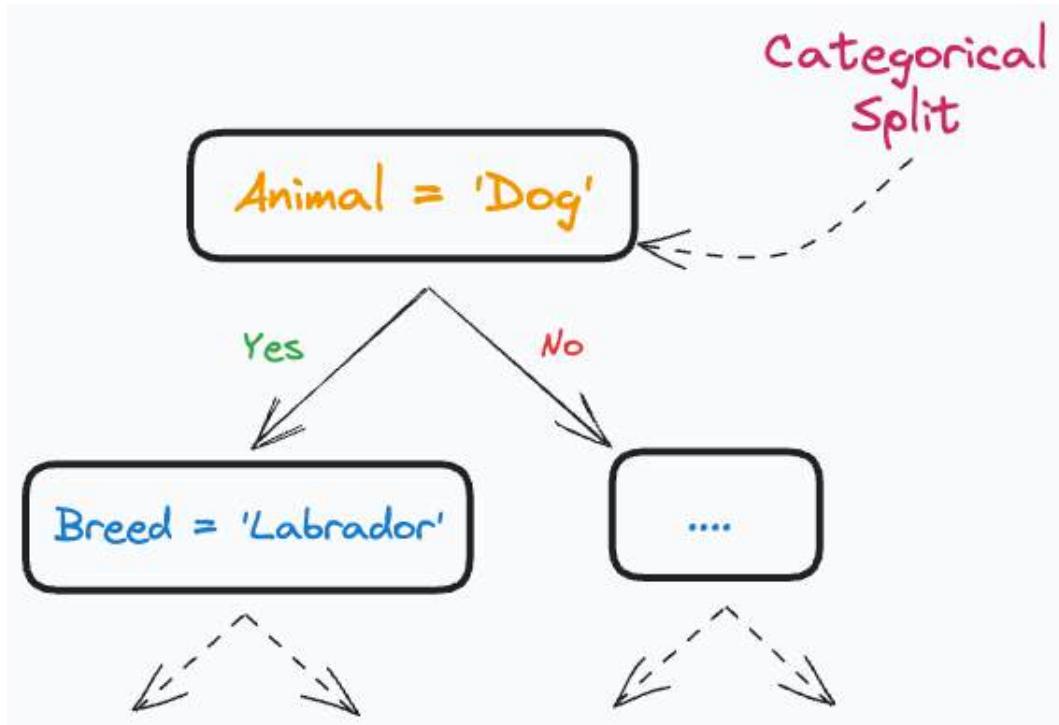




Categorization of algorithms based on categorical data encoding requirement

As shown above, many ML algorithms typically work well even without categorical data encoding. These include decision trees, random forests, naive bayes, gradient boosting, and more.

Consider a decision tree, for instance. It can split the data based on exact categorical feature values. This makes categorical feature encoding an unnecessary step.



Thus, it's important to understand the nature of your data and the algorithm you intend to use.

You may never need to encode categorical data if the algorithm is insensitive to it.

👉 Over to you: Where would you place k-nearest neighbors in this chart? Let me know :)



# Scikit-LLM: Integrate Sklearn API with Large Language Models

The screenshot shows the Scikit-LLM GitHub repository. It features two main code snippets. The top snippet, titled 'Set Config', shows how to import the library and set OpenAI API keys:

```
# 1) Imports
from skllm.config import SKLLMConfig
from skllm import ZeroShotGPTClassifier

# 2) Set OpenAI API Key and Organisation
SKLLMConfig.set_openai_key("<YOUR_KEY>")
SKLLMConfig.set_openai_org("<YOUR_ORG>")
```

The bottom snippet, titled 'Use Sklearn-like API on LLMs', shows a classification pipeline:

```
# 3) Define your classification dataset
X = ["Good product", "Does not work"]
y = ["positive", "negative"]

# 4) Define GPT Classifier
clf = ZeroShotGPTClassifier("gpt-3.5-turbo")

# 5) Fit Classifier
clf.fit(X, y)

# 6) Use Classifier for predictions
>>> clf.predict(X)
["positive", "negative"]
```

A curly brace on the left groups the first two snippets under the heading 'Set Config'. Another curly brace on the left groups the last four snippets under the heading 'Use Sklearn-like API on LLMs'.

Scikit-LLM is an open-source tool that offers a sklearn-compatible wrapper around OpenAI's API.

In simple words, it combines the power of LLMs with the elegance of sklearn API.

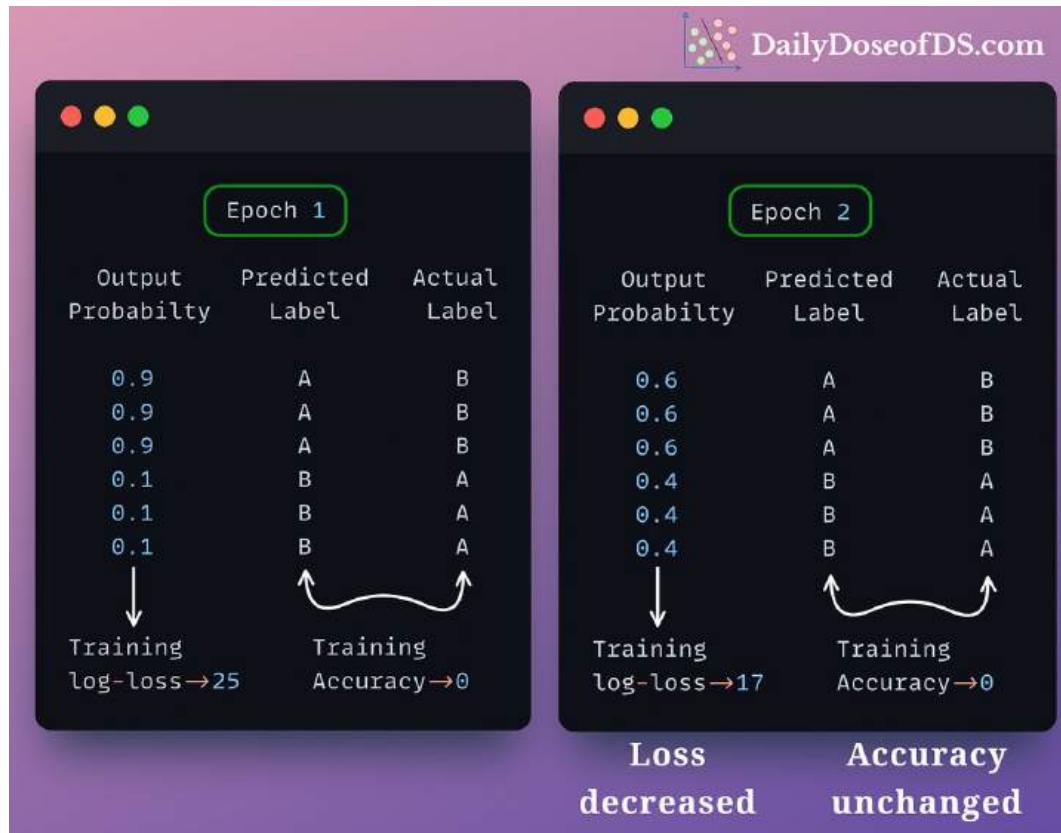
Thus, you can leverage LLMs using common sklearn functions such as fit, predict, score, etc.

What's more, you can also place LLMs in the sklearn pipeline.

Get started: [Scikit-LLM GitHub](https://github.com/Scikit-Learn-Contributors/scikit-llm).



# The Counterintuitive Behaviour of Training Accuracy and Training Loss



Intuitively, the training accuracy and loss are expected to be always inversely correlated.

It is expected that better predictions should lead to a lower training loss.

But that may not always be true.

In other words, you may see situations where the training loss decreases. Yet, the training accuracy remains unchanged (or even decreases).





Training loss and training accuracy decreasing

## But how could that happen?

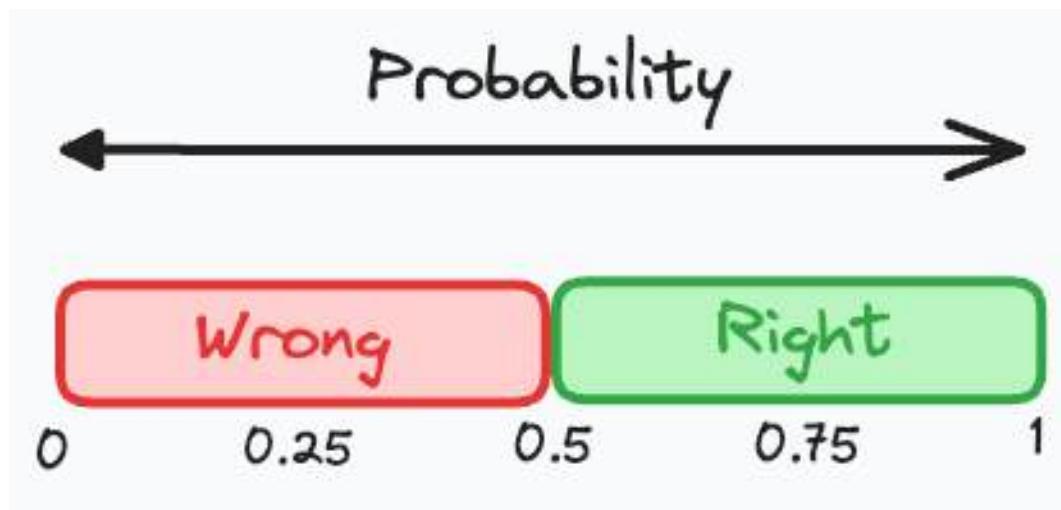
Firstly, understand that during training, the network ONLY cares about optimizing the loss function.

It does not care about the accuracy at all.

Accuracy is an external measure we introduce to measure the model's performance.

Now, when estimating the accuracy, the only thing we consider is whether we got a sample right or not.

Think of accuracy on a specific sample as a discrete measure of performance. Either right or wrong. That's it.



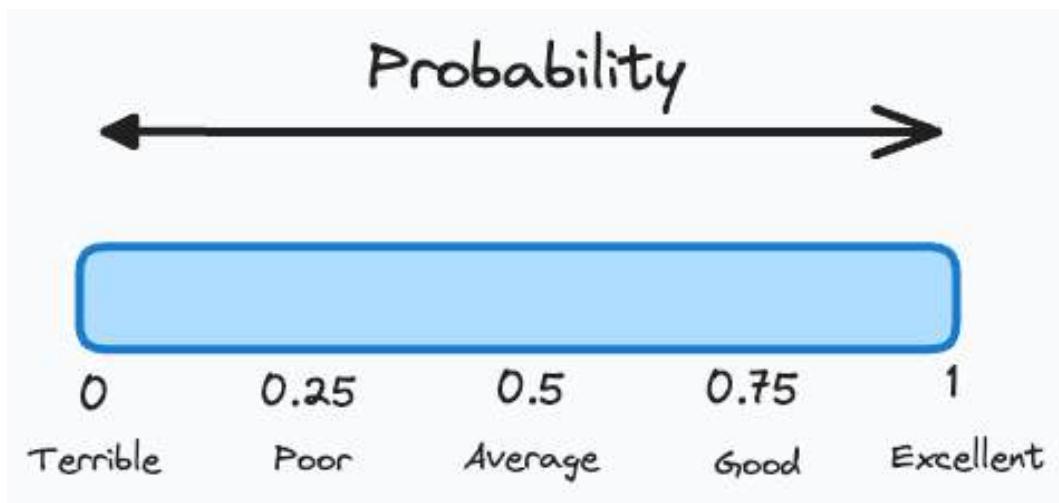
Accuracy computation

In other words, accuracy does not care if the network predicts a dog with 0.6 probability or 0.99 probability. They both have the same meaning (assuming the classification threshold is, say, 0.5).

However, the loss function is different.

It is a continuous measure of performance.

It considers how confident the model is about a prediction.



Loss spectrum

Thus, the loss function cares if the network predicted a dog with 0.6 probability or 0.99 probability.

This, at times, leads to the counterintuitive behavior of decreasing loss yet stable (or decreasing) accuracy.

If the training loss and accuracy both are decreasing, it may mean that the model is becoming:



Training loss and accuracy decreasing

More and more confident on correct predictions, and at the same time...

Less confident on its incorrect predictions.

But overall, it is making more mistakes than before.

If the training loss is decreasing, but the accuracy is stable, it may mean that the model is becoming:



Training loss decreasing, accuracy stable



More confident with its predictions. Given more time, it should improve.

But currently, it is not entirely confident to push them on either side of the probability threshold.

Having said that, remember that these kinds of fluctuations are quite normal, and you are likely to experience them.

The objective is to make you understand that while training accuracy and loss do appear to be seemingly negatively correlated, you may come across such counterintuitive situations.

Over to you: What do you think could be some other possible reasons for this behavior?



# A Highly Overlooked Point In The Implementation of Sigmoid Function

The screenshot shows a Jupyter Notebook cell with the following code:

```
def sigmoid(x):
    z = np.exp(-x)
    return 1/(1+z)

>>> sigmoid(+1000)    ✓
>>> sigmoid(-1000)    ✗
```

An annotation points from the text "Overflow for large negative values" to the second command, which fails.

The screenshot shows a Jupyter Notebook cell with the following code:

```
def sigmoid(x):
    if x > 0:
        z = np.exp(-x)
        return 1/(1+z)
    else:
        z = np.exp(x)
        return z/(1+z)
```

An annotation points from the text "Use two variations of sigmoid" to the first variation of the function.

The screenshot shows a Jupyter Notebook cell with the following commands:

```
>>> sigmoid(+1000)    ✓
>>> sigmoid(-1000)    ✓
```

An annotation points from the text "Prevent Overflow" to the second variation of the function.

There are two variations of the sigmoid function:

**Standard:** with an exponential term  $e^{-x}$  in the denominator only.



$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

**Rearranged:** with an exponential term  $e^x$  in both numerator and denominator.

$$\sigma(x) = \frac{e^x}{1 + e^x}$$

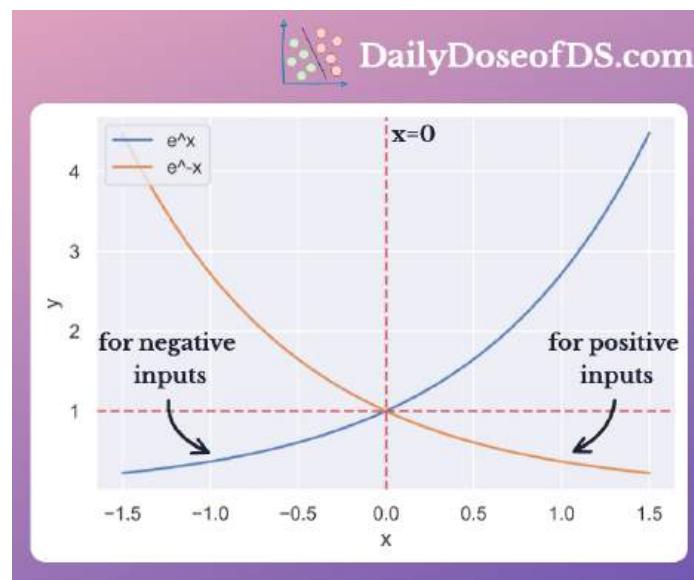
The standard sigmoid function can be easily computed for positive values. However, for large negative values, it raises overflow errors.

This is because, for large negative inputs,  $e^{-x}$  gets bigger and bigger.

To avoid this, use both variations of sigmoid.

Standard variation for positive inputs. This prevents overflow that may occur for negative inputs.

Rearranged variation for negative inputs. This prevents overflow that may occur for positive inputs.





This way, you can maintain numerical stability by preventing overflow errors in your ML pipeline.

Having said that, luckily, if you are using an existing framework, like Pytorch, you don't need to worry about this.

These implementations offer numerical stability by default. However, if you have a custom implementation, do give it a thought.

**Over to you:**

1. Sigmoids's two-variation implementation that I have shared above isn't vectorized. What is your solution to vectorize this?
2. What are some other ways in which numerical instability may arise in an ML pipeline? How to handle them?



# The Ultimate Categorization of Clustering Algorithms

Type of Clustering Algorithm	Visual Overview	Description	Algorithm(s)
Centroid-based		Cluster points based on proximity to centroid	KMeans KMeans++ KMedoids
Connectivity-based		Cluster points based on proximity between clusters	Hierarchical Clustering (Agglomerative and Divisive)
Density-based		Cluster points based on their density instead of proximity	DBSCAN OPTICS HDBSCAN
Graph-based		Cluster points based on graph distance	Affinity Propagation Spectral Clustering
Distribution-based		Cluster points based on their likelihood of belonging to the same distribution.	Gaussian Mixture Models
Compression-based		Transform data to a lower dimensional space and then perform clustering	BIRCH

Clustering is one of the core branches of unsupervised learning in ML.

It involves grouping data points together based on their inherent patterns or characteristics.

By identifying similarities (and dissimilarities) within a dataset, clustering helps in revealing underlying structures, discovering hidden patterns, and gaining insights into the data.

While centroid-based is the most common class of clustering, there's a whole world of algorithms beyond that, which we all should be aware of.

- **Centroid-based:** Cluster data points based on proximity to centroids.



- **Connectivity-based:** Cluster points based on proximity between clusters.
- **Density-based:** Cluster points based on their density.
- **Graph-based:** Cluster points based on graph distance.
- **Distribution-based:** Cluster points based on their likelihood of belonging to the same distribution.
- **Compression-based:** Transform data to a lower dimensional space and then perform clustering



# Improve Python Run-time Without Changing A Single Line of Code

The image shows three screenshots illustrating Python performance optimization:

- Code Editor:** Shows the Python code:

```
result = []
for a in range(10000):
    for b in range(10000):
        if (a+b)%11 == 0:
            result.append((a,b))
```
- Terminal 1 (Python):** Shows the command \$ python big\_loop.py and the output # Run-time: 10.9s.
- Terminal 2 (Pypy):** Shows the command \$ pypy big\_loop.py and the output # Run-time: 0.41s. An annotation above it says "Over 25x Faster" with an arrow pointing to the Pypy terminal.

Python's default interpreter — CPython, isn't smart at dealing with for-loops, lists, and more.

It serves as a standard interpreter for Python and offers no built-in optimization.

Pypy, however, is an alternative implementation of CPython, which is much smarter.

## How does it work?

It takes existing Python code and generates fast machine code using just-in-time compilation.



As a result, post compilation, the code runs at native machine code speed.

And Pypy can be used without modifying a single line of existing Python code.

👉 You should consider Pypy when:

- you're dealing with standard Python objects.
- speedups aren't possible with NumPy/Pandas.

So remember...

When you have some native Python code, don't run it with the default interpreter of Python.

Instead, look for alternative smarter implementations, such as Pypy.

Pypy will help you:

- improve run-time of code, and
- execute it at machine speed,
- without modifying the code.

Find some more benchmarking results between Python and Pypy below:

The screenshot shows a Mac desktop with two terminal windows side-by-side. Both windows have a dark theme with light-colored text. The top bar of each window displays the file name (fib.py or pi.py) and the word "Pypy" or "Python".

**Left Terminal (Python):**

- Content: Python code for calculating Fibonacci numbers and approximating pi.
- Output:
  - \$ python fib.py # N=35  
# Time: 2.53s
  - \$ python fib.py # N=45  
# Time: 296s
  - \$ python pi.py # n\_terms=10^8  
# Time: 14.7s

**Right Terminal (Pypy):**

- Content: Python code for calculating Fibonacci numbers and approximating pi.
- Output:
  - \$ pypy fib.py # N=35  
# Time: 0.08s (~30x Faster)
  - \$ pypy fib.py # N=45  
# Time: 11.2s (~27x Faster)
  - \$ pypy pi.py # n\_terms=10^8  
# Time: 0.49s (~30x Faster)

Get started with Pypy here: [Pypy docs](http://pypy.org).



# A Lesser-Known Feature of the Merge Method in Pandas

avichawla.substack.com

```
df1 = pd.DataFrame({'colA': ['A', 'B', 'C'],
                     'colB': [1, 2, 3]})

df2 = pd.DataFrame({'colA': ['A', 'B', 'C'],
                     'colC': [4, 5, 6]})

pd.merge(df1, df2, on = 'colA', validate='one_to_one')
```

Unique keys

	colA	colB	colC
0	A	1	4
1	B	2	5
2	C	3	6

One-one Merge  
Validated!  
No Merge Error

```
df1 = pd.DataFrame({'colA': ['A', 'B', 'C'],
                     'colB': [1, 2, 3]})

df2 = pd.DataFrame({'colA': ['A', 'B', 'B'],
                     'colC': [4, 5, 6]})
```

Repeated keys

```
pd.merge(df1, df2, on = 'colA', validate='one_to_one')
```

MergeError: Merge keys are not unique in right dataset; not a one-to-one merge  
One-one Merge Invalidated!

When merging two DataFrames, one may want to perform some checks to ensure the integrity of the merge operation.

For instance, if one of the two DataFrames has repeated keys, it will result in duplicated rows.

But this may not be desired.

The good thing is that you can check this with Pandas.

Pandas' `merge()` method provides a `validate` parameter, which checks if the merge is of a specified type or not.

- “`one_to_one`”: Merge keys should be unique in both DataFrames.
- “`one_to_many`”: Merge keys should be unique in the left DataFrame.
- “`many_to_one`”: Merge keys should be unique in the right DataFrame.



- “many\_to\_many”: Merge keys may or may not be unique in both DataFrames.

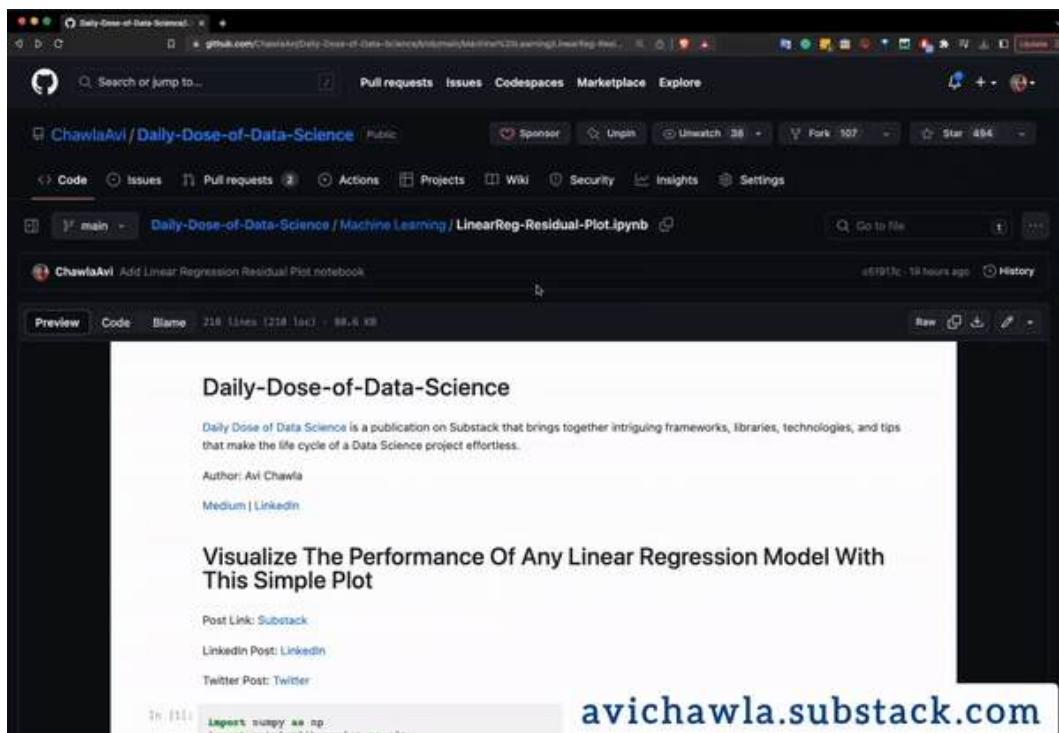
Pandas raises a **Merge Error** if the merge operation does not conform to the specified type.

This helps you prevent errors that can often go unnoticed.

Over to you: What are some other hidden treasures you know of in Pandas? Let me know :)



# The Coolest GitHub-Colab Integration You Would Ever See



If you have opened a notebook in GitHub, change the domain from "github" to "github~~t~~colab".

As a result, the notebook will directly open in Google Colab. Isn't that cool?

Try it out today.

Open any Jupyter Notebook hosted on GitHub, and change the domain as suggested above.



# Most Sklearn Users Don't Know This About Its LinearRegression Implementation

```
>>> X.shape
(100000, 2000) # Large no. of features

# Train LinearReg and SGDReg
>>> lin_reg = LinearRegression().fit(X, y)
>>> sgd_reg = SGDRegressor(...).fit(X, y)

# Model Weights (first 5 features):
lin_reg → [-0.36, -0.45, -0.62, 0.76, 0.84]
sgd_reg → [-0.37, -0.45, -0.62, 0.77, 0.85]

# Model Intercept:
lin_reg → 0.00241
sgd_reg → 0.00237

# Training Run-time:
lin_reg → 51 seconds
★ sgd_reg → 18 seconds ~3x Faster
```

**Similar weights**

**Similar intercept**

**★ sgd\_reg → 18 seconds ~3x Faster**

Sklearn's LinearRegression class implements the ordinary least square (OLS) method to find the best fit.

Some important characteristics of OLS are:

- It is a deterministic algorithm. If run multiple times, it will always converge to the same weights.
- It has no hyperparameters.
- It involves matrix inversion, which is cubic in relation to the no. of features. This gets computationally expensive with many features.

Read this answer to learn more about OLS' run-time complexity: [StackOverflow](#).



SGDRegressor, however:

- is a stochastic algorithm. It finds an approximate solution using optimization.
- has hyperparameters.
- involves gradient descent, which is relatively inexpensive.

Now, if you have many features, Sklearn's LinearRegression will be computationally expensive.

This is because it relies on OLS, which involves matrix inversion. And as mentioned above, inverting a matrix is cubic in relation to the total features.

This explains the run-time improvement provided by Sklearn's SGDRegressor over LinearRegression.

So remember...

When you have many features, avoid using Sklearn's LinearRegression.

Instead, use the [\*\*SGDRegressor\*\*](#).

- This will help you:
- Improve run-time.
- Avoid memory errors.

Implement batching (if needed). I have covered this in one of my previous posts here: [\*\*A Lesser-Known Feature of Sklearn To Train Models on Large Datasets\*\*](#).

👉 Over to you: What are some tradeoffs between using LinearRegression vs. SGDRegressor?



# Break the Linear Presentation of Notebooks With Stickyland

The screenshot illustrates the Stickyland extension for JupyterLab, demonstrating how it breaks the linear presentation of notebooks. It shows four different modes:

- A Notebook Cell:** A standard Jupyter notebook cell displaying code and output.
- B Sticky Dock:** A floating window where multiple cells can be organized and run simultaneously. It includes a "New" button, a list of existing cells, and a "Select new cell type" dropdown.
- C Sticky Cell:** A floating window containing a single cell's output, with an "auto-run" toggle switch.
- D Floating Cell:** A floating window containing a full notebook interface with multiple cells and their outputs.

The background of the screenshot features the Stickyland logo and the URL [avichawla.substack.com](http://avichawla.substack.com).

If you are a JupyterLab user, here's a pretty interesting extension for you.

Stickyland is an open-source tool that lets you break the linear presentation of a notebook.

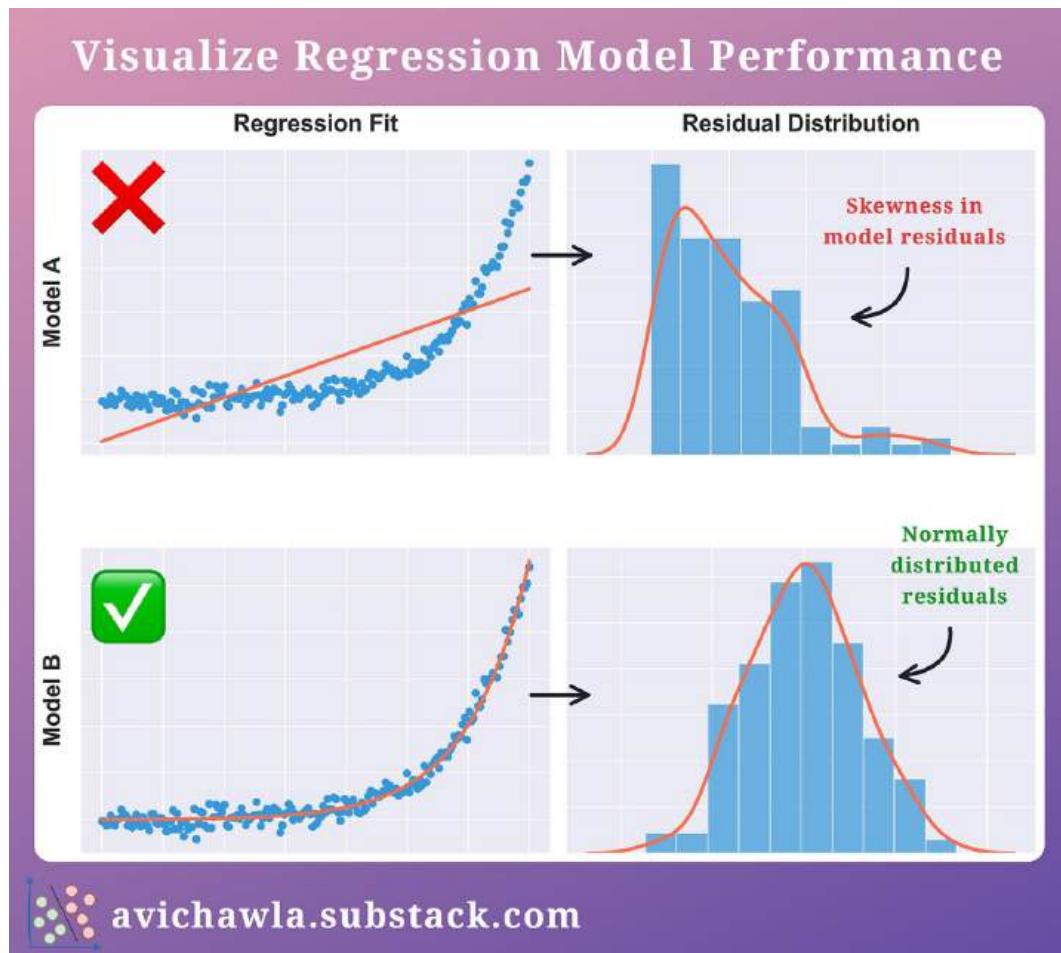
With Stickyland, you can:

- Create floating cells (code and markdown)
- Auto-run cells when any changes take place
- Take notes in Jupyter
- Organize the notebook as a dashboard

Find more info about Stickyland here: [GitHub](#) | [Paper](#).



# Visualize The Performance Of Any Linear Regression Model With This Simple Plot



Linear regression assumes that the model residuals (=actual-predicted) are normally distributed.

If the model is underperforming, it may be due to a violation of this assumption.

A residual distribution plot is a great way to verify this and also determine the model's performance.

As the name suggests, it depicts the distribution of residuals (=actual-predicted).

A good residual plot will:

- Follow a normal distribution
- NOT reveal trends in residuals



A bad residual plot will:

- Show skewness
- Reveal patterns in residuals

Thus, the more normally distributed the residual plot looks, the more confident you can be about your model.

This is especially useful when the regression line is difficult to visualize, i.e., in a high-dimensional dataset.

So remember...

After running a linear model, always check the distribution of the residuals.

This will help you:

- Validate the model's assumptions
- Determine how good your model is
- Find ways to improve it (if needed)

👉 Over to you: What are some other ways/plots to determine the linear model's performance?

Thanks for reading!



# Waterfall Charts: A Better Alternative to Line/Bar Plot



To visualize the change in value over time, a line (or bar) plot may not always be an apt choice.

A line-plot (or bar-plot) depicts the actual values in the chart. Thus, at times, it can be difficult to visually estimate the scale of incremental changes.

Instead, create a waterfall chart. It elegantly depicts these rolling differences.

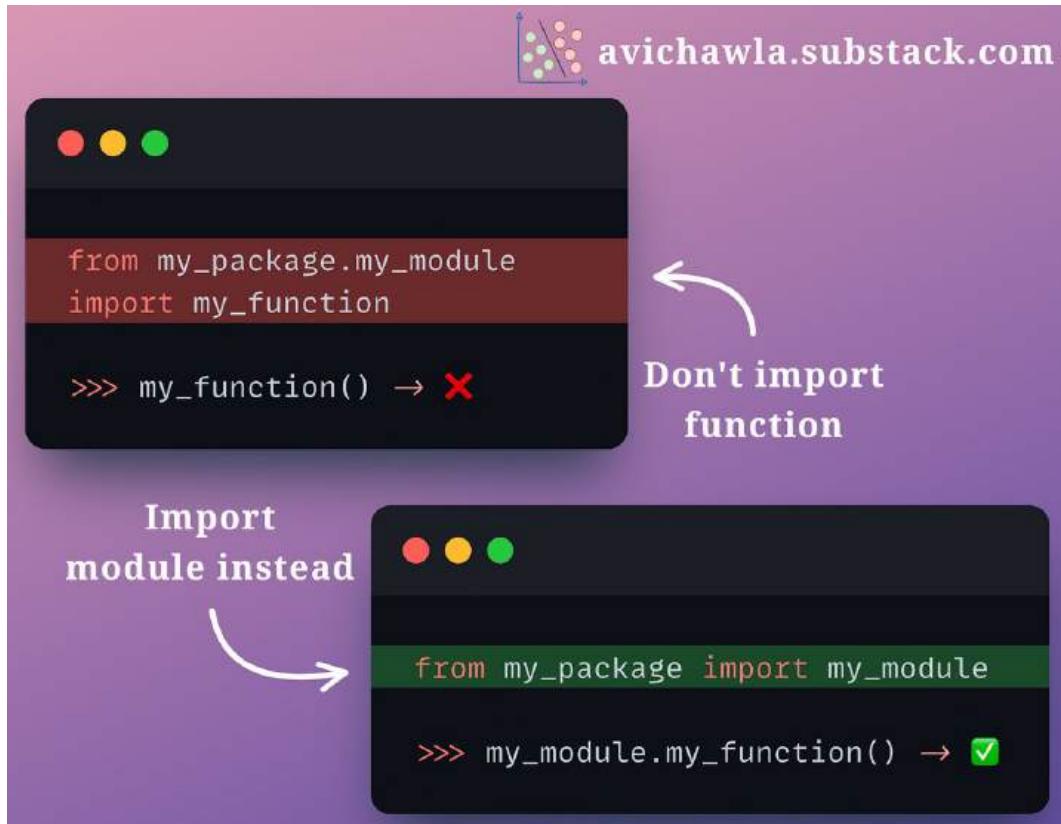
Here, the start and final values are represented by the first and last bars. Also, the marginal changes are automatically color-coded, making them easier to interpret.

Over to you: What are some other cases where Bar/Line plots aren't an ideal choice?

Read more here: [GitHub](#).



# What Does The Google Styling Guide Say About Imports



Recently, I was reviewing Google's Python style guide. Here's what it says about imports.

When writing import statements, it is recommended to import the module instead of a function.

Quoting from the style guide:

Use import statements for packages and modules only, not for individual classes or functions.

**Advantages:**

- 1. Namespace clarity:** Importing the module makes the source of the function clear during invocation. For instance, `my_module.my_function()` tells that `my_function` is defined in `my_module`.
- 2. Fewer import statements:** If you intend to use many functions from a specific module, importing each function may not be feasible, as shown below:



The diagram illustrates two code snippets side-by-side. The top snippet shows 'Multiple imports' where two functions from a module are imported separately:

```
from my_package.my_module import my_function1
from my_package.my_module import my_function2

>>> my_function1()
>>> my_function2()
```

The bottom snippet shows a 'Single import' where the entire module is imported, and then functions are called using the module name as a prefix:

```
from my_package import my_module

>>> my_module.my_function1()
>>> my_module.my_function2()
```

Curved arrows point from the labels 'Multiple imports' and 'Single import' to their respective code examples.

## Disadvantage:

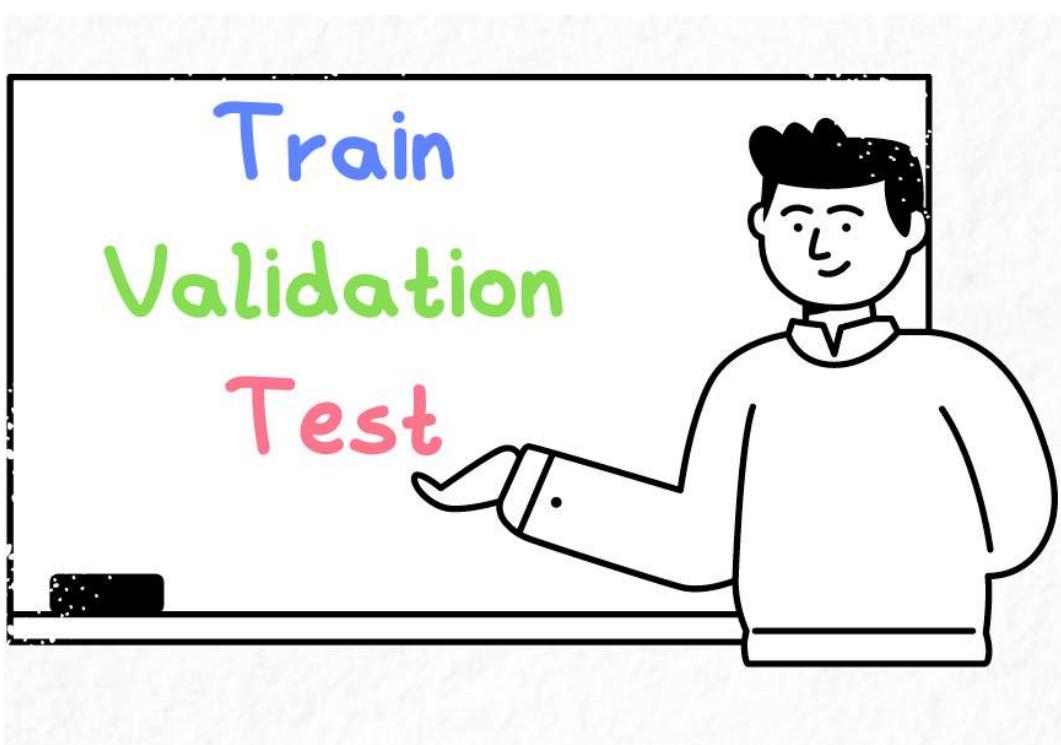
- **Explicitness:** When you import a function instead of a module, it's immediately clear which functions you intend to use from the module.

Over to you: While the guideline is intended for Google's internal stuff, it may be applicable elsewhere. Yet, when would you still prefer importing functions over modules? Let me know :)

Read the full guide here: [Google style guide](https://google.github.io/styleguide/pyguide.html#3.5-imports).



# How To Truly Use The Train, Validation and Test Set



Everyone knows about the train, test, and validation sets. But very few understand how to use them correctly.

Here's what you should know about splitting data and using it for ML models.

Begin by splitting the data into:

- Train
- Validation
- Test

Now, assume that the test data does not even exist. Forget about it instantly.

Begin with the train set. This is your whole world now.

- Analyze it
- Transform it
- Model it



As you finish modeling, you'd want to measure the model's performance on unseen data.

Bring in the validation set now.

Based on validation performance, improve the model.

Here's how you iteratively build your model:

- Train using a train set
- Evaluate it using the validation set
- Improve the model
- Evaluate again using the validation set
- Improve the model again
- and so on.

Until...

You start overfitting the validation set. This indicates that you exploited and polluted the validation set.

No worries.

Merge it with the train set and generate a new split of train and validation.

Note: Rely on cross-validation if needed, especially when you don't have much data. You may still use cross-validation if you have enough data. But it can be computationally intensive.

Now, if you are happy with the model's performance, evaluate it on test data.

---

#### 👉 What you use a test set for:

Get a final and unbiased review of the model.

#### 👉 What you DON'T use a test set for:

Analysis, decision-making, etc.

---

If the model is underperforming on the test set, no problem. Go back to the modeling stage and improve it.

BUT (and here's what most people do wrong)!

They use the same test set again.

**Not allowed!**



Think of it this way.

Your professor taught you in class. All in-class lessons and examples are the train set.

The professor gave you take-home assignments, which act like validation sets.

You get some wrong and some right.

Based on this, you adjust your topic fundamentals, i.e., improve the model.

If you keep solving the same take-home assignment, then you will eventually overfit it, won't you?

That is why we bring in a new validation set after some iterations.

The final exam day paper is your test set.

If you do well, awesome!

But if you fail, the professor cannot give you the same exam paper next time, can they? You know what's inside.

That is why we always use a specific test set only ONCE.

Once you do, merge it with the train and validation set and generate a new split.

Repeat.



# Restart Jupyter Kernel Without Losing Variables

```
In [ ]: # Define variable  
value = 2  
  
In [ ]: # Store it using magic command  
%store value  
  
Restart Kernel  
  
In [ ]: # Restore variable  
%store -r value  
  
In [ ]: value  
  
In [ ]:
```

[avichawla.substack.com](http://avichawla.substack.com)

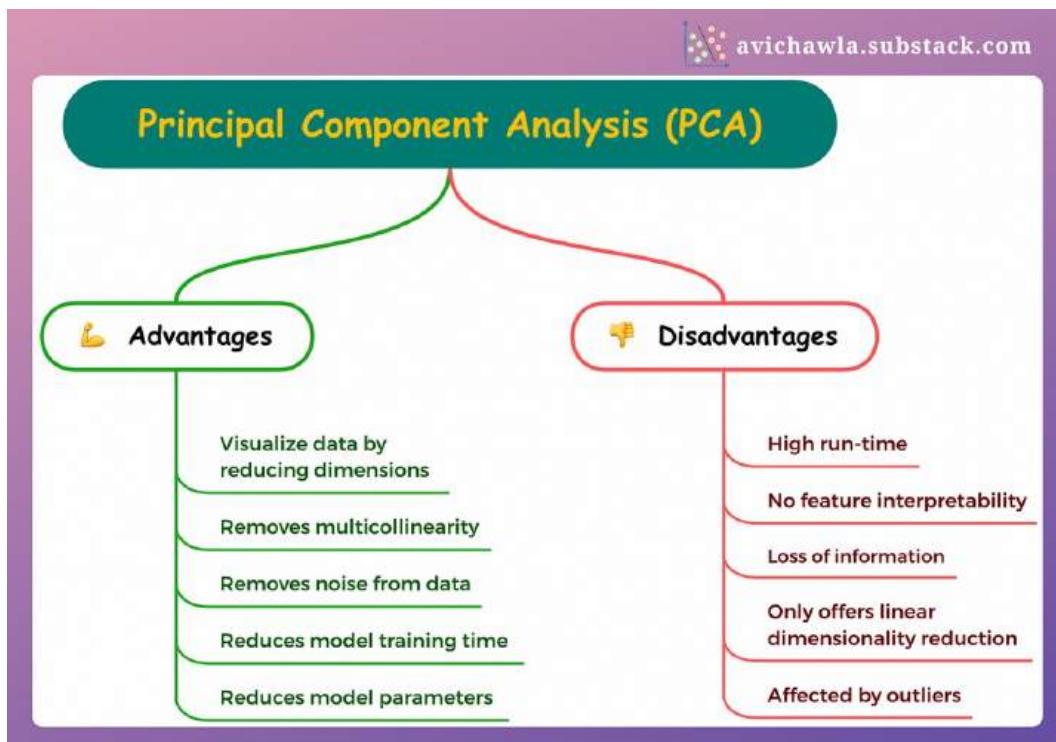
While working in a Jupyter Notebook, you may want to restart the kernel due to several reasons. Instead of dumping the variables to disk before restarting Jupyter and reading them back, use the **store magic command**.

It allows you to store and retrieve a variable back even if you restart the kernel. This avoids the hassle of dumping the object to disk.

Over to you: What are some other cool Jupyter hacks that you know of?



# The Advantages and Disadvantages of PCA To Consider Before Using It



PCA is possibly the most popular dimensionality reduction technique.

If you wish to know how PCA works, I have a highly simplified post here: [A Visual and Overly Simplified Guide to PCA](#).

Yet, it is equally important to be aware of what we get vs. what we compromise when we use PCA.

The above visual depicts five common pros and cons of using PCA.

## Advantages

By reducing the data to two dimensions, you can easily visualize it.

PCA removes multicollinearity. Multicollinearity arises when two features are correlated. PCA produces a set of new orthogonal axes to represent the data, which, as the name suggests, are uncorrelated.

PCA removes noise. By reducing the number of dimensions in the data, PCA can help remove noisy and irrelevant features.

PCA reduces model parameters: PCA can help reduce the number of parameters in machine learning models.



PCA reduces model training time. By reducing the number of dimensions, PCA simplifies the calculations involved in a model, leading to faster training times.

## Disadvantages

The run-time of PCA is cubic in relation to the number of dimensions of the data. This can be computationally expensive at times for large datasets.

$$\text{Runtime} : O(nd^2 + d^3)$$

*d : dimensions*

*n : samples*

PCA transforms the original input variables into new principal components (or dimensions). The new dimensions offer no interpretability.

While PCA simplifies the data and removes noise, it always leads to some loss of information when we reduce dimensions.

PCA is a linear dimensionality reduction technique, but not all real-world datasets may be linear. Read more about this in my previous post here: [The Limitation of PCA Which Many Folks Often Ignore.](#)

PCA gets affected by outliers. This can distort the principal components and affect the accuracy of the results.



# Loss Functions: An Algorithm-wise Comprehensive Summary



avichawla.substack.com

Algorithm	Commonly Used Loss Function or Training Methodology
Linear Regression	Mean Squared Error
Logistic Regression	Cross-Entropy Loss
Decision Tree Classifier	Information Gain or Gini impurity
Decision Tree Regressor	Mean Squared Error
Random Forest Classifier	Information Gain or Gini impurity
Random Forest Regressor	Mean Squared Error
Support Vector Machines (SVMs)	Hinge Loss
k-Nearest Neighbors	No loss function as kNN is non-parameteric
Naive Bayes	No loss function
Neural Networks	Regression: Mean Squared Error Classification: Cross-Entropy Loss
AdaBoost	Exponential loss
Gradient Boosting   LightGBM   CatBoost   XGBoost	Regression: Mean Squared Error Classification: Cross-Entropy Loss
KMeans Clustering	No loss function as KMeans is unsupervised

Loss functions are a key component of ML algorithms.

They specify the objective an algorithm should aim to optimize during its training. In other words, loss functions tell the algorithm what it should be trying to minimize or maximize in order to improve its performance.





Therefore, knowing which loss functions are best suited for specific ML algorithms is extremely crucial.

The above visual depicts the most commonly used loss functions with various ML algorithms.

**1. Linear Regression:** Mean Squared Error (MSE). This can be used with and without regularization, depending on the situation.

**2. Logistic regression:** Cross-Entropy Loss or Log Loss, with and without regularization.

### **3. Decision Tree and Random Forest:**

- a. Classifier: Gini impurity or information gain.
- b. Regressor: Mean Squared Error (MSE)

**4. Support Vector Machines (SVMs):** Hinge loss. Read more: [Wikipedia](#).

**5. k-Nearest Neighbors (kNN):** No loss function. kNN is a non-parametric lazy learning algorithm. It works by retrieving instances from the training data, and making predictions based on the k nearest neighbors to the test data instance.

**6. Naive Bayes:** No loss function. Can you guess why? Let me know in the comments if you need help.

**7. Neural Networks:** They can use a variety of loss functions depending on the type of problem. The most common are:

- a. Regression: Mean Squared Error (MSE).
- b. Classification: Cross-Entropy Loss.

**8. AdaBoost:** Exponential loss function. AdaBoost is an ensemble learning algorithm. It combines multiple weak classifiers to form a strong classifier. In each iteration of the algorithm, AdaBoost assigns weights to the misclassified instances from the previous iteration. Next, it trains a new weak classifier and minimizes the weighted exponential loss.

### **9. Other Boosting Algorithms:**

- a. Regression: Mean Squared Error (MSE).
- b. Classification: Cross-Entropy Loss.

**10. KMeans Clustering:** No loss function.

Over to you: Which algorithms have I missed?



# Is Data Normalization Always Necessary Before Training ML Models?

Data normalization is commonly used to improve the performance and stability of ML models.

This is because normalization scales the data to a standard range. This prevents a specific feature from having a strong influence on the model's output. What's more, it ensures that the model is more robust to variations in the data.



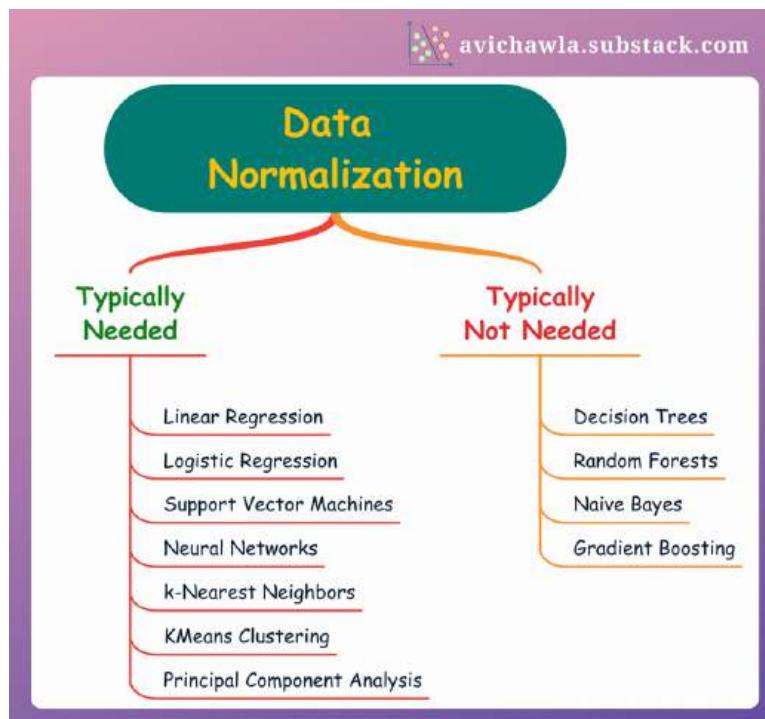
Different scales of columns

For instance, in the image above, the scale of **Income** could massively impact the overall prediction. Normalizing the data by scaling both features to the same range can mitigate this and improve the model's performance.

But is it always necessary?

While normalizing data is crucial in many cases, knowing when to do it is also equally important.

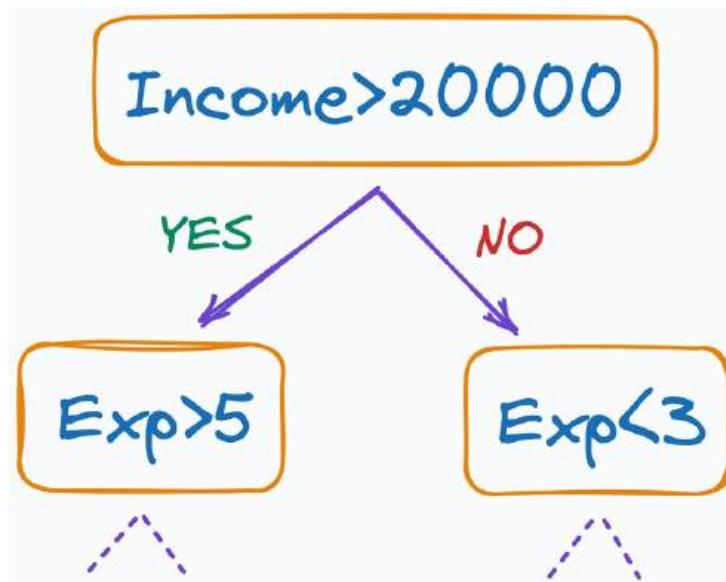
The following visual depicts which algorithms typically need normalized data and which don't.



Categorization of algorithms based on normalized data requirement

As shown above, many algorithms typically do not need normalized data. These include decision trees, random forests, naive bayes, gradient boosting, and more.

Consider a decision tree, for instance. It splits the data based on thresholds determined solely by the feature values, regardless of their scale.



Decision tree



Thus, it's important to understand the nature of your data and the algorithm you intend to use.

You may never need data normalization if the algorithm is insensitive to the scale of the data.

Over to you: What other algorithms typically work well without normalizing data? Let me know :)



# Annotate Data With The Click Of A Button Using Pigeon

The screenshot shows a Jupyter notebook cell with Python code for annotation. The code imports the Pigeon library and uses its `annotate` function on a list of image files, specifying categories ('cat' or 'dog') and a display function. Below the code, there are three buttons: 'cat', 'dog', and 'skip'. A large image of a brown tabby cat is centered below the buttons. At the bottom of the cell, the annotated data is shown as a list of tuples:

```
annotations ## get labels
[('./cats and dogs/image1.jpeg', 'cat'),
 ('./cats and dogs/image2.jpeg', 'dog'),
 ('./cats and dogs/image3.jpeg', 'dog')]
```

To perform supervised learning, you need labeled data. But if your data is unlabeled, you must spend some time annotating/labeling it.

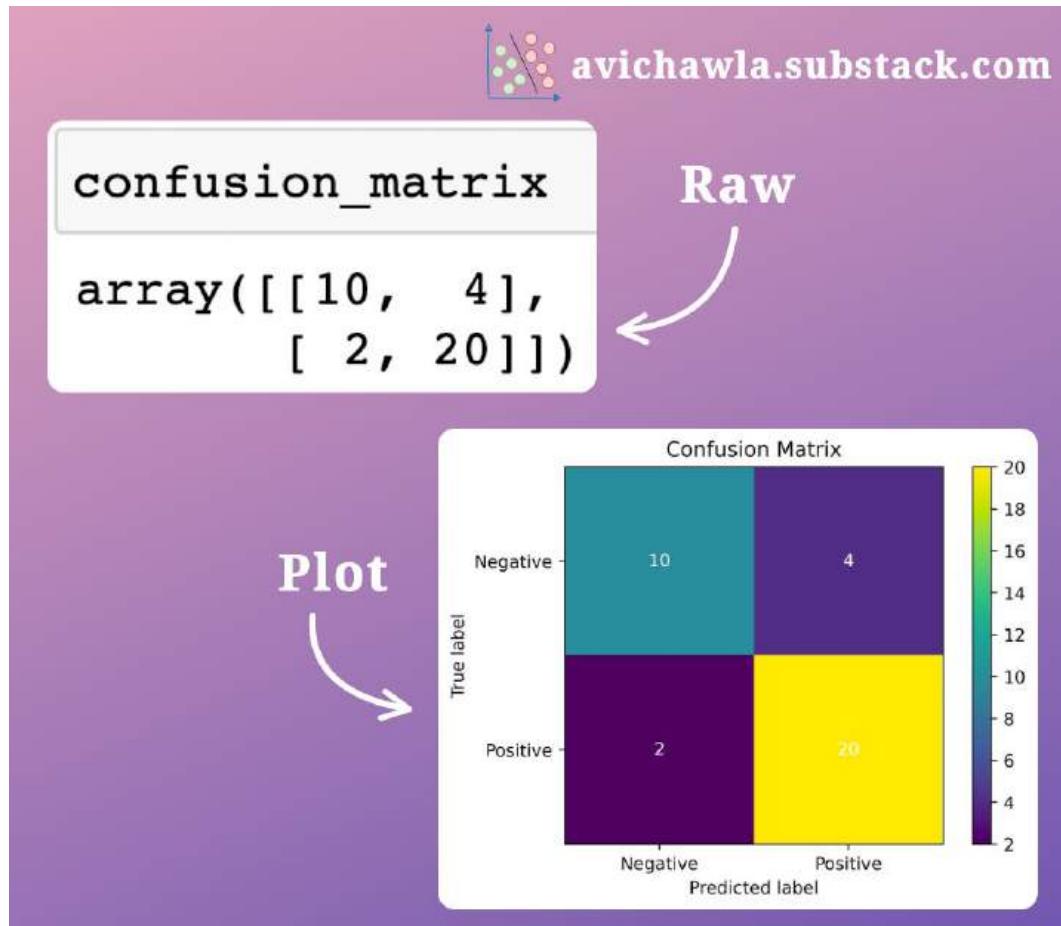
To do this quickly in a Jupyter notebook, use Pigeon. With this, you can annotate your data by simply defining buttons and labeling samples.

Read more: [Pigeon GitHub](#).



# Enrich Your Confusion Matrix With A Sankey Diagram

A confusion matrix is mostly interpreted as is, i.e., raw numbers. Sometimes though, it is also visualized by plotting it.



Traditional ways of previewing confusion matrix

Yet, both these ways are not interactive and truly elegant.

Plotting a confusion matrix as a Sankey diagram is an option worth exploring here.

If you wish to read more about Sankey Diagrams, read my previous post here: [Analyse Flow Data With Sankey Diagrams](#).

As demonstrated below, one can interactively interpret the number of instances belonging to each class and how they were classified.



```
confusion_matrix = np.array([[10, 4],  
                            [2, 20]])  
  
plot_confusion_matrix_as_sankey(confusion_matrix, ['Fraud', 'Legit'])
```



## Confusion Matrix Sankey Diagram



### Confusion matrix as Sankey Diagram

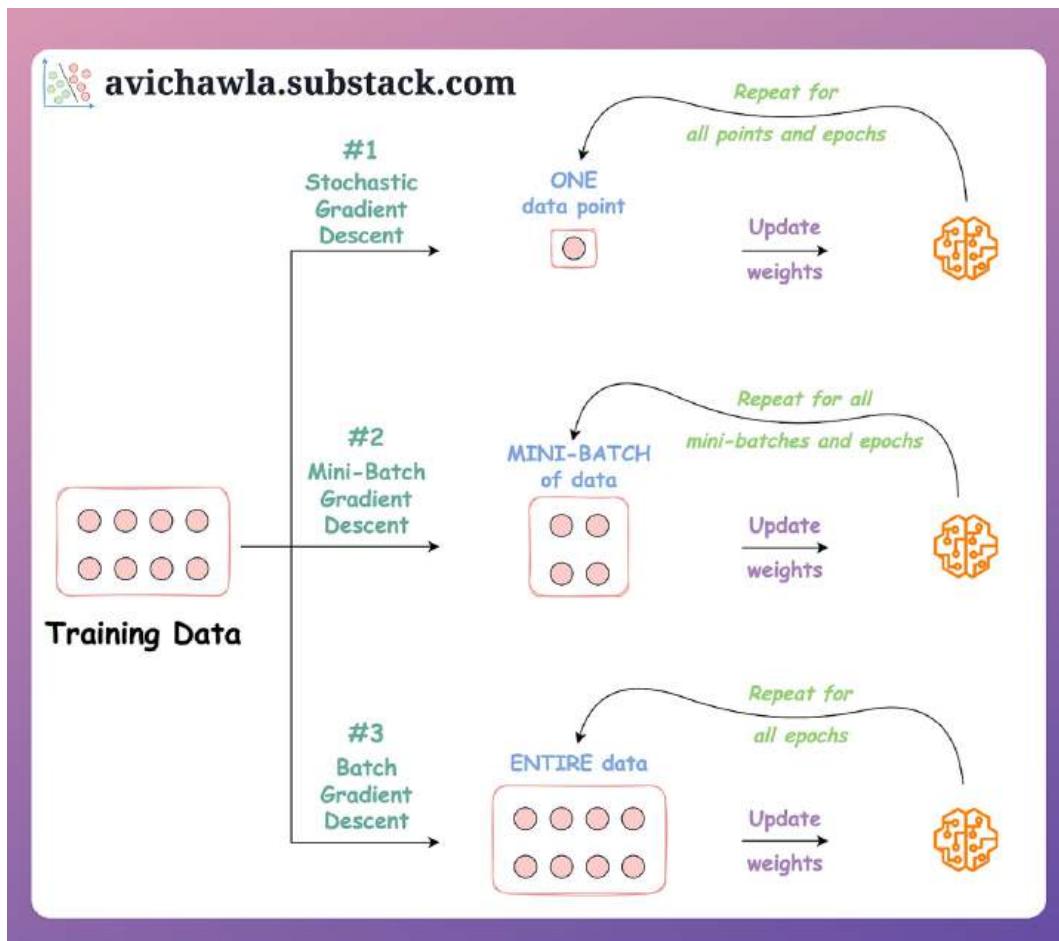
What's more, hovering over the links gives more info about those instances, which can offer better interpretability.

What do you think? Is this a better approach than the traditional ones?  
Let me know :)

**Find the code for creating the above Confusion Matrix Sankey diagram here: [Notebook](#).**



# A Visual Guide to Stochastic, Mini-batch, and Batch Gradient Descent

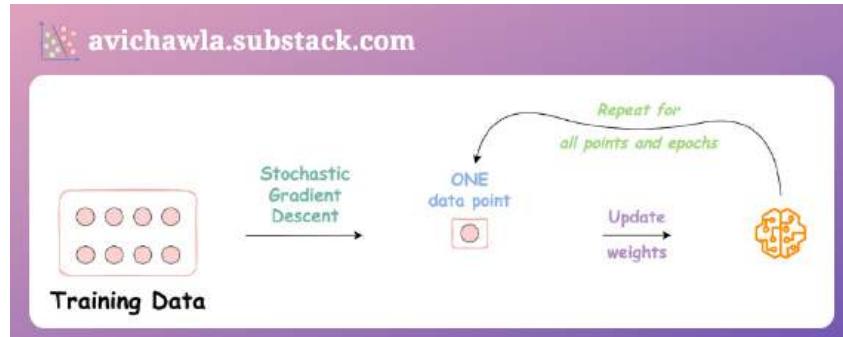


Gradient descent is a widely used optimization algorithm for training machine learning models.

Stochastic, mini-batch, and batch gradient descent are three different variations of gradient descent, and they are distinguished by the number of data points used to update the model weights at each iteration.

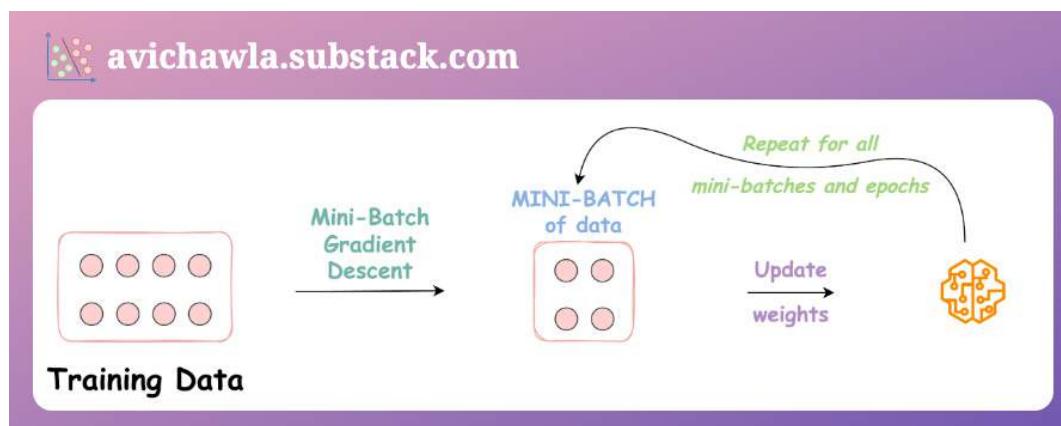


- ◆ Stochastic gradient descent: Update network weights using one data point at a time.



- **Advantages:**
  - Easier to fit in memory.
  - Can converge faster on large datasets and can help avoid local minima due to oscillations.
- **Disadvantages:**
  - Noisy steps can lead to slower convergence and require more tuning of hyperparameters.
  - Computationally expensive due to frequent updates.
  - Loses the advantage of vectorized operations.

- ◆ Mini-batch gradient descent: Update network weights using a few data points at a time.



- **Advantages:**
  - More computationally efficient than batch gradient descent due to vectorization benefits.

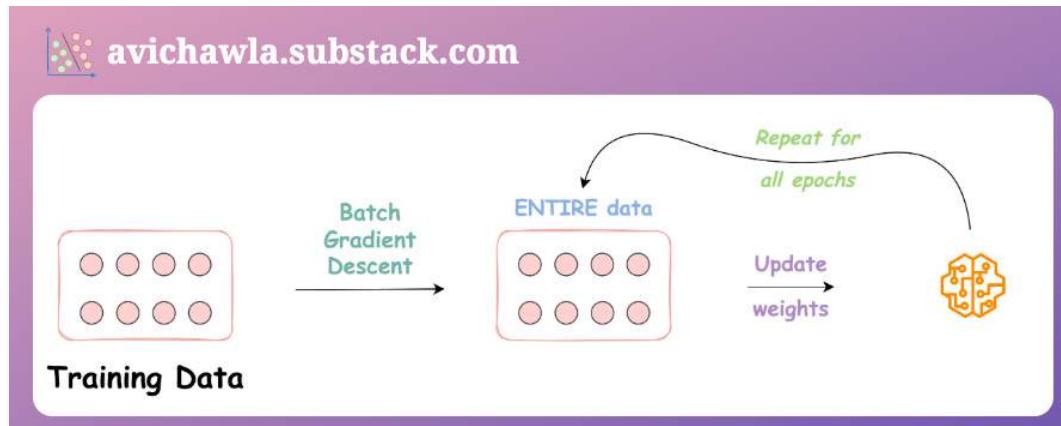


- Less noisy updates than stochastic gradient descent.

- **Disadvantages:**

- Requires tuning of batch size.
- May not converge to a global minimum if the batch size is not well-tuned.

◆ Batch gradient descent: Update network weights using the entire data at once.



- **Advantages:**

- Less noisy steps taken towards global minima.
- Can benefit from vectorization.
- Produces a more stable convergence.

- **Disadvantages:**

- Enforces memory constraints for large datasets.
- Computationally slow as many gradients are computed, and all weights are updated at once.

Over to you: What are some other advantages/disadvantages you can think of? Let me know :)



# A Lesser-Known Difference Between For-Loops and List Comprehensions

```
>>> a = 1  
>>> for a in range(6):  
    pass  
  
>>> print(a)  
5 # Output
```

```
>>> a = 1  
>>> [... for a in range(6)]  
  
>>> print(a)  
1 # Output
```

In the code above, the for-loop updated the existing variable (`a`), but list comprehension didn't. Can you guess why? Read more to know.

A loop variable is handled differently in for-loops and list comprehensions.

A for-loop leaks the loop variable into the surrounding scope. In other words, once the loop is over, you can still access the loop variable.

We can verify this below:



```
>>> for loop_var in range(6):
...
>>> print(loop_var) ## Loop variable accessible
5
```

No error

In the main snippet above, as the loop variable (`a`) already existed, it was overwritten in each iteration.

But a list comprehension does not work this way. Instead, the loop variable always remains local to the list comprehension. It is never leaked outside.

We can verify this below:

```
>>> [... for loop_var in range(6)]
>>> print(loop_var)
NameError: name 'loop_var' is not defined
```

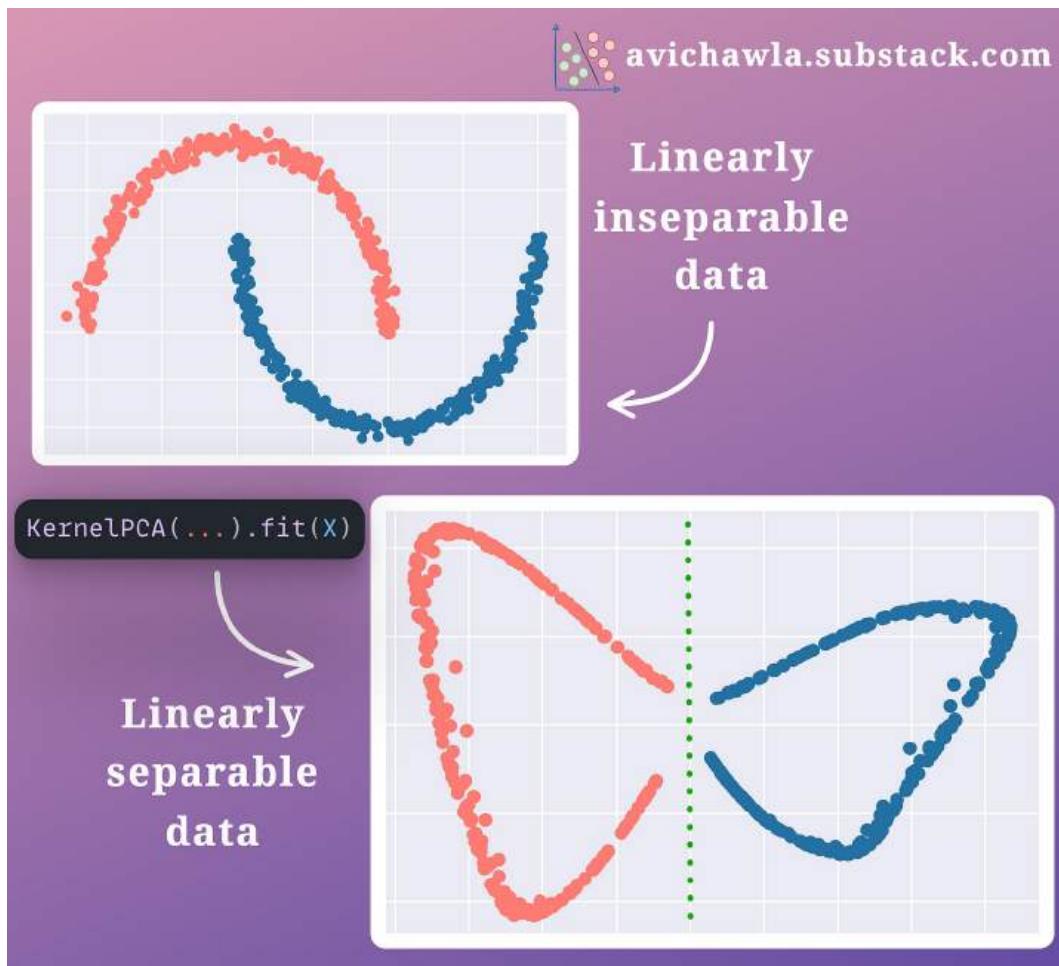
Error

That is why the existing variable (`a`), which was also used inside the list comprehension, remained unchanged. The list comprehension defined the loop variable (`a`) local to its scope.

Over to you: What are some other differences that you know of between for-loops and list comprehension?

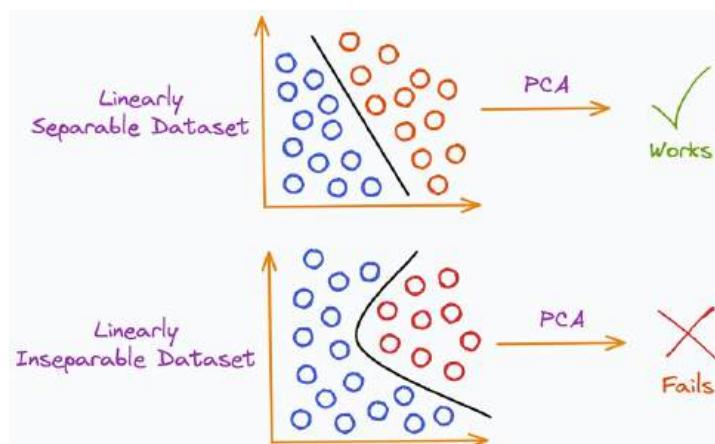


# The Limitation of PCA Which Many Folks Often Ignore



Imagine you have a classification dataset. If you use PCA to reduce dimensions, it is inherently assumed that your data is linearly separable.

But it may not be the case always. Thus, PCA will fail in such cases.





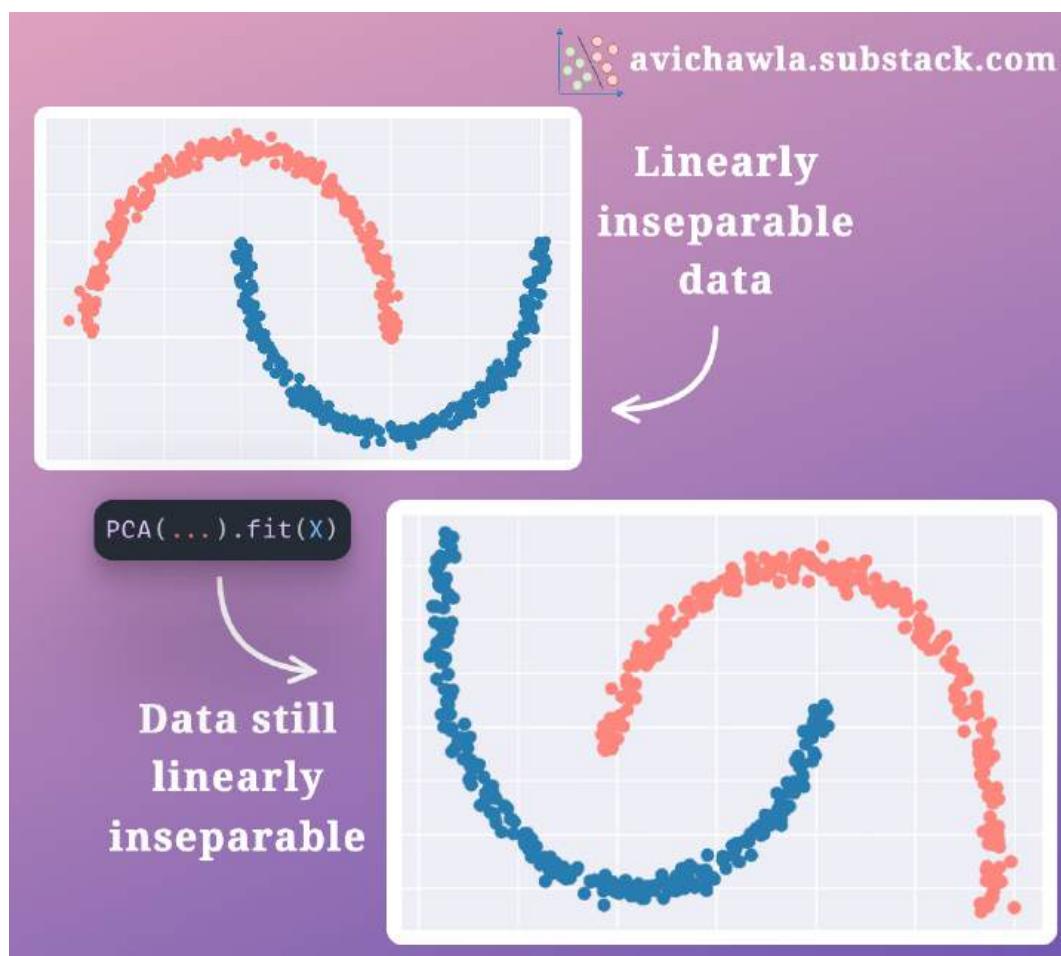
If you wish to read how PCA works, I would highly recommend reading one of my previous posts: [A Visual and Overly Simplified Guide to PCA](#).

To resolve this, we use the kernel trick (or the KernelPCA). The idea is to:

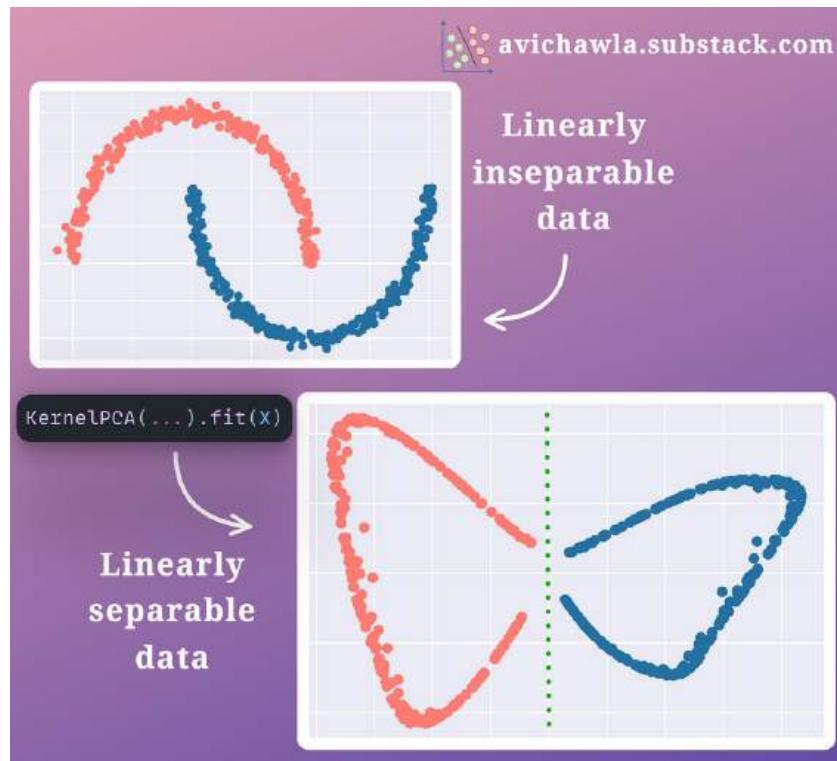
Project the data to another space using a kernel function, where the data becomes linearly separable.

Apply the standard PCA algorithm to the transformed data.

For instance, in the image below, the original data is linearly inseparable. Using PCA directly does not produce any desirable results.



But as mentioned above, KernelPCA first transforms the data to a linearly separable space and then applies PCA, resulting in a linearly separable dataset.



Sklearn provides a KernelPCA wrapper, supporting many popularly used kernel functions. You can find more details here: [Sklearn Docs](#).

Having said that, it is also worth noting that the run-time of PCA is cubic in relation to the number of dimensions of the data.

$$\text{Runtime} : O(nd^2 + d^3)$$

$d$  : dimensions

$n$  : samples

When we use a KernelPCA, typically, the original data (in  $n$  dimensions) is projected to a new higher dimensional space (in  $m$  dimensions;  $m > n$ ). Therefore, it increases the overall run-time of PCA.



# Magic Methods: An Underrated Gem of Python OOP



avichawla.substack.com

Magic Method	Syntax	Usage/Description
<code>__new__</code>	<code>__new__(cls, *args, **kwargs):</code>	Invoked before <code>__init__</code> to allocate memory to object
<code>__init__</code>	<code>__init__(self, *args, **kwargs):</code>	Invoked after <code>__new__</code> to initialise the object
<code>__str__</code>	<code>__str__(self):</code>	Invoked when <code>str(obj)</code> or <code>print(obj)</code> is used
<code>__int__</code>	<code>__int__(self):</code>	Invoked when <code>int(obj)</code> is used
<code>__len__</code>	<code>__len__(self):</code>	Invoked when <code>len(obj)</code> is used
<code>__call__</code>	<code>__call__(self, *args, **kwargs):</code>	Invoked when class object is called as a function: <code>obj()</code>
<code>__getitem__</code>	<code>__getitem__(self, key):</code>	Invoked when object is indexed: <code>obj[key]</code>
<code>__setitem__</code>	<code>__setitem__(self, key, value):</code>	Invoked when object is indexed and value is set: <code>obj[key]=value</code>
<code>__delitem__</code>	<code>__delitem__(self, key):</code>	Invoked when object's index is deleted: <code>del obj[key]</code>
<code>__contains__</code>	<code>__contains__(self, item):</code>	Invoked when the <code>in</code> operator is used: <code>item in obj</code>
<code>__bool__</code>	<code>__bool__(self):</code>	Invoked when object is used in boolean context: <code>if obj</code> or <code>bool(obj)</code>
<code>__iter__</code>	<code>__iter__(self):</code>	Invoked when object is iterated: <code>for x in obj</code>
<code>__eq__</code>	<code>__eq__(self, other):</code>	Invoked when <code>==</code> operator is used to compare two objects: <code>obj1 == obj2</code>
<code>__ne__</code>	<code>__ne__(self, other):</code>	Invoked when <code>!=</code> operator is used to compare two objects: <code>obj1 != obj2</code>
<code>__add__</code>	<code>__add__(self, other):</code>	Invoked when two objects are added: <code>obj1 + obj2</code>
<code>__mul__</code>	<code>__mul__(self, other):</code>	Invoked when two objects are multiplied: <code>obj1 * obj2</code>
<code>__abs__</code>	<code>__abs__(self):</code>	Invoked to compute absolute value of object: <code>abs(obj)</code>
<code>__neg__</code>	<code>__neg__(self):</code>	Invoked when unary operator <code>-</code> is used on an object: <code>-obj</code>
<code>__invert__</code>	<code>__invert__(self):</code>	Invoked when <code>~(tilde)</code> operator is used to invert an object: <code>~obj</code>

Magic Methods (also called **dunder methods**) are special methods defined inside a Python class' implementation.

*On a side note, the word “Dunder” is short for Double **Underscore**.*

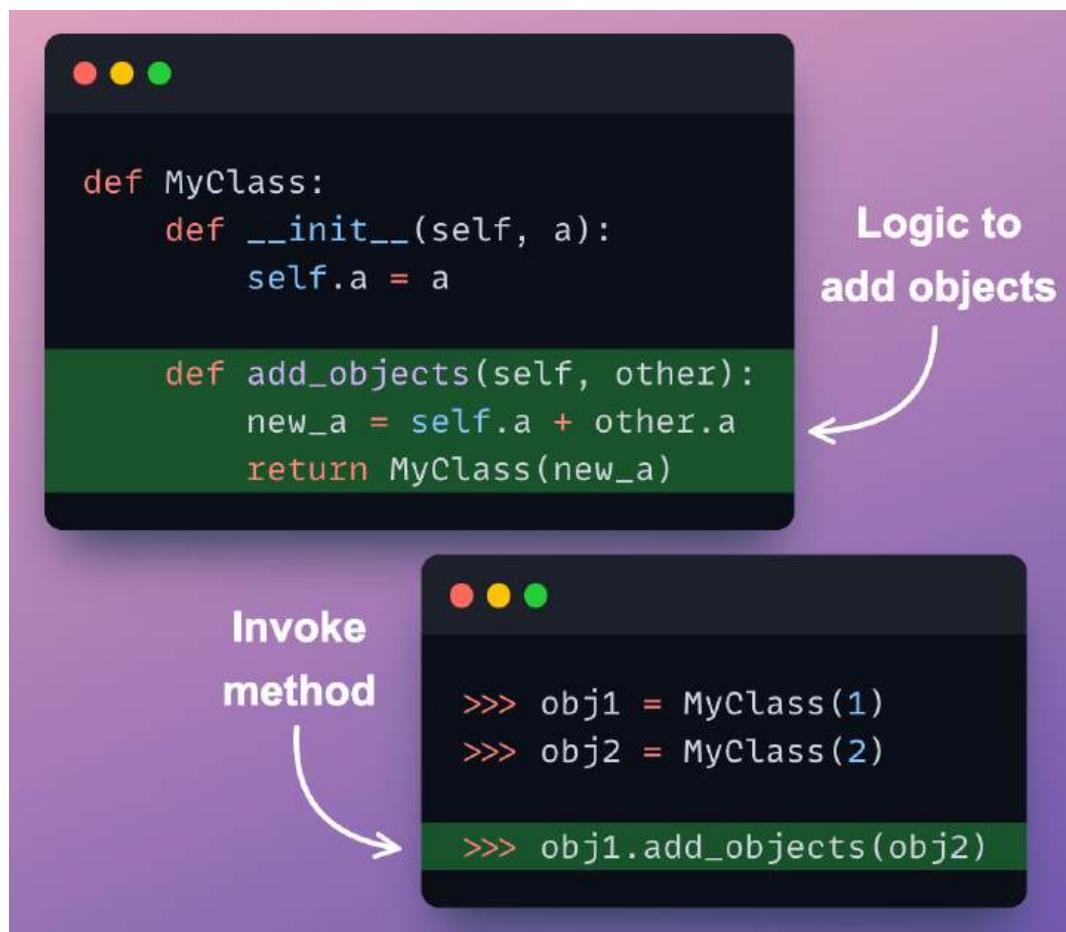
They are prefixed and suffixed with double underscores, such as `__len__`, `__str__`, and many more.



Magic Methods offer immense flexibility to define the behavior of class objects in certain scenarios.

For instance, say we want to define a custom behavior for adding two objects of our class (`obj1 + obj2`).

An obvious and straightforward way to do this is by defining a method, say `add_objects()`, and passing the two objects as its argument.



While the above approach will work, invoking a method explicitly for adding two objects isn't as elegant as using the `+` operator:

```
>>> obj1 + obj2
```



This is where magic methods come in. In the above example, implementing the `__add__` magic method will allow you to add the two objects using the `+` operator instead.

Thus, magic methods allow us to make our classes more intuitive and easier to work with.

As a result, awareness about them is extremely crucial for developing elegant, and intuitive pipelines.

The visual summarizes **~20** most commonly used magic methods in Python.

Over to you: What other magic methods will you include here? Which ones do you use the most? Let me know :)



# The Taxonomy Of Regression Algorithms That Many Don't Bother To Remember

	Regression Type	Description	Equation/Loss Function
Linear Regression	Simple Linear Regression	One independent (x) and one dependent (y) variable	$\hat{y} = wx + b$ $Loss = \sum \frac{(y - \hat{y})^2}{n}$
	Polynomial Linear Regression	Polynomial features ( $x, x^2, \dots, x^n$ ) and one dependent (y) variable	$\hat{y} = w_1x + w_2x^2 + \dots + b$ $Loss = \sum \frac{(y - \hat{y})^2}{n}$
	Multiple Linear Regression	Arbitrary features ( $x_1, x_2, \dots, x_n$ ) and one dependent (y) variable	$\hat{y} = w_1x_1 + w_2x_2 + \dots + b$ $Loss = \sum \frac{(y - \hat{y})^2}{n}$
Regularized Regression	Ridge Regression	Linear Regression with L2 Regularization	$Loss = \sum \frac{(y - \hat{y})^2}{n} + \lambda \sum_{i=1}^n w_i^2$
	Lasso Regression	Linear Regression with L1 Regularization	$Loss = \sum \frac{(y - \hat{y})^2}{n} + \lambda \sum_{i=1}^n  w_i $
	Elastic Net	Linear Regression with BOTH L1 and L2 Regularization	$Loss = \sum \frac{(y - \hat{y})^2}{n} + \lambda((1 - \alpha) * \sum_{L2} w_i^2 + \alpha * \sum_{L1}  w_i )$
Categorical Probability	Logistic Regression	One (or more) independent variable(s) to predict BINARY outcome probability	$P(X) = \frac{1}{1 + e^{-(w_1x_1 + w_2x_2 + \dots + b)}}$
	Multinomial Logistic Regression (or Softmax Regression)	One (or more) independent variable(s) to predict MULTIPLE categorical probabilities	$P(Y = k X) = \frac{e^{score_k}}{\sum_{j=1}^K e^{score_j}}$

Regression algorithms allow us to model the relationship between a dependent variable and one or more independent variables.

After estimating the parameters of a regression model, we can gain insight into how changes in one variable affect another.

Being widely used in data science, an awareness of their various forms is crucial to precisely convey which algorithm you are using.

Here are eight of the most standard regression algorithms described in a single line:

## Linear Regression

- **Simple linear regression:** One independent (x) and one dependent (y) variable.
- **Polynomial Linear Regression:** Polynomial features and one dependent (y) variable.



- **Multiple Linear Regression:** Arbitrary features and one dependent ( $y$ ) variable.

## Regularized Regression

- **Lasso Regression:** Linear Regression with L1 Regularization.
- **Ridge Regression:** Linear Regression with L2 Regularization.
- **Elastic Net:** Linear Regression with BOTH L1 and L2 Regularization.

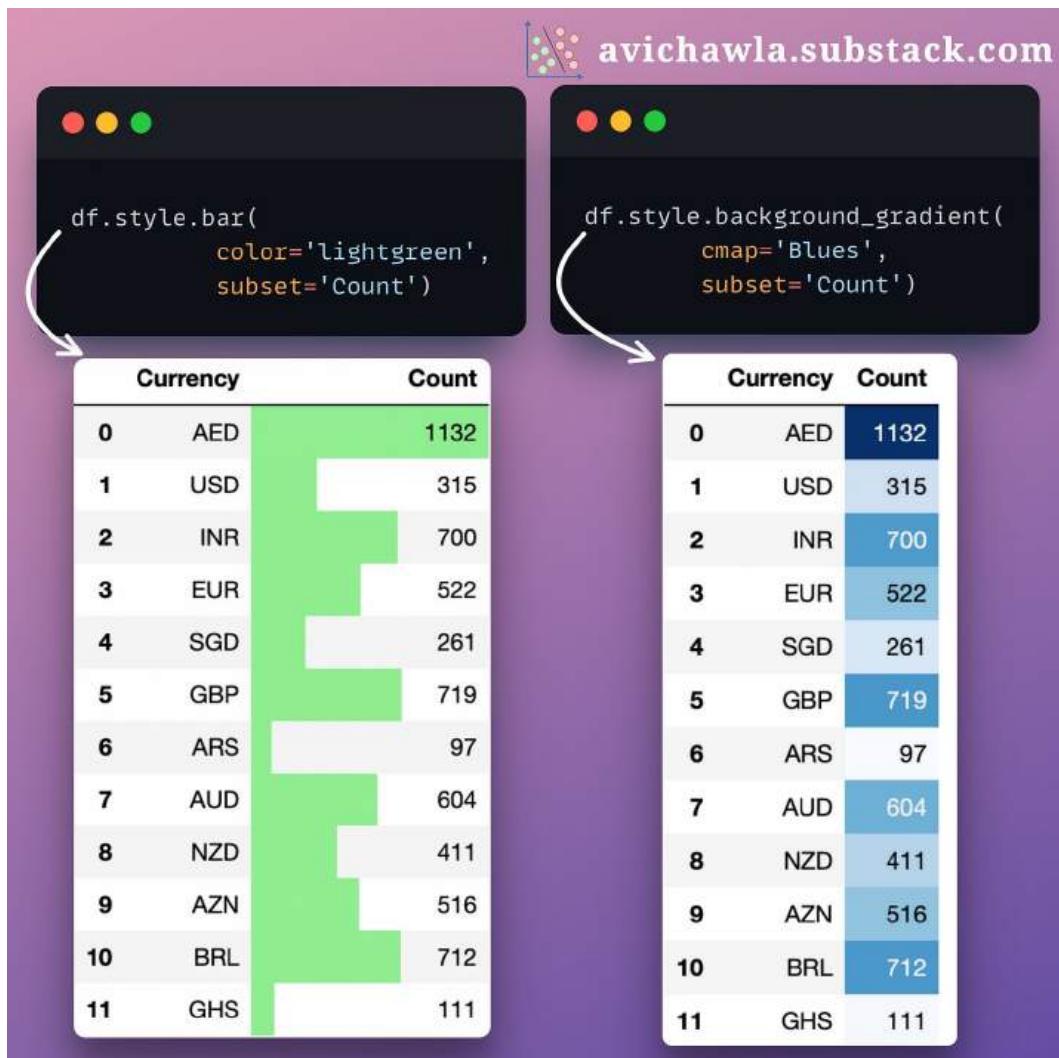
## Categorical Probability Prediction

- **Logistic Regression:** Predict binary outcome probability.
- **Multinomial Logistic Regression (or Softmax Regression):** Predict multiple categorical probabilities.

Over to you: What other regressions algorithms will you include here?



# A Highly Overlooked Approach To Analysing Pandas DataFrames



Instead of previewing raw DataFrames, styling can make data analysis much easier and faster. Here's how.

Jupyter is a web-based IDE. Anything you print is rendered using HTML and CSS.

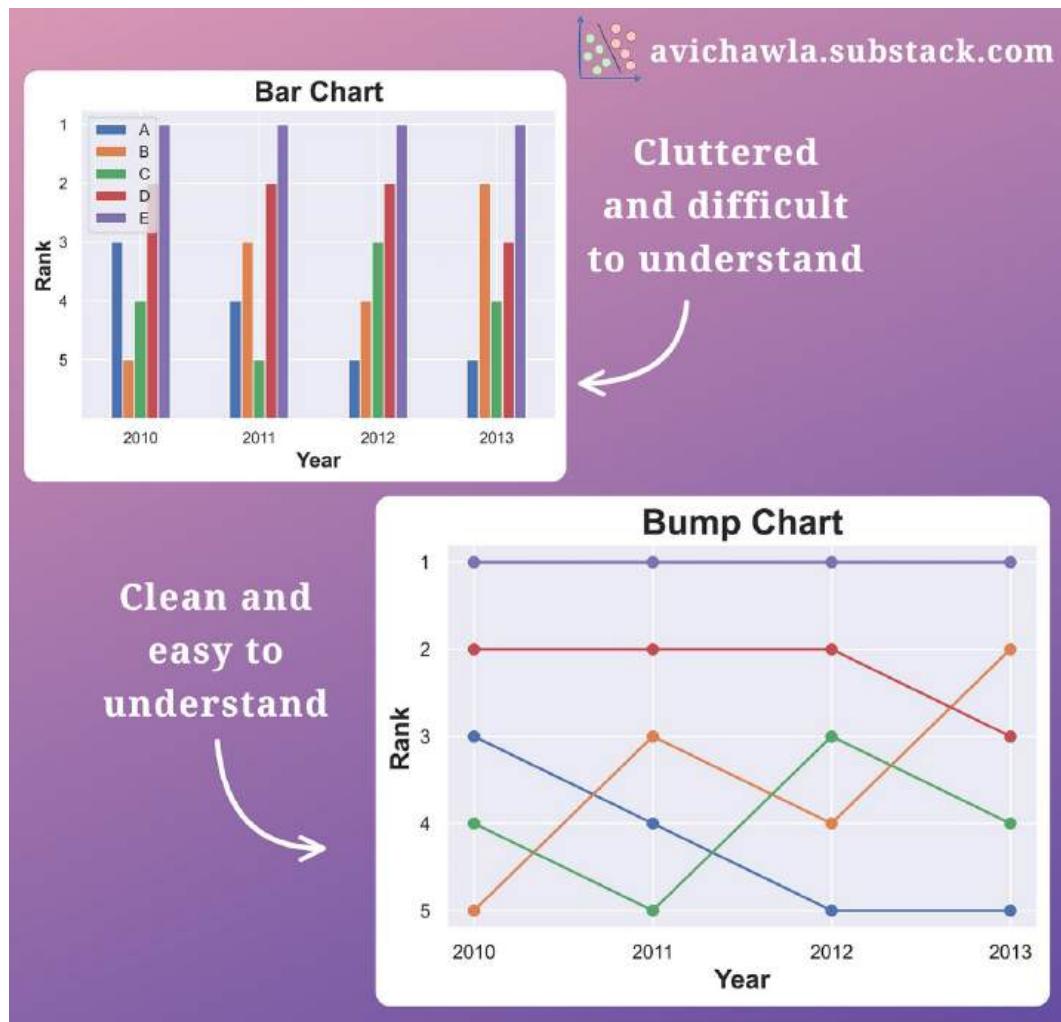
This means you can style your output in many different ways.

To style Pandas DataFrames, use its Styling API (**df.style**). As a result, the DataFrame is rendered with the specified styling.

Read more here: [Documentation](#).



# Visualise The Change In Rank Over Time With Bump Charts



When visualizing the change in rank over time, using a bar chart may not be appropriate. Instead, try Bump Charts.

They are specifically used to visualize the rank of different items over time.

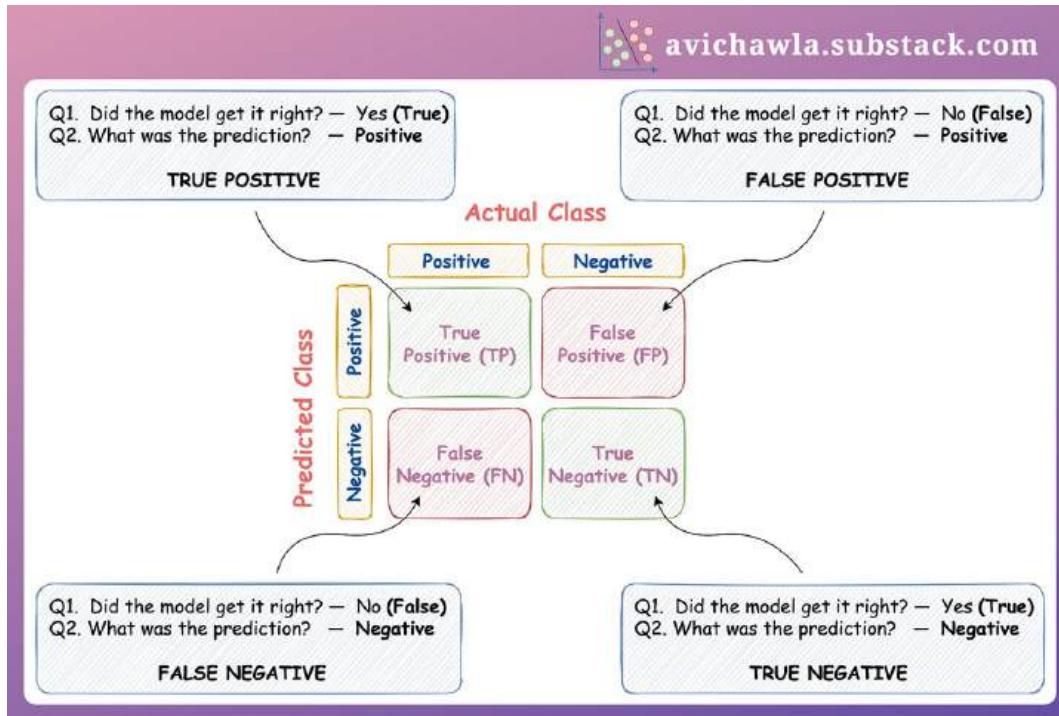
In contrast to the commonly used bar chart, they are clear, elegant, and easy to comprehend.

Over to you: What are some other bar chart alternatives to try in such cases? Let me know :)

Find the code for creating a bump chart in Python here: [Notebook](#).



# Use This Simple Technique To Never Struggle With TP, TN, FP and FN Again



Do you often struggle to label model predictions as TP, TN, FP and FN, and comprehend them? If yes, here's a simple guide to help you out.

When labeling any prediction, ask yourself two questions:

Did the model get it right? **Answer: yes (or True) / no (or False).**

What was the predicted class? **Answer: Positive/Negative.**

Next, combine the above two answers to get the final label.

For instance, say the actual and predicted class were positive.

Did the model get it right? The answer is **yes (or TRUE).**

What was the predicted class? The answer is **POSITIVE.**

Final label: **TRUE POSITIVE.**

As an exercise, try labeling the following predictions. Consider the “Cat” class as “Positive” and the “Dog” class as “Negative”.



True Class	Predicted Class	Did the model get it right?	What was the predicted class?
		-	-
		-	-
		-	-
		-	-



# The Most Common Misconception About Inplace Operations in Pandas



avichawla.substack.com

Method	Run-time	
	<i>inplace=False</i>	<i>inplace=True</i>
<code>df.replace()</code>	140 $\mu\text{s}$	244 $\mu\text{s}$ (Slow)
<code>df.sort_values()</code>	374 $\mu\text{s}$	450 $\mu\text{s}$ (Slow)
<code>df.reset_index()</code>	35 $\mu\text{s}$	10 $\mu\text{s}$ (Fast)
<code>df.drop()</code>	200 $\mu\text{s}$	262 $\mu\text{s}$ (Slow)
<code>df.fillna()</code>	90 $\mu\text{s}$	222 $\mu\text{s}$ (Slow)
<code>df.dropna()</code>	750 $\mu\text{s}$	1088 $\mu\text{s}$ (Slow)
<code>df.drop_duplicates()</code>	856 $\mu\text{s}$	1058 $\mu\text{s}$ (Slow)
<code>df.rename()</code>	151 $\mu\text{s}$	152 $\mu\text{s}$ (Equal)

Pandas users often modify a DataFrame inplace expecting better performance. Yet, it may not always be efficient. Here's why.

The image compares the run-time of inplace and non-in-place operations. In most cases, inplace operations are slow.

Why?

Contrary to common belief, most inplace operations DO NOT prevent the creation of a new copy. It is just that inplace assigns the copy back to the same address.



But during this assignment, Pandas performs some extra checks (SettingWithCopy) to ensure that the DataFrame is being modified correctly. This, at times, can be an expensive operation.

Yet, in general, there is no guarantee that an inplace operation is faster.

What's more, inplace operations do not allow chaining multiple operations, such as this:

The diagram illustrates two ways to perform a series of operations on a DataFrame. The top section, labeled "Method chaining", shows a single line of code: `df.reset_index().fillna(0).drop_duplicates()`. A curved arrow points from this code to the bottom section, which is labeled "No chaining with Inplace". The bottom section shows three separate lines of code, each with `inplace = True`: `df.reset_index(inplace = True)`, `df.fillna(0, inplace = True)`, and `df.drop_duplicates(inplace = True)`.

Method chaining

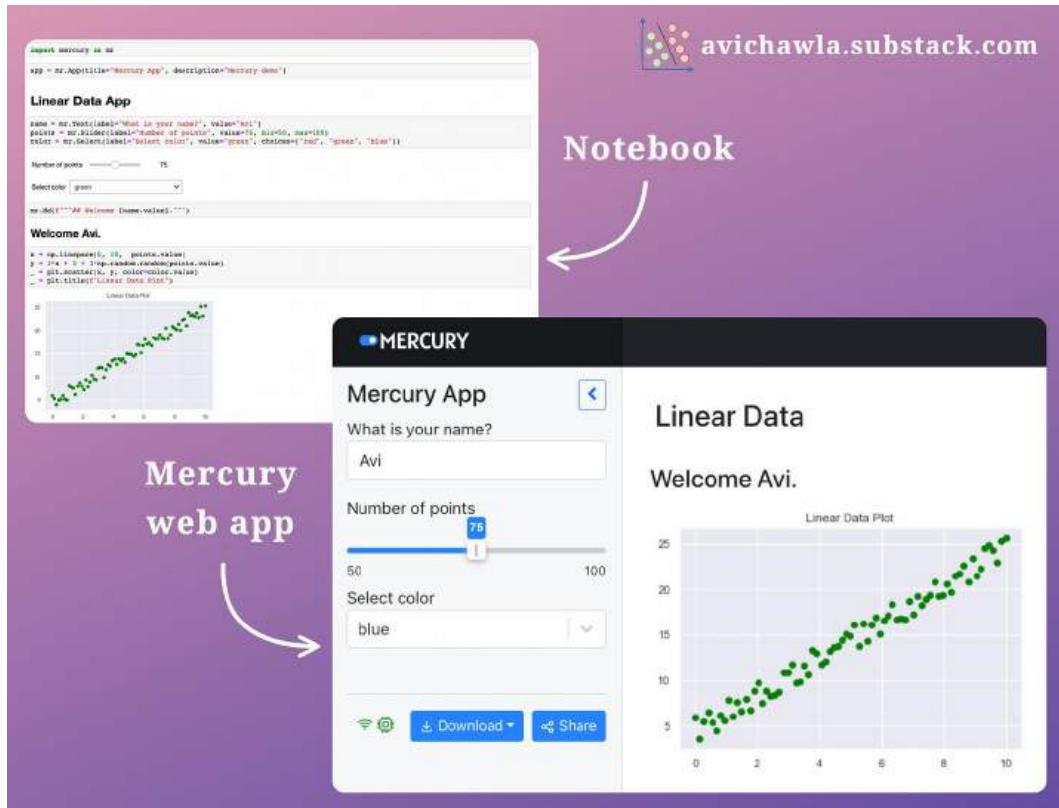
```
df.reset_index().fillna(0).drop_duplicates()
```

No chaining with Inplace

```
df.reset_index(inplace = True)
df.fillna(0, inplace = True)
df.drop_duplicates(inplace = True)
```



# Build Elegant Web Apps Right From Jupyter Notebook with Mercury



Exploring and sharing data insights using Jupyter is quite common for data folks. Yet, an interactive app is better for those who don't care about your code and are interested in your results.

While creating presentations is possible, it can be time-consuming. Also, one has to leave the comfort of a Jupyter Notebook.

Instead, try Mercury. It's an open-source tool that converts your jupyter notebook to a web app in no time. Thus, you can create the web app without leaving the notebook.

A quick demo is shown below:



The image shows a side-by-side comparison between a Jupyter Notebook and a Mercury application. On the left, the Jupyter Notebook interface displays Python code for generating a scatter plot. On the right, the Mercury application interface shows a live preview of the scatter plot titled "Linear Data Plot". The Mercury app includes input fields for the user's name ("Avi") and the number of points (set to 75), and a dropdown menu for selecting colors (set to "blue"). The plot itself shows a linear relationship with blue data points.

What's more, all updates to the Jupyter Notebook are instantly reflected in the Mercury app.

In contrast to the widely-adopted streamlit, web apps created with Mercury can be:

Exported as PDF/HTML.

Showcased as a live presentation.

Secured with authentication to restrict access.



# Become A Bilingual Data Scientist With These Pandas to SQL Translations



avichawla.substack.com

Operation	Pandas	SQL
Read CSV	<code>pd.read_csv(file)</code>	<code>LOAD DATA INFILE 'data.csv' INTO TABLE table FIELDS TERMINATED BY '' LINES TERMINATED BY '\n' IGNORE 1 ROWS;</code>
Print first 10 (or k) rows	<code>df.head(10)</code>	<code>SELECT * FROM table LIMIT 10;</code>
Dimensions	<code>df.shape</code>	<code>SELECT count(*) FROM table;</code>
Datatype	<code>df.dtypes</code>	<code>DESCRIBE table;</code>
Filter Data	<code>df[df.column&gt;10]</code>	<code>SELECT * FROM table where column&gt;10;</code>
Select column(s)	<code>df.column</code>	<code>SELECT column FROM table;</code>
Sort	<code>df.sort_values("column")</code>	<code>SELECT * FROM table ORDER BY column;</code>
Fill NaN	<code>df.column.fillna(0)</code>	<code>UPDATE table SET column=0 WHERE column IS NULL;</code>
Join	<code>pd.merge(df1, df2, on ="col", how = "inner")</code>	<code>SELECT * FROM table1 JOIN table2 ON (table1.col = table2.col);</code>
Concatenate	<code>pd.concat((df1, df2))</code>	<code>SELECT * FROM table1 UNION ALL table2;</code>
Group	<code>df.groupby("column"). agg_col.mean()</code>	<code>SELECT column, avg(agg_col) FROM table GROUP BY column;</code>
Unique values	<code>df.column.unique()</code>	<code>SELECT DISTINCT column FROM table;</code>
Rename column	<code>df.rename(columns = {"old_name": "new_name"})</code>	<code>ALTER TABLE table RENAME COLUMN old_name TO new_name;</code>
Delete column	<code>df.drop(columns = ["column"])</code>	<code>ALTER TABLE table DROP COLUMN column;</code>

SQL and Pandas are both powerful tools for data scientists to work with data.



Together, SQL and Pandas can be used to clean, transform, and analyze large datasets, and to create complex data pipelines and models.

Thus, proficiency in both frameworks can be extremely valuable to data scientists.

This visual depicts a few common operations in Pandas and their corresponding translations in SQL.

I have a detailed blog on Pandas to SQL translations with many more examples. Read it here: [Pandas to SQL blog](#).

Over to you: What other Pandas to SQL translations will you include here?



# A Lesser-Known Feature of Sklearn To Train Models on Large Datasets

The screenshot shows a Substack article with the title "avichawla.substack.com". The main content consists of two code snippets in a terminal window.

**Top Snippet:**

```
from sklearn.linear_model
import SGDClassifier

clf = SGDClassifier(...)

clf.fit(X, y)
```

**Text on the right:** Failed due to memory constraints

**Bottom Snippet:**

```
# 1. Load data in chunks (say, 1000 rows at once).
data = pd.read_csv("data.csv", chunksize = 1000)

# 2. Train from mini-batches using partial_fit.
for batch in data:
    clf.partial_fit(batch["X"],
                    batch["y"],
                    classes = [0, 1, 2])
```

It is difficult to train models with sklearn when you have plenty of data. This may often raise memory errors as the entire data is loaded in memory. But here's what can help.

Sklearn implements the **partial\_fit** API for various algorithms, which offers incremental learning.

As the name suggests, the model can learn incrementally from a mini-batch of instances. This prevents limited memory constraints as only a few instances are loaded in memory at once.



As shown in the main image, `clf.fit(X, y)` takes the entire data, and thus, it may raise memory errors. But, loading chunks of data and invoking the `clf.partial_fit()` method prevents this and offers seamless training.

Also, remember that while using the **partial\_fit** API, a mini-batch may not have instances of all classes (especially the first mini-batch). Thus, the model will be unable to cope with new/unseen classes in subsequent mini-batches. Therefore, you should pass a list of all possible classes in the `classes` parameter.

Having said that, it is also worth noting that not all sklearn estimators implement the **partial\_fit** API. Here's the list:

- Classification

- `sklearn.naive_bayes.MultinomialNB`
- `sklearn.naive_bayes.BernoulliNB`
- `sklearn.linear_model.Perceptron`
- `sklearn.linear_model.SGDClassifier`
- `sklearn.linear_model.PassiveAggressiveClassifier`

- Regression

- `sklearn.linear_model.SGDRegressor`
- `sklearn.linear_model.PassiveAggressiveRegressor`

- Clustering

- `sklearn.cluster.MiniBatchKMeans`

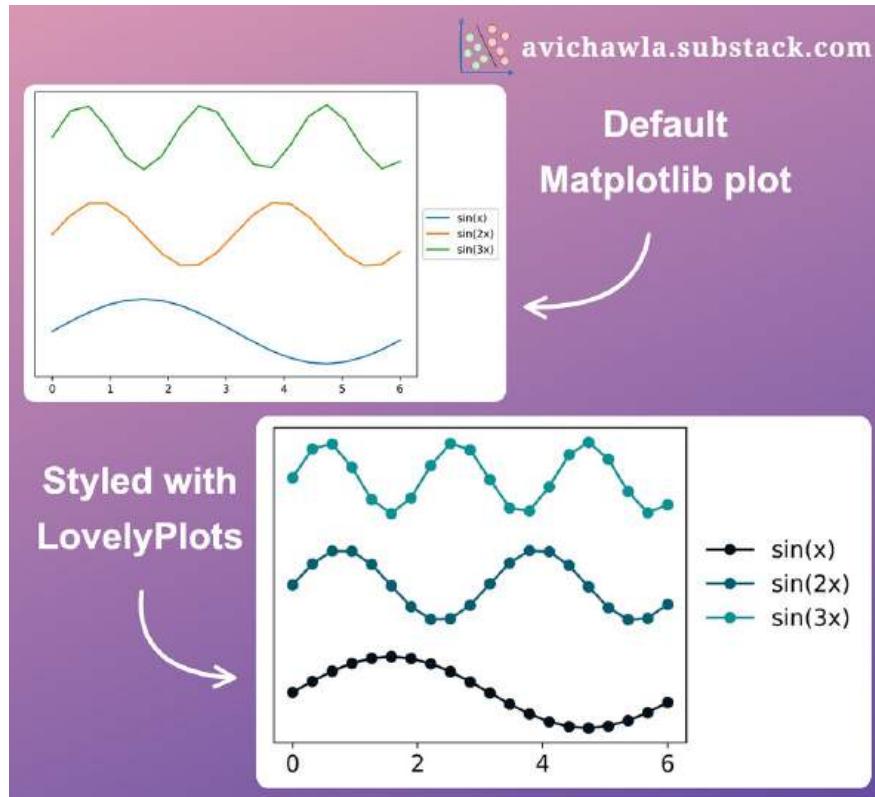
- Decomposition / feature Extraction

- `sklearn.decomposition.MiniBatchDictionaryLearning`
- `sklearn.cluster.MiniBatchKMeans`

Yet, it is surely worth exploring to see if you can benefit from it :)



# A Simple One-Liner to Create Professional Looking Matplotlib Plots



The default styling of matplotlib plots appears pretty basic at times. Here's how you can make them appealing.

To create professional-looking plots for presentations, reports, or scientific papers, try LovelyPlots.

It provides many style sheets to improve their default appearance, by simply adding just one line of code.

To install LovelyPlots, run the following command:

```
pip install LovelyPlots
```

Next, import the matplotlib library, and change the style as follows: (You don't have to import LovelyPlots anywhere)

```
import matplotlib.pyplot as plt  
plt.style.use(style) ## change to the style provided by  
LovelyPlots
```

Print the list of all possible styles as follows:

```
plt.style.available
```

Get Started: [LovelyPlots Repository](#).



[blog.DailyDoseofDS.com](http://blog.DailyDoseofDS.com)



# Avoid This Costly Mistake When Indexing A DataFrame

df.shape

(32768000, 9)

avichawla.substack.com

## First column then row

```
%timeit df["col"]["row"]
```

2.96  $\mu$ s ± 7.17 ns per loop

## First row then column

```
%timeit df.loc["row"]["col"]
```

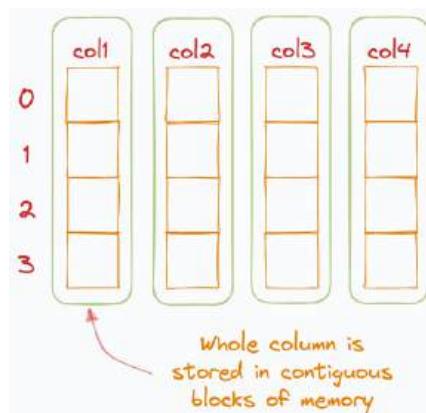
45.4  $\mu$ s ± 384 ns per loop

Selecting a column first is over 15x faster

When indexing a dataframe, choosing whether to select a column first or slice a row first is pretty important from a run-time perspective.

As shown above, selecting the column first is over **15 times** faster than slicing the row first. Why?

As I have talked before, Pandas DataFrame is a column-major data structure. Thus, consecutive elements in a column are stored next to each other in memory.

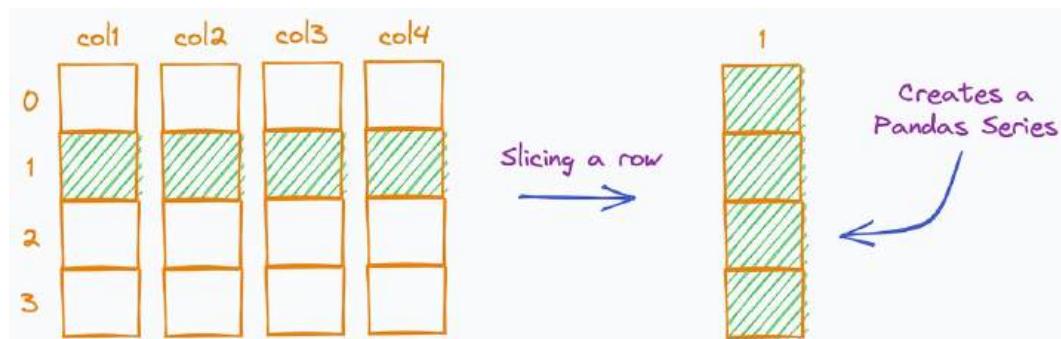




As processors are efficient with contiguous blocks of memory, accessing a column is much faster than accessing a row (read more about this in one of my previous posts [here](#)).

But when you slice a row first, each row is retrieved by accessing non-contiguous blocks of memory, thereby making it slow.

Also, once all the elements of a row are gathered, Pandas converts them to a Series, which is another overhead.



We can verify this conversion below:

A screenshot of a Jupyter Notebook cell. The code shows the creation of a DataFrame 'df' with two columns 'A' and 'B', and two rows containing values 1, 2 and 3, 4 respectively. Then, the command `df.loc[0]` is run, which returns a Series object for the first row. The output shows the values 1 and 2, with the note "Name: 0, dtype: int64". Finally, the type of the result is checked with `type(df.loc[0])`, which outputs "pandas.core.series.Series". A pink box highlights the word "Series" in the output, and a white arrow points from the text "Creates a Pandas Series" to this highlighted word.

```
>>> df
   A  B
0  1  2
1  3  4

>>> df.loc[0]
A    1
B    2
Name: 0, dtype: int64

>>> type(df.loc[0])
pandas.core.series.Series
```

Series object

Instead, when you select a column first, elements are retrieved by accessing contiguous blocks of memory, which is way faster. Also, a column is inherently a Pandas Series. Thus, there is no conversion overhead involved like above.



The screenshot shows a Jupyter Notebook cell with the following code and output:

```
>>> df
      A   B
0   1   2
1   3   4

>>> df["A"]
0    1
1    3
Name: A, dtype: int64

>>> type(df["A"])
pandas.core.series.Series
```

A pink callout box labeled "Series object" points to the last line of the output, which is the type of the selected column.

Overall, by accessing the column first, we avoid accessing non-contiguous memory access, which does happen when we access the row first.

This makes selecting the column first faster than slicing a row first in indexing operations.

If you are confused about what selecting, indexing, slicing, and filtering mean, here's what you should read next:

<https://avichawla.substack.com/p/are-you-sure-you-are-using-the-correct>.



# 9 Command Line Flags To Run Python Scripts More Flexibly

## Python Command Line Flags

Python Flag	Description	Usage
<code>python -c</code>	Run a single python command	<code>python -c "print('Hello')"</code>
<code>python -i</code>	Run interactive Python shell after running a script	<code>python -i script.py</code>
<code>python -O</code>	Ignore assert statements	<code>python -O script.py</code>
<code>python -OO</code>	Ignore assert statements and docstrings	<code>python -OO script.py</code>
<code>python -W</code>	Ignore warnings	<code>python -W script.py</code>
<code>python -m</code>	Run a module as a script	<code>python -m my_package.my_module</code>
<code>python -v</code>	Enable verbose mode. Prints more information about what interpreter is doing	<code>python -v script.py</code>
<code>python -x</code>	Ignore the first line of the script (often the shebang line)	<code>python -x script.py</code>
<code>python -E</code>	Ignore all Python Environment variables	<code>python -E script.py</code>



[avichawla.substack.com](http://avichawla.substack.com)

When invoking a Python script, you can specify various options/flags. They are used to modify the behavior of the Python interpreter when it runs a script or module.

Here are 9 of the most commonly used options:

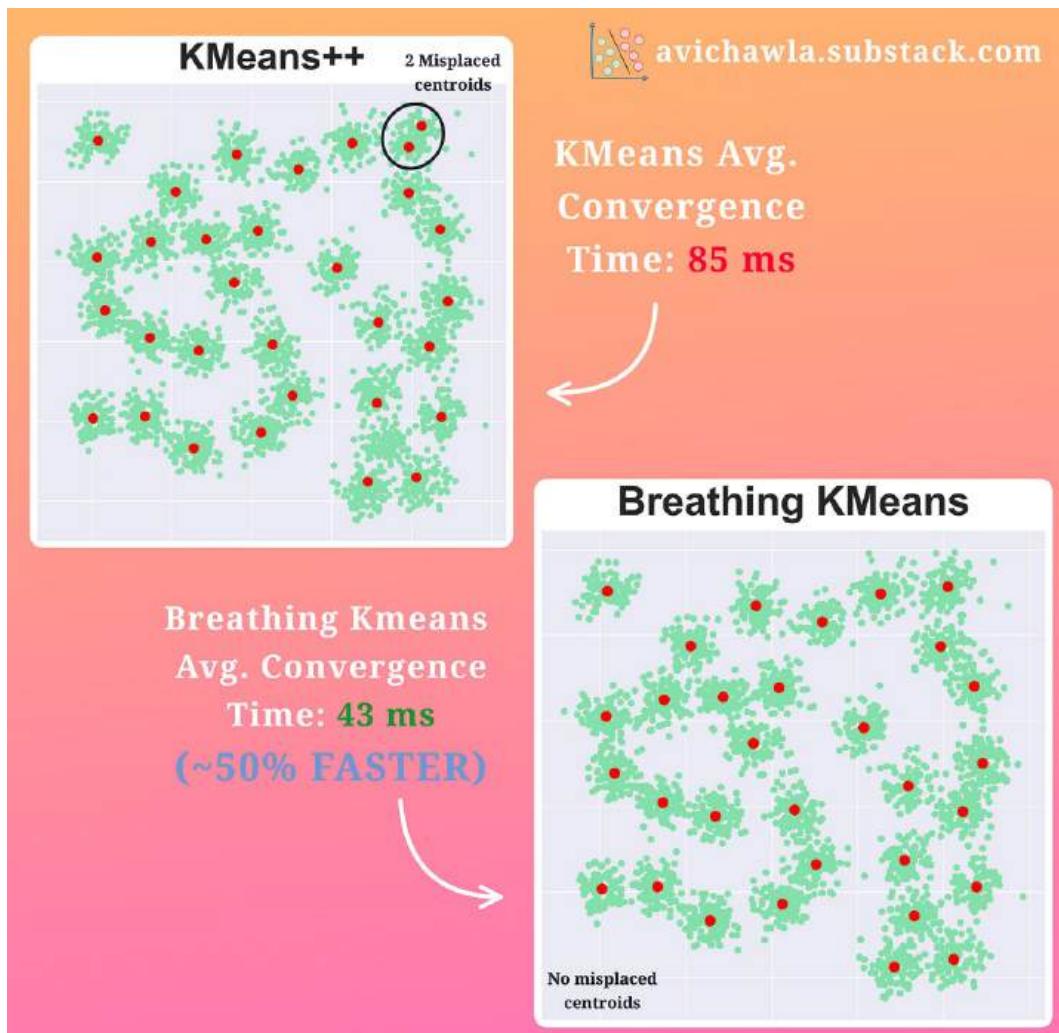


- ◆ **python -c**: Run a single Python command. Useful for running simple one-liners or testing code snippets.
- ◆ **python -i**: Run the script as usual and enter the interactive mode instead of terminating the program. Useful for debugging as you can interact with objects created during the program.
- ◆ **python -O**: Ignore assert statements (This is alphabet ‘O’). Useful for optimizing code by removing debugging code.
- ◆ **python -OO**: Ignore assert statements and discard docstrings. Useful for further optimizing code by removing documentation strings.
- ◆ **python -W**: Ignore all warnings. Useful for turning off warnings temporarily and focusing on development.
- ◆ **python -m**: Run a module as a script.
- ◆ **python -v**: Enter verbose mode. Useful for printing extra information during program execution.
- ◆ **python -x**: Skip the first line. Useful for removing shebang lines or other comments at the start of a script.
- ◆ **python -E**: ignore all Python environment variables. Useful for ensuring a consistent program behavior by ignoring environment variables that may affect program execution.

Which ones have I missed? Let me know :)



# Breathing KMeans: A Better and Faster Alternative to KMeans



The performance of KMeans is entirely dependent on the centroid initialization step. Thus, obtaining inaccurate clusters is highly likely.

While KMeans++ offers smarter centroid initialization, it does not always guarantee accurate convergence (read how KMeans++ works in my [previous post](#)). This is especially true when the number of clusters is high. Here, repeating the algorithm may help. But it introduces an unnecessary overhead in run-time.

Instead, Breathing KMeans is a better alternative here. Here's how it works:

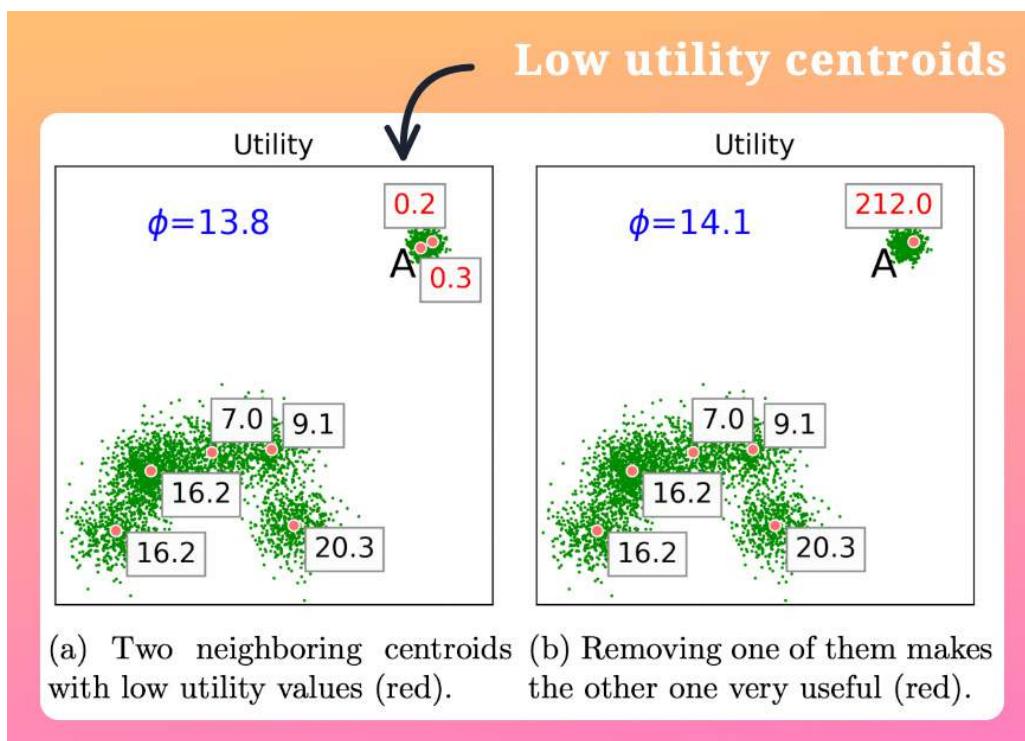
- **Step 1:** Initialise  $k$  centroids and run KMeans without repeating. In other words, don't re-run it with different initializations. Just run it once.



- **Step 2 — Breathe in step:** Add  $m$  new centroids and run KMeans with  $(k+m)$  centroids without repeating.
- **Step 3 — Breathe out step:** Remove  $m$  centroids from existing  $(k+m)$  centroids. Run KMeans with the remaining  $k$  centroids without repeating.
- **Step 4:** Decrease  $m$  by 1.
- **Step 5:** Repeat Steps 2 to 4 until  $m=0$ .

**Breathe in** step inserts new centroids close to the centroids with the largest errors. A centroid's error is the sum of the squared distance of points under that centroid.

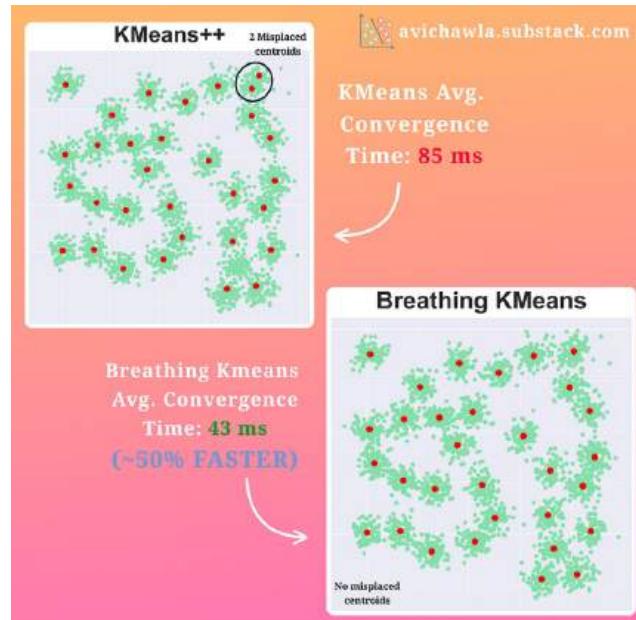
**Breathe out** step removes centroids with low utility. A centroid's utility is proportional to its distance from other centroids. The intuition is that if two centroids are pretty close, they are likely falling in the same cluster. Thus, both will be assigned a low utility value, as demonstrated below.



With these repeated breathing cycles, Breathing KMeans provides a faster and better solution than KMeans. In each cycle, new centroids are added at “good” locations, and centroids with low utility are removed.



In the figure below, KMeans++ produced two misplaced centroids.



However, Breathing KMeans accurately clustered the data, with a 50% improvement in run-time.

You can use Breathing KMeans by installing its open-source library, **bkmeans**, as follows:

```
pip install bkmeans
```

Next, import the library and run the clustering algorithm:

```
import numpy as np
from bkmeans import BKMeans

# generate random data set
X=np.random.rand(1000,2)

# create BKMeans instance
bkm = BKMeans(n_clusters=100)

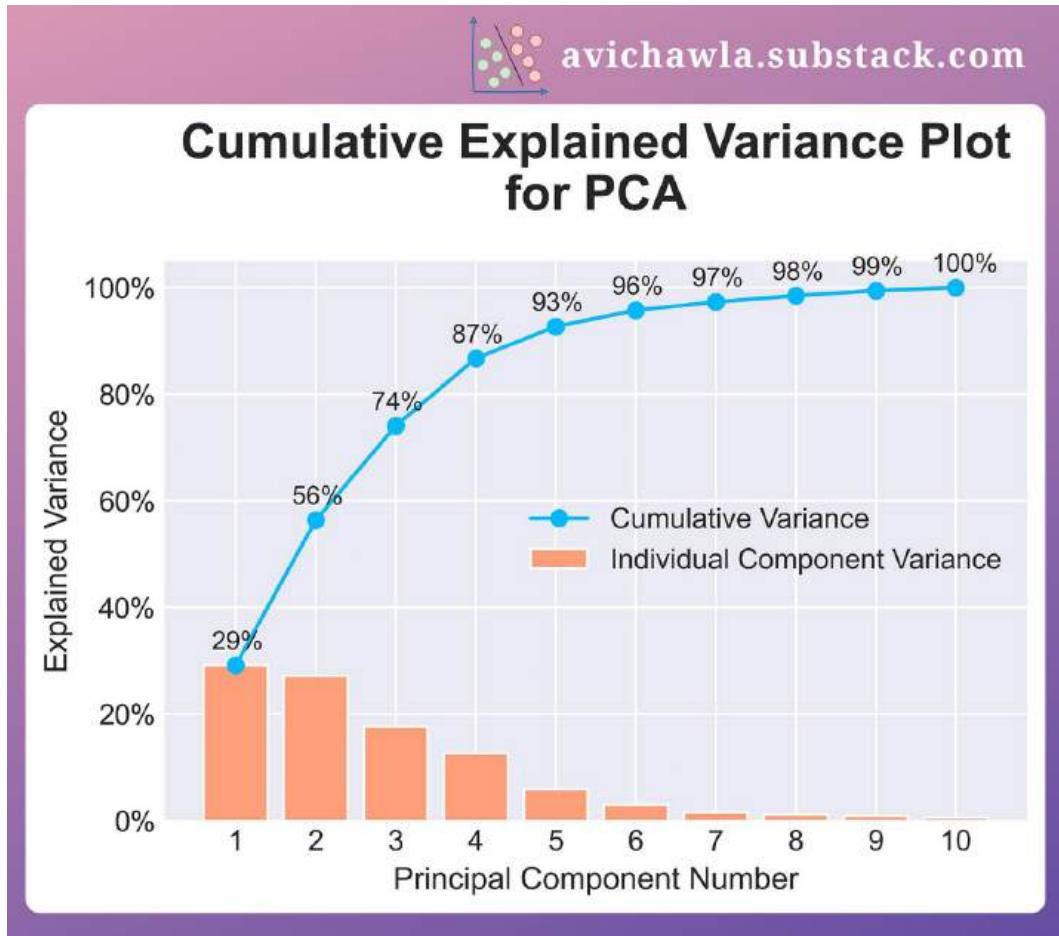
# run the algorithm
bkm.fit(X)
```

In fact, the `BKMeans` class inherits from the `KMeans` class of `sklearn`. So you can specify other parameters and use any of the other methods on the `BKMeans` object as needed.

More details about Breathing KMeans: [GitHub](#) | [Paper](#).



# How Many Dimensions Should You Reduce Your Data To When Using PCA?

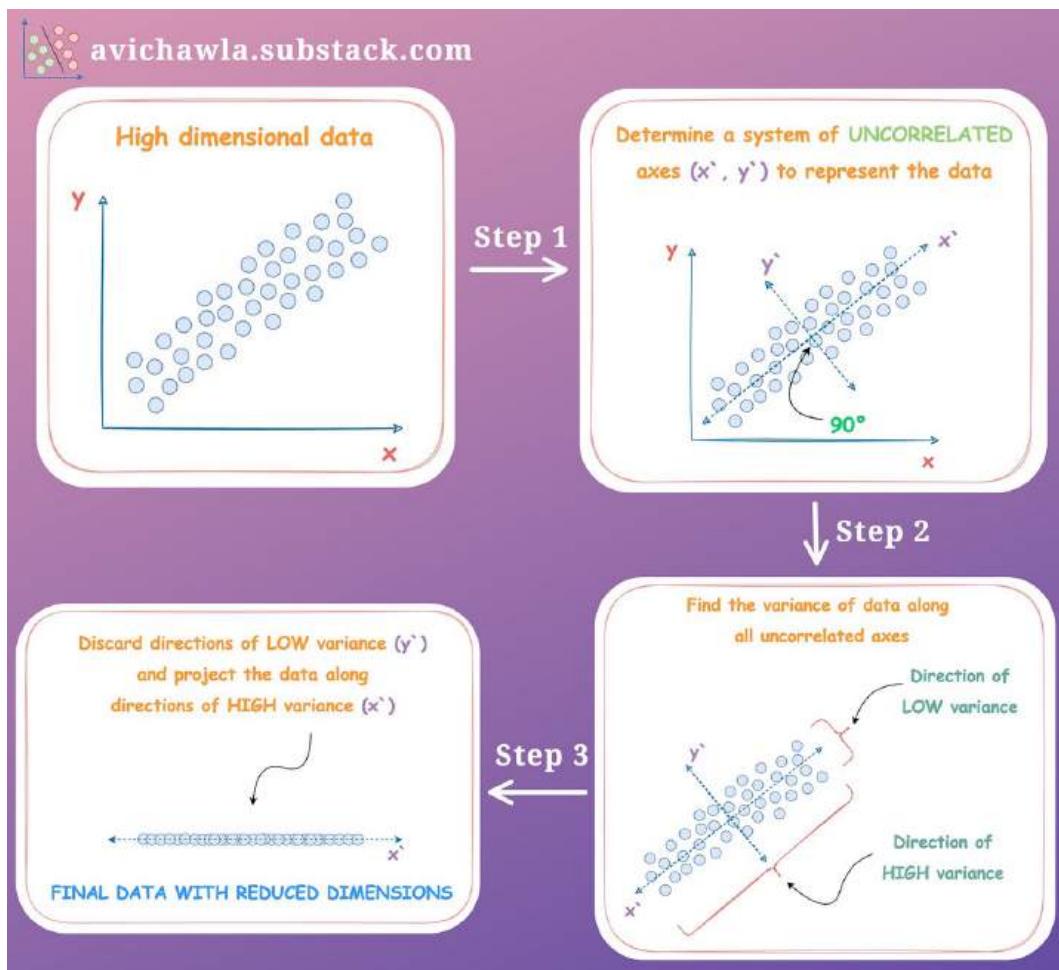


When using PCA, it can be difficult to determine the number of components to keep. Yet, here's a plot that can immensely help.

---

Note: If you don't know how PCA works, feel free to read my detailed post: [A Visual Guide to PCA](#).

Still, here's a quick step-by-step refresher. Feel free to skip this part if you remember my PCA post.



**Step 1.** Take a high-dimensional dataset ( $(\mathbf{x}, \mathbf{y})$  in the above figure) and represent it with uncorrelated axes ( $(\mathbf{x}', \mathbf{y}')$  in the above figure). Why uncorrelated?

This is to ensure that data has zero correlation along its dimensions and each new dimension represents its individual variance.

For instance, as data represented along  $(\mathbf{x}, \mathbf{y})$  is correlated, the variance along  $\mathbf{x}$  is influenced by the spread of data along  $\mathbf{y}$ .

Instead, if we represent data along  $(\mathbf{x}', \mathbf{y}')$ , the variance along  $\mathbf{x}'$  is not influenced by the spread of data along  $\mathbf{y}'$ .

The above space is determined using eigenvectors.

**Step 2.** Find the variance along all uncorrelated axes  $(\mathbf{x}', \mathbf{y}')$ . The eigenvalue corresponding to each eigenvector denotes the variance.

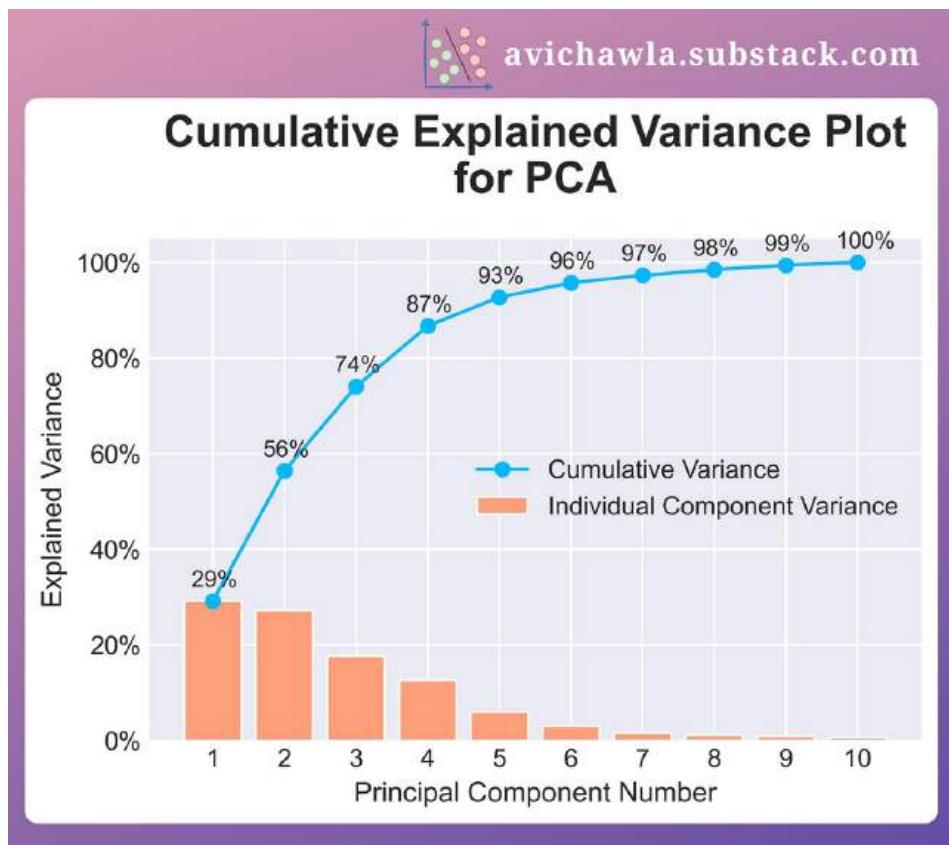
**Step 3.** Discard the axes with low variance. How many dimensions to discard (or keep) is a hyperparameter, which we will discuss below. Project the data along the retained axes.



When reducing dimensions, the purpose is to retain enough variance of the original data.

As each principal component explains some amount of variance, cumulatively plotting the component-wise variance can help identify which components have the most variance.

This is called a cumulative explained variance plot.



For instance, say we intend to retain **~85%** of the data variance. The above plot clearly depicts that reducing the data to four components will do that.

Also, as expected, all ten components together represent 100% variance of the data.

Creating this plot is pretty simple in Python. **Find the code here: [PCA-CEV Plot](#).**



# 🚀 Mito Just Got Supercharged With AI!

The screenshot shows the Mito spreadsheet interface. At the top, there's a code editor window with the following Python code:

```
In [1]: import mitosheet  
mitosheet.sheet(analyses_to_replay="id-utbdzhhmvd")
```

Below the code editor is a toolbar with various icons for file operations like Undo, Redo, Import, Export, and a prominent AI icon highlighted with a green arrow. The main area is a data grid titled "City | City" containing 14 rows of employee data. The columns are: Name, Company, City, Salary, Status, and Rating. The data includes names like Johnney Maynard, Michael Williams, and Linda Rodriguez, along with their respective company names and city locations. The bottom right corner of the interface indicates "(100 rows, 6 cols)".

Personally, I am a big fan of no-code data analysis tools. They are extremely useful in eliminating repetitive code across projects—thereby boosting productivity.

Yet, most no-code tools are often limited in terms of the functionality they support. Thus, flexibility is usually a big challenge while using them.

Mito is an incredible open-source tool that allows you to analyze your data within a spreadsheet interface in Jupyter without writing any code.

What's more, Mito recently supercharged its spreadsheet interface with AI. As a result, you can now analyze data in a notebook with text prompts.

One of the coolest things about using Mito is that each edit in the spreadsheet automatically generates an equivalent Python code. This makes it convenient to reproduce the analysis later.



## Automatic code generation

```
from mitosheet.public.v3 import *; register_analysis("id-utbdzhhmhd");
import pandas as pd

# Imported employee_dataset.csv
employee_dataset = pd.read_csv(r'employee_dataset.csv')

# group on city and find avg salary and rating
df2 = employee_dataset.groupby('City').agg({'Salary': 'mean', 'Rating': 'mean'})

# top 5 employees with highest salary
top_employees = employee_dataset.nlargest(5, 'Salary')
```

You can install Mito using pip as follows:

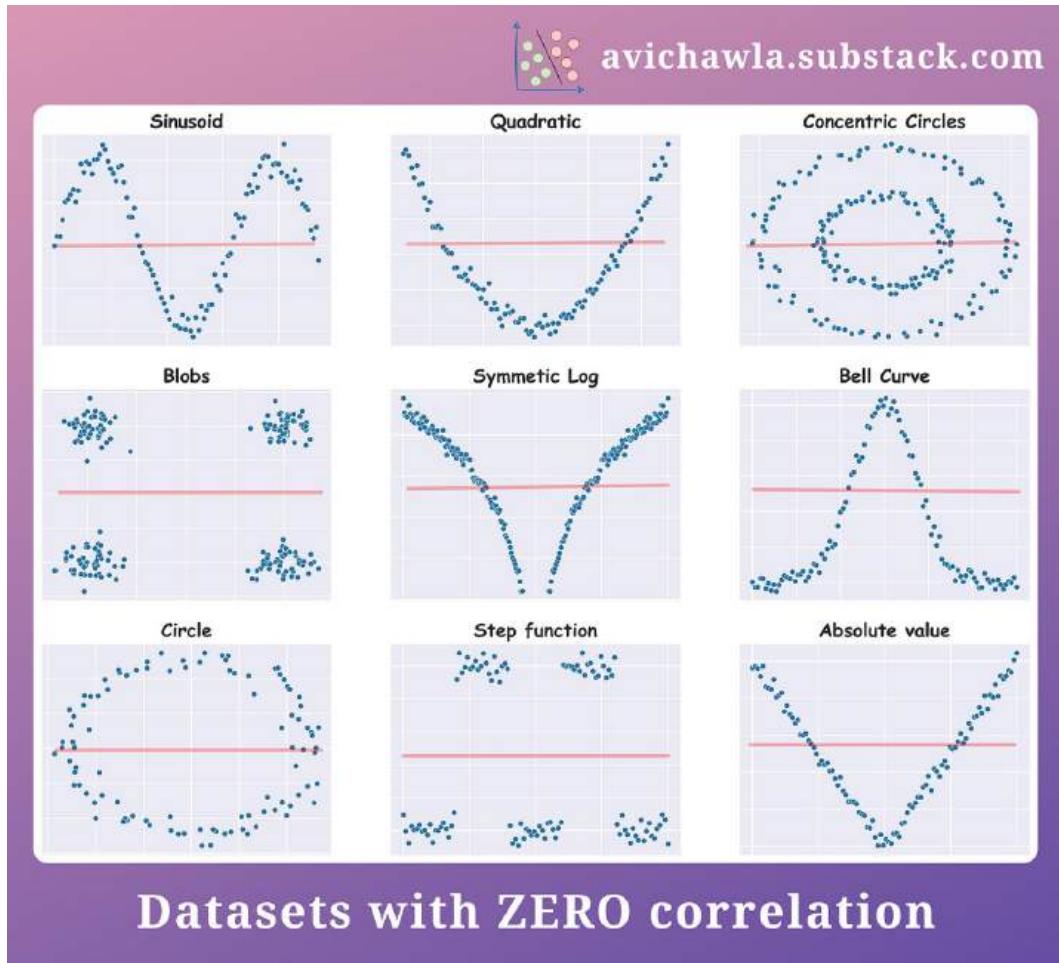
```
python -m pip install mitosheet
```

Next, to activate it in Jupyter, run the following two commands:

```
python -m jupyter nbextension install --py --user mitosheet
python -m jupyter nbextension enable --py --user mitosheet
```



# Be Cautious Before Drawing Any Conclusions Using Summary Statistics



While analyzing data, one may be tempted to draw conclusions solely based on its statistics. Yet, the actual data might be conveying a totally different story.

Here's a visual depicting nine datasets with approx. zero correlation between the two variables. But the summary statistic (Pearson correlation in this case) gives no clue about what's inside the data.

What's more, data statistics could be heavily driven by outliers or other artifacts. I covered this in a previous post [here](#).

Thus, the importance of looking at the data cannot be stressed enough. It saves you from drawing wrong conclusions, which you could have made otherwise by looking at the statistics alone.

For instance, in the sinusoidal dataset above, Pearson correlation may make you believe that there is no association between the two variables. However, remember that it is only quantifying the extent of



a linear relationship between them. Read more about this in another one of my previous posts [here](#).

Thus, if there's any other non-linear relationship (quadratic, sinusoid, exponential, etc.), it will fail to measure that.



# Use Custom Python Objects In A Boolean Context

```
without_bool.py
```

```
class Cart:  
    def __init__(self):  
        self.items = []  
  
# No __bool__ method  
  
my_cart = Cart()  
  
if my_cart:  
    print("Cart Not Empty")  
else:  
    print("Cart Empty")  
  
"Cart Not Empty" # Output
```

Object of custom class evaluated to True by default

```
with_bool.py
```

```
class Cart:  
    def __init__(self):  
        self.items = []  
  
    def __bool__(self):  
        return len(self.items)>0  
  
my_cart = Cart()  
  
if my_cart:  
    print("Cart Not Empty")  
else:  
    print("Cart Empty")  
  
"Cart Empty" # Output
```

Object evaluated to False

In a boolean context, Python always evaluates the objects of a custom class to True. But this may not be desired in all cases. Here's how you can override this behavior.

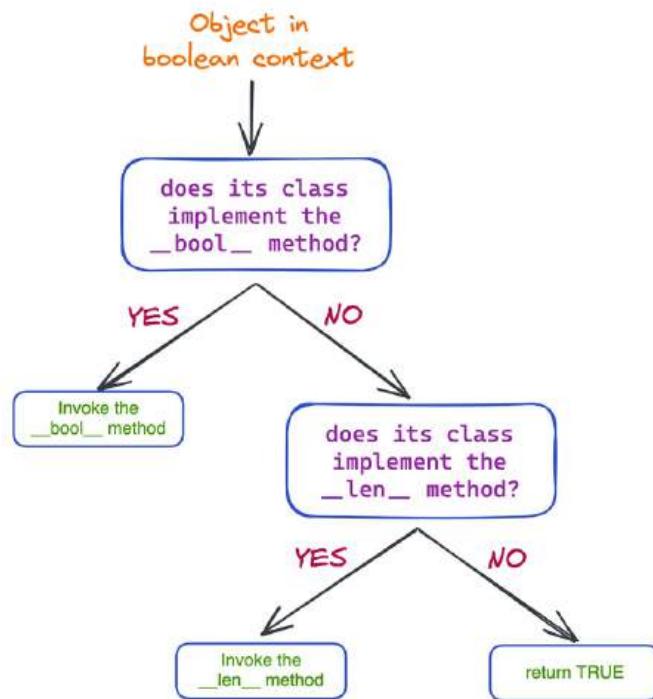
The `__bool__` dunder method is used to define the behavior of an object when used in a boolean context. As a result, you can specify explicit conditions to determine the truthiness of an object.

This allows you to use class objects in a more flexible and intuitive way.

As demonstrated above, without the `__bool__` method (*without\_bool.py*), the object evaluates to True. But implementing the `__bool__` method lets us override this default behavior (*with\_bool.py*).

## Some additional good-to-know details

When we use ANY object (be it instantiated from a custom or an in-built class) in a boolean context, here's what Python does:

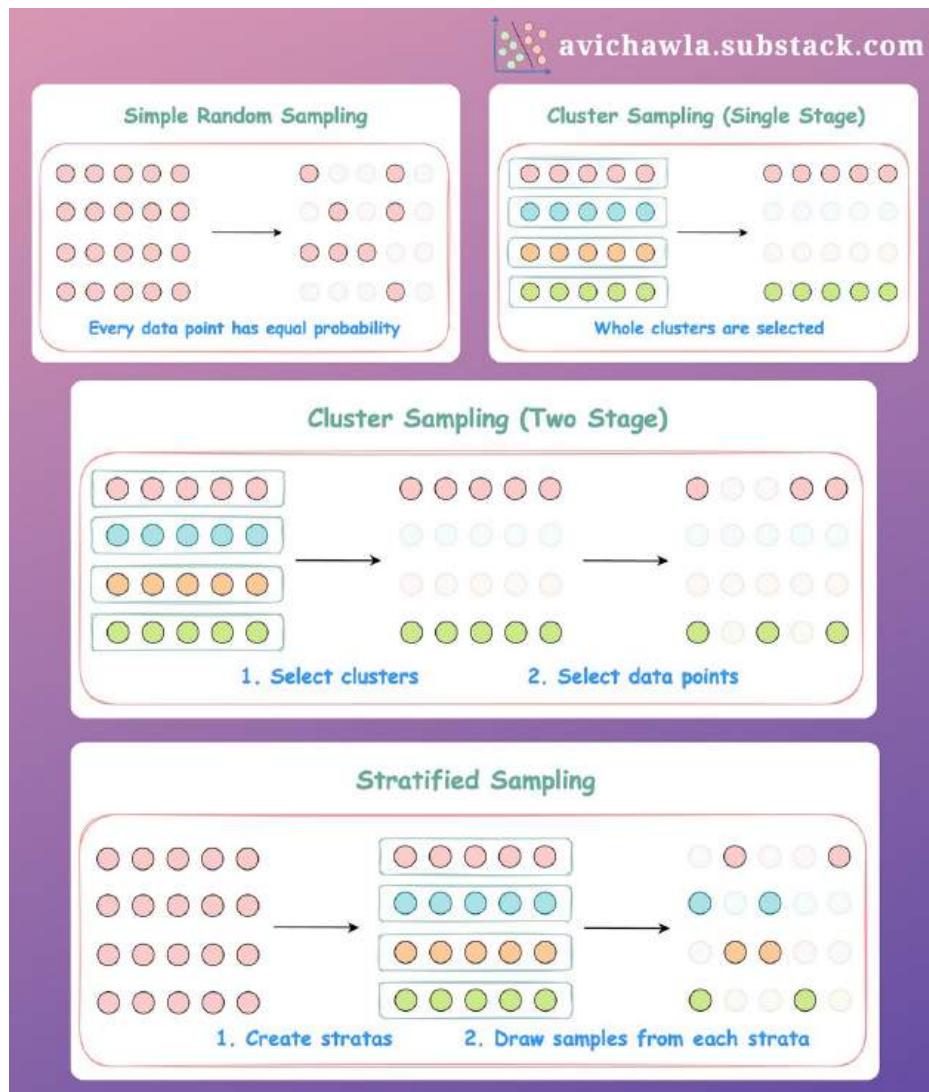


First, Python checks for the `__bool__` method in its class implementation. If found, it is invoked. If not, Python checks for the `__len__` method. If found, `__len__` is invoked. Otherwise, Python returns True.

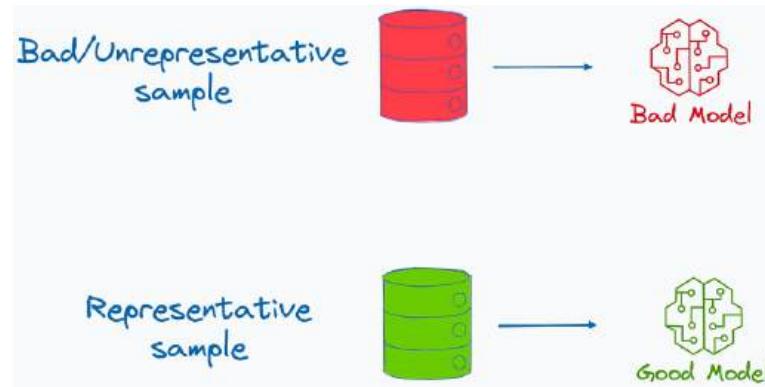
This explains the default behavior of objects instantiated from a custom class. As the `Cart` class implemented neither the `__bool__` method nor the `__len__` method, the `cart` object was evaluated to True.



# A Visual Guide To Sampling Techniques in Machine Learning



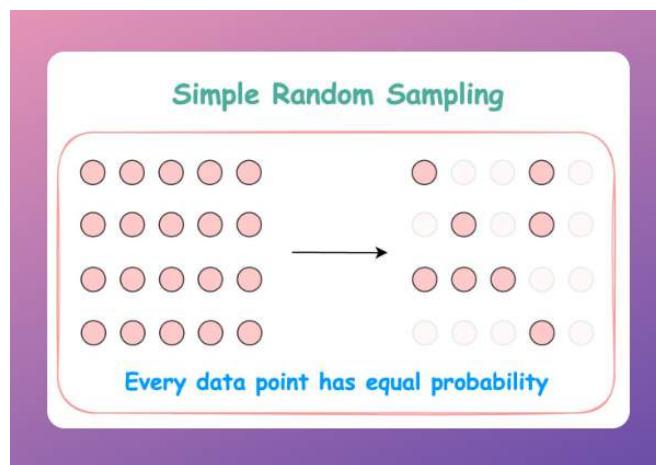
When you are dealing with large amounts of data, it is often preferred to draw a relatively smaller sample and train a model. But any mistakes can adversely affect the accuracy of your model.



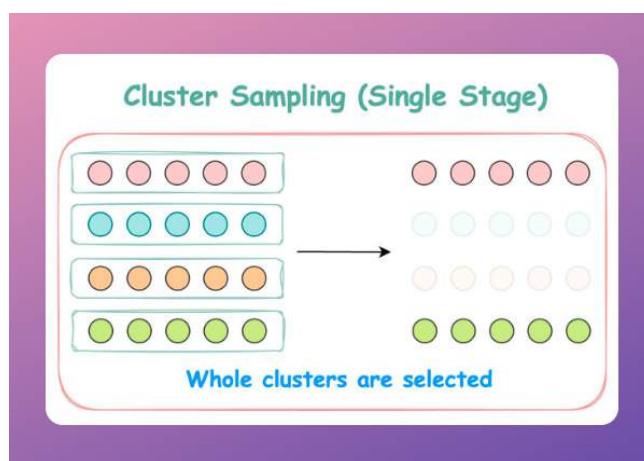
This makes sampling a critical aspect of training ML models.

Here are a few popularly used techniques that one should know about:

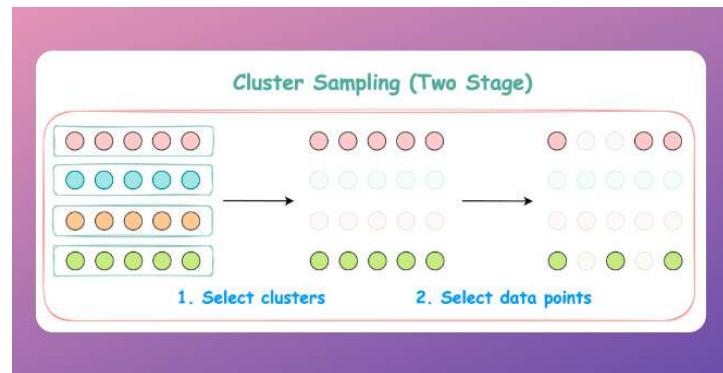
- ◆ **Simple random sampling:** Every data point has an equal probability of being selected in the sample.



- ◆ **Cluster sampling (single-stage):** Divide the data into clusters and select a few entire clusters.

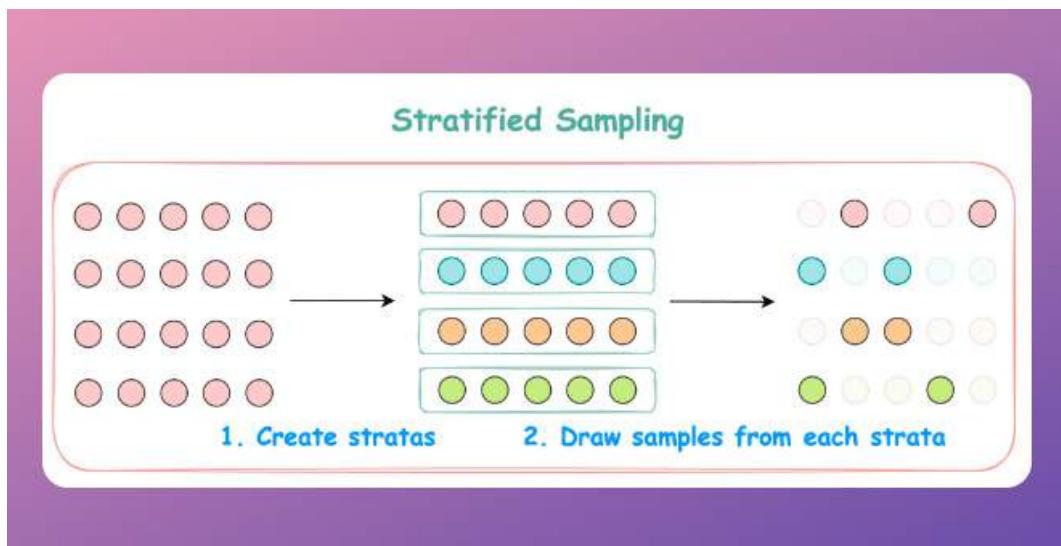


- ◆ **Cluster sampling (two-stage):** Divide the data into clusters, select a few clusters, and choose points from them randomly.





- ◆ **Stratified sampling:** Divide the data points into homogenous groups (based on age, gender, etc.), and select points randomly.



What are some other sampling techniques that you commonly resort to?



# You Were Probably Given Incomplete Info About A Tuple's Immutability

The screenshot shows a Python terminal window with the following code:

```
>>> my_tuple = (1, [2, 3])  
  
>>> my_tuple  
(1, [2, 3])  
  
>>> my_tuple[1].append(4) # No Error  
  
>>> my_tuple  
(1, [2, 3, 4])
```

A white curly brace on the left side of the terminal indicates that the entire tuple assignment is being referred to. An annotation on the right side of the terminal reads "Tuple Modified" with an arrow pointing to the last line of code.

 [avichawla.substack.com](http://avichawla.substack.com)

When we say tuples are immutable, many Python programmers think that the values inside a tuple cannot change. But this is not true.

The immutability of a tuple is solely restricted to the identity of objects it holds, not their value.

In other words, say a tuple has two objects with IDs **1** and **2**. Immutability says that the collection of IDs referenced by the tuple (and their order) can never change.

Yet, there is **NO** such restriction that the individual objects with IDs **1** and **2** cannot be modified.

Thus, if the elements inside the tuple are mutable objects, you can indeed modify them.

And as long as the collection of IDs remains the same, the immutability of a tuple is not violated.



This explains the demonstration above. As `append` is an inplace operation, the collection of IDs didn't change. Thus, Python didn't raise an error.

We can also verify this by printing the collection of object IDs referenced inside the tuple before and after the `append` operation:

The screenshot shows a Jupyter Notebook cell with the following code:

```
>>> my_tuple = (1, [2, 3])  
  
>>> id(my_tuple[0]), id(my_tuple[1])  
(583145, 434810)  
  
>>> my_tuple[1].append(4)  
  
>>> id(my_tuple[0]), id(my_tuple[1])  
(583145, 434810)
```

A red curly brace on the left side of the code block groups the first two lines, with the text "Same IDs" written vertically next to it. The URL [avichawla.substack.com](http://avichawla.substack.com) is visible at the bottom right of the slide.

As shown above, the IDs pre and post append are the same. Thus, immutability isn't violated.



# A Simple Trick That Significantly Improves The Quality of Matplotlib Plots



Matplotlib plots often appear dull and blurry, especially when scaled or zoomed. Yet, here's a simple trick to significantly improve their quality.

Matplotlib plots are rendered as an image by default. Thus, any scaling/zooming drastically distorts their quality.

Instead, always render your plot as a scalable vector graphic (SVG). As the name suggests, they can be scaled without compromising the plot's quality.

As demonstrated in the image above, the plot rendered as SVG clearly outshines and is noticeably sharper than the default plot.



The following code lets you change the render format to SVG. If the difference is not apparent in the image above, I would recommend trying it yourself and noticing the difference.

```
from matplotlib_inline.backend_inline import set_matplotlib_formats  
set_matplotlib_formats('svg')
```

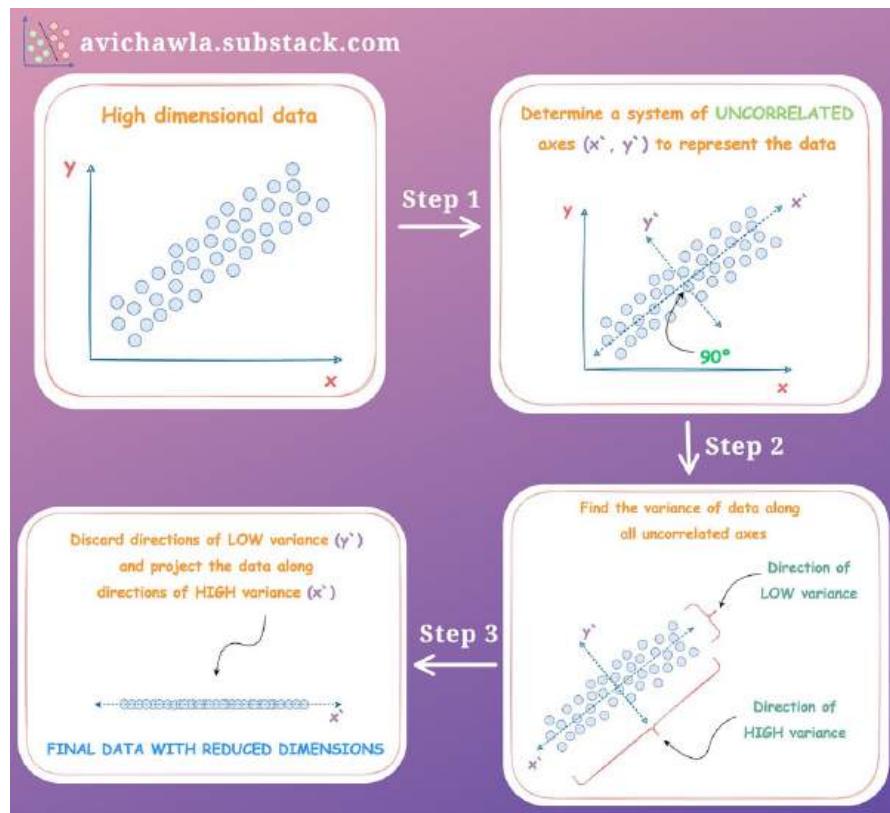
Alternatively, you can also use the following code:

```
%config InlineBackend.figure_format = 'svg'
```

P.S. If there's a chance that you don't know what is being depicted in the bar plot above, check out this [YouTube video by Numberphile](#).



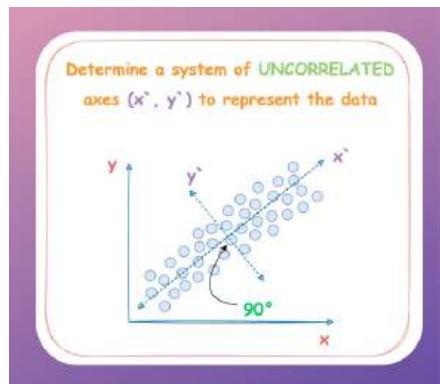
# A Visual and Overly Simplified Guide to PCA



Many folks often struggle to understand the core essence of principal component analysis (PCA), which is widely used for dimensionality reduction. Here's a simplified visual guide depicting what goes under the hood.

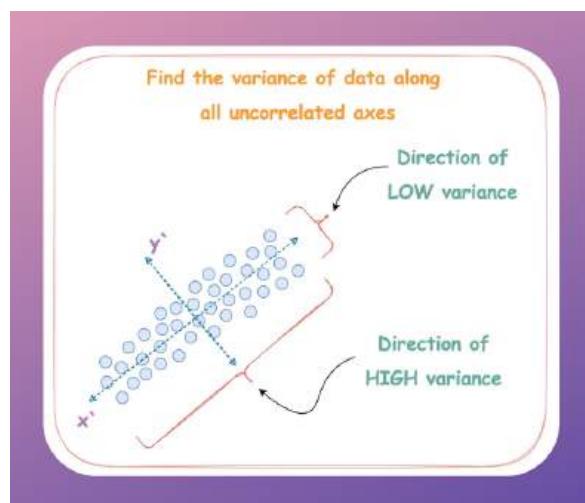
In a gist, while reducing the dimensions, the aim is to retain as much variation in data as possible.

To begin with, as the data may have correlated features, the first step is to determine a new coordinate system with orthogonal axes. This is a space where all dimensions are uncorrelated.



The above space is determined using the data's eigenvectors.

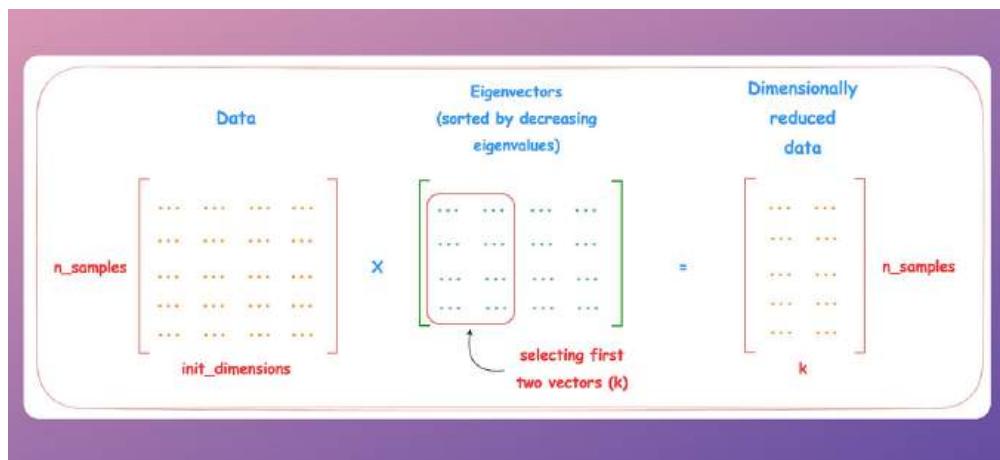
Next, we find the variance of our data along these uncorrelated axes. The variance is represented by the corresponding eigenvalues.



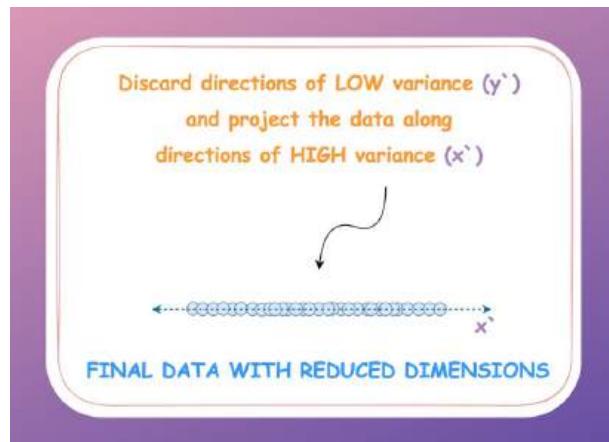
Next, we decide the number of dimensions we want our data to have post-reduction (a hyperparameter), say two. As our aim is to retain as much variance as possible, we select two eigenvectors with the highest eigenvalues.

Why highest, you may ask? As mentioned above, the variance along an eigenvector is represented by its eigenvalue. Thus, selecting the top two eigenvalues ensures we retain the maximum variance of the overall data.

Lastly, the data is transformed using a simple matrix multiplication with the top two vectors, as shown below:



After reducing the dimension of the 2D dataset used above, we get the following.



This is how PCA works. I hope this algorithm will never feel daunting again :)



# Supercharge Your Jupyter Kernel With ipyflow

This is a pretty cool Jupyter hack I learned recently.

While using Jupyter, you must have noticed that when you update a variable, all its dependent cells have to be manually re-executed.

Also, at times, isn't it difficult to determine the exact sequence of cell executions that generated an output?

This is tedious and can get time-consuming if the sequence of dependent cells is long.

To resolve this, try **ipyflow**. It is a supercharged kernel for Jupyter, which tracks the relationship between cells and variables.

```
In [1]: import numpy as np

Automatic Execution of Dependent Cells

In [2]: %flow mode reactive

In [ ]: x = 10 ## Updating x automatically executes its dependents

In [ ]: y = np.sin(x) ## Dependent on x
z = np.cos(x) ## Dependent on x

In [ ]: output = y**2 + z**2 ## Dependent on y and z
output

Export Code

In [ ]: from ipyflow import code
print(code(output))

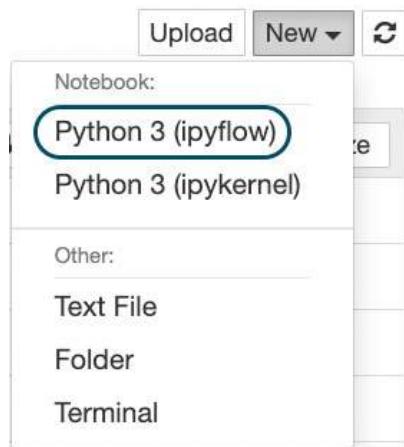
In [ ]:
```

Thus, at any point, you can obtain the corresponding code to reconstruct any symbol.

What's more, its magic command enables an automatic recursive re-execution of dependent cells if a variable is updated.

As shown in the demo above, updating the variable X automatically triggers its dependent cells.

Do note that **ipyflow** offers a different kernel from the default kernel in Jupyter. Thus, once you install **ipyflow**, select the following kernel while launching a new notebook:



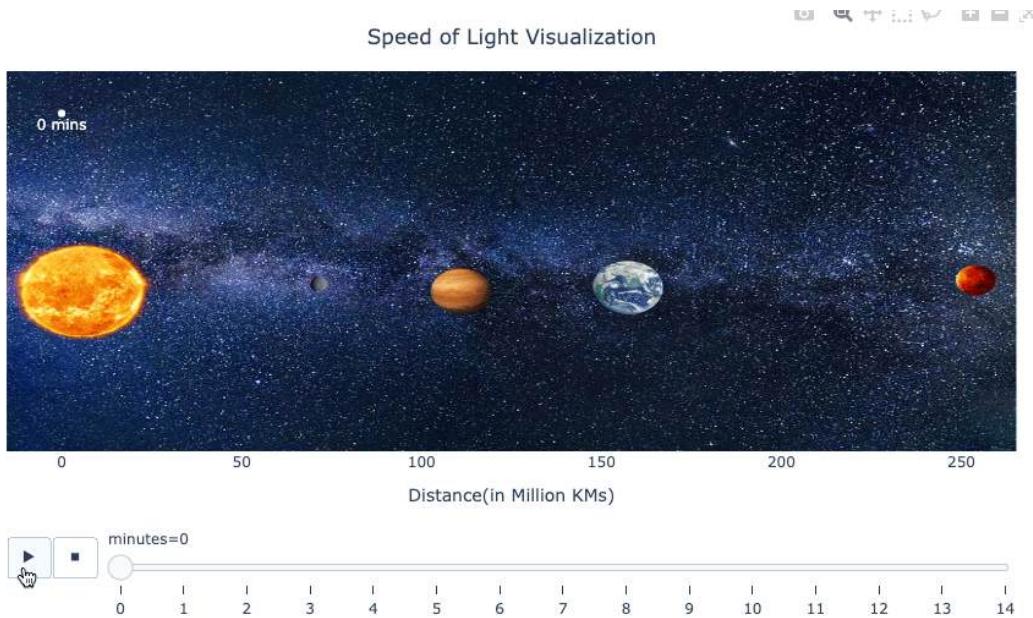
Find more details here: [ipyflow](#).



# A Lesser-known Feature of Creating Plots with Plotly

Plotly is pretty diverse when it comes to creating different types of charts. While many folks prefer it for interactivity, you can also use it to create animated plots.

Here's an animated visualization depicting the time taken by light to reach different planets after leaving the Sun.



Several functions in Plotly support animations using the **animation\_frame** and **animation\_group** parameters.

The core idea behind creating an animated plot relies on plotting the data one frame at a time.

For instance, consider we have organized the data frame-by-frame, as shown below:



	planets	x_position	y_position	frame_id
0	Sun	0.0	0.0	0
1	Mercury	70.0	0.0	0
2	Venus	110.0	0.0	0
3	Earth	150.0	0.0	0
4	Mars	250.0	0.0	0
5	Light	0.0	0.2	0
...				
6	Sun	0.0	0.0	1
7	Mercury	70.0	0.0	1
8	Venus	110.0	0.0	1
9	Earth	150.0	0.0	1
10	Mars	250.0	0.0	1
11	Light	18.0	0.2	1
...				

Now, if we invoke the scatter method with the **animation\_frame** argument, it will plot the data frame-by-frame, giving rise to an animation.

```
import plotly.express as px

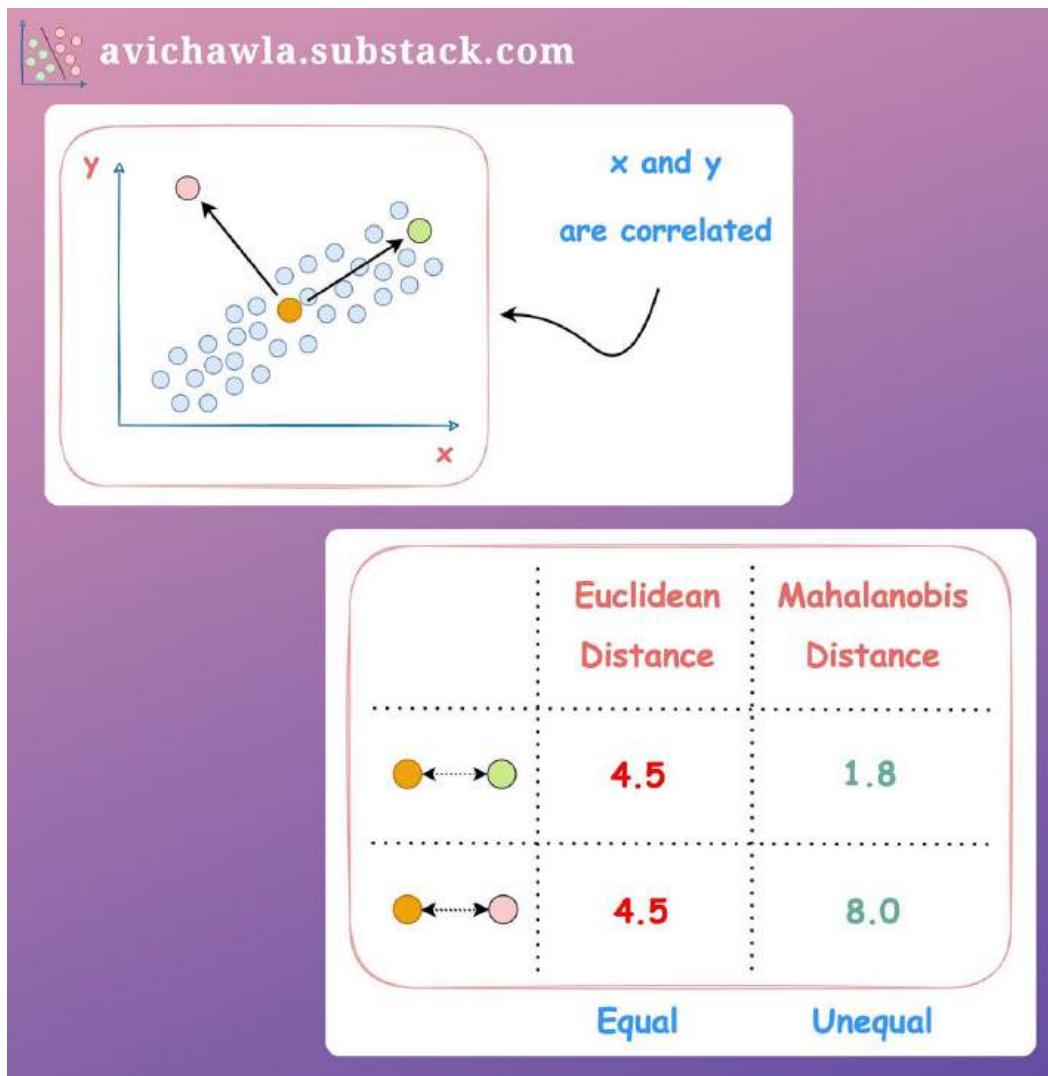
>>> px.scatter(df,
               x="x_position",
               y="y_position",
               color = "planets",
               animation_frame="frame_id")
```

In the above function call, the data corresponding to **frame\_id=0** will be plotted first. This will be replaced by the data with **frame\_id=1** in the next frame, and so on.

Find the code for this post here: [GitHub](#).



# The Limitation Of Euclidean Distance Which Many Often Ignore



Euclidean distance is a commonly used distance metric. Yet, its limitations often make it inapplicable in many data situations.

Euclidean distance assumes independent axes, and the data is somewhat spherically distributed. But when the dimensions are correlated, euclidean may produce misleading results.

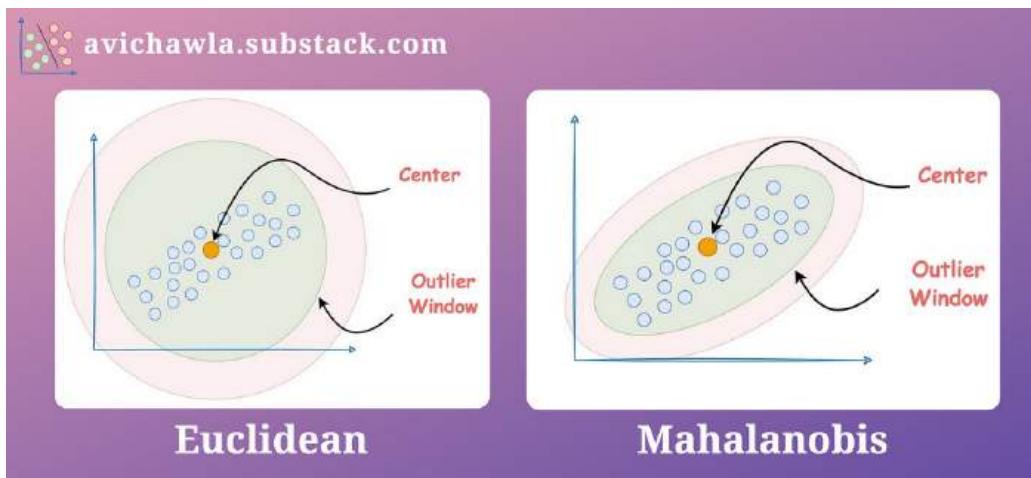
Mahalanobis distance is an excellent alternative in such cases. It is a multivariate distance metric that takes into account the data distribution.

As a result, it can measure how far away a data point is from the distribution, which Euclidean cannot.



As shown in the image above, Euclidean considers pink and green points equidistant from the central point. But Mahalanobis distance considers the green point to be closer, which is indeed true, taking into account the data distribution.

Mahalanobis distance is commonly used in outlier detection tasks. As shown below, while Euclidean forms a circular boundary for outliers, Mahalanobis, instead, considers the distribution—producing a more practical boundary.



Essentially, Mahalanobis distance allows the data to construct a coordinate system for itself, in which the axes are independent and orthogonal.

Computationally, it works as follows:

- **Step 1:** Transform the columns into uncorrelated variables.
- **Step 2:** Scale the new variables to make their variance equal to 1.
- **Step 3:** Find the Euclidean distance in this new coordinate system, where the data has a unit variance.

So eventually, we do reach Euclidean. However, to use Euclidean, we first transform the data to ensure it obeys the assumptions.

Mathematically, it is calculated as follows:

$$D^2 = (\mathbf{x} - \boldsymbol{\mu})^T \cdot \mathbf{C}^{-1} \cdot (\mathbf{x} - \boldsymbol{\mu})$$

- $\mathbf{x}$ : rows of your dataset (Shape:  $n\_samples \times n\_dimensions$ ).
- $\boldsymbol{\mu}$ : mean of individual dimensions (Shape:  $1 \times n\_dimensions$ ).

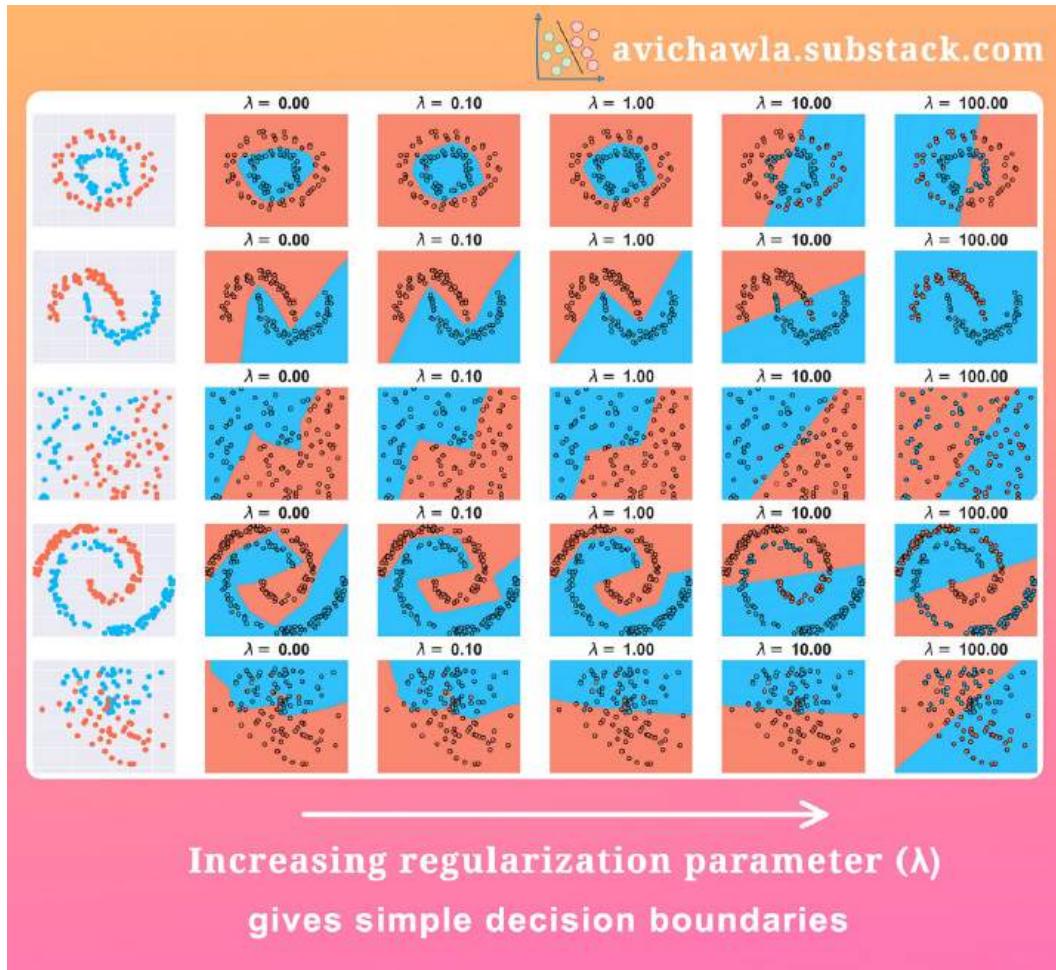


- $C^{-1}$ : Inverse of the covariance matrix  
(Shape:  $n\_dimensions \times n\_dimensions$ ).
- $D^2$ : Square of the Mahalanobis distance  
(Shape:  $n\_samples \times n\_samples$ ).

Find more info here: [Scipy docs](#).



# Visualising The Impact Of Regularisation Parameter



Regularization is commonly used to prevent overfitting. The above visual depicts the decision boundary obtained on various datasets by varying the regularization parameter.

As shown, increasing the parameter results in a decision boundary with fewer curvatures. Similarly, decreasing the parameter produces a more complicated decision boundary.

But have you ever wondered what goes on behind the scenes? Why does increasing the parameter force simpler decision boundaries?

To understand that, consider the cost function equation below (this is for regression though, but the idea stays the same for classification).

It is clear that the cost increases linearly with the parameter  $\lambda$ .



Cost Function = Loss + L2 Weight Penalty

$$= \underbrace{\sum_{i=1}^M (y_i - \sum_{j=1}^N x_{ij} w_j)^2}_{\text{Squared Error}} + \lambda \underbrace{\sum_{j=1}^N w_j^2}_{\text{L2 Regularization Term}}$$

**Higher the value of  $\lambda$ ,  
higher the penalty**

Now, if the parameter is too high, the penalty becomes higher too. Thus, to minimize its impact on the overall cost function, the network is forced to approach weights that are closer to zero.

This becomes evident if we print the final weights for one of the models, say one at the bottom right (last dataset, last model).

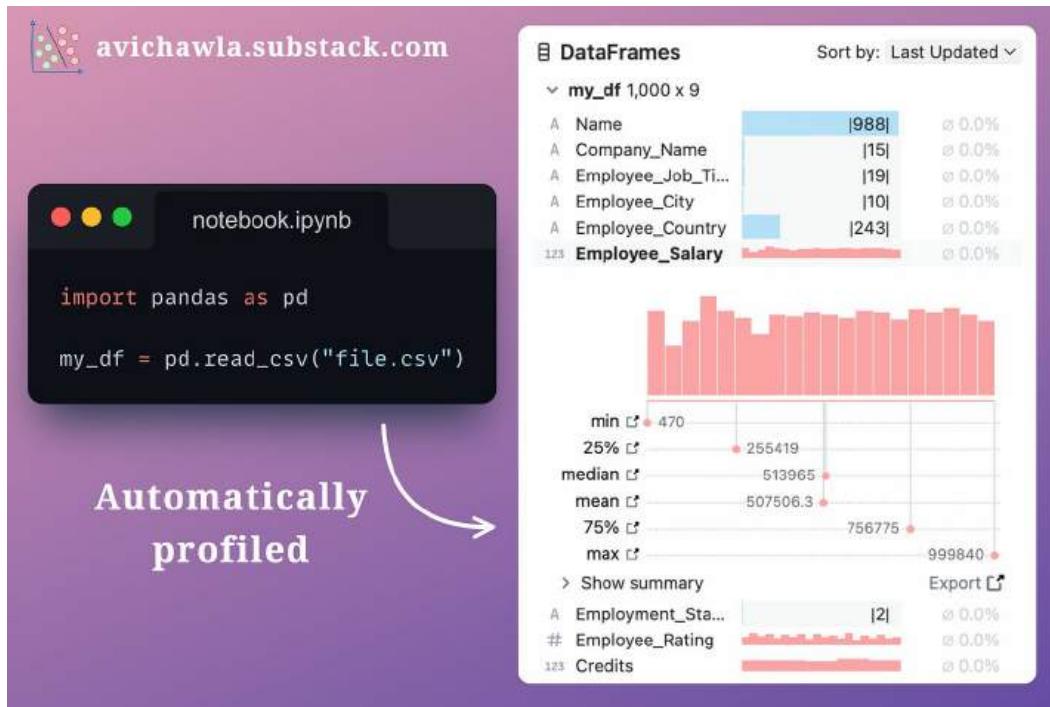
**All weights close to zero**

```
In [17]: clf.coefs_
Out[17]: array([[ 8.35476806e-06, -1.29066987e-05,  1.49535843e-05,
   8.43964067e-06,  5.46943218e-06,  1.18557175e-05,
  1.01037005e-05,  3.70503012e-06,  2.12142850e-06,
 -9.78452613e-06],
 [-1.35980250e-05,  1.52132934e-05,  3.30938991e-06,
  7.41538247e-07,  1.68626879e-05,  1.14315983e-05,
  6.64292409e-07, -1.40798113e-06,  1.31551207e-05,
  2.52379486e-05]])
```

Having smaller weights effectively nullifies many neurons, producing a much simpler network. This prevents many complex transformations, that could have happened otherwise.



# AutoProfiler: Automatically Profile Your DataFrame As You Work



Pandas AutoProfiler: Automatically profile Pandas DataFrames at each execution, without any code.

AutoProfiler is an open-source dataframe analysis tool in jupyter. It reads your notebook and automatically profiles every dataframe in your memory as you change them.

In other words, if you modify an existing dataframe, AutoProfiler will automatically update its corresponding profiling.

Also, if you create a new dataframe (say from an existing dataframe), AutoProfiler will automatically profile that as well, as shown below:



The screenshot shows a Jupyter Notebook cell with the following code:

```
import pandas as pd  
my_df = pd.read_csv("file.csv")  
  
new_df = my_df.sample(100)
```

Annotations on the left side of the cell indicate:

- A curved arrow points from the text "New DataFrame" to the line `new_df = my_df.sample(100)`.
- A curved arrow points from the text "Profile" to the word "Profile" in the output.

The output of the cell is a DataFrames profile report titled "DataFrames". It shows the following information for the "new\_df" DataFrame (100 x 9):

Column	Type	Count	Null %
Name	object	100	0.0%
Company_Name	object	15	0.0%
Employee_Job_Ti...	object	19	0.0%
Employee_City	object	10	0.0%
Employee_Country	object	85	0.0%
Employee_Salary	float64	2	0.0%
Employment_Sta...	float64	2	0.0%
Employee_Rating	float64	2	0.0%
Credits	float64	2	0.0%

Below this, it shows the "my\_df" DataFrame (1,000 x 9) with the same columns and their types.

Profiling info includes column distribution, summary stats, null stats, and many more. Moreover, you can also generate the corresponding code, with its export feature.

The screenshot shows a Jupyter Notebook cell with the following code:

```
import pandas as pd  
my_df = pd.read_csv("file.csv")  
  
my_df[my_df["Name"] == "Sarah Smith"]
```

Annotations on the left side of the cell indicate:

- A curved arrow points from the text "Code added in cell" to the line `my_df[my_df["Name"] == "Sarah Smith"]`.
- A curved arrow points from the text "Export code" to the "Export" button in the profile report.

The output of the cell is a DataFrames profile report titled "DataFrames". It shows the following information for the filtered DataFrame (1,000 x 9):

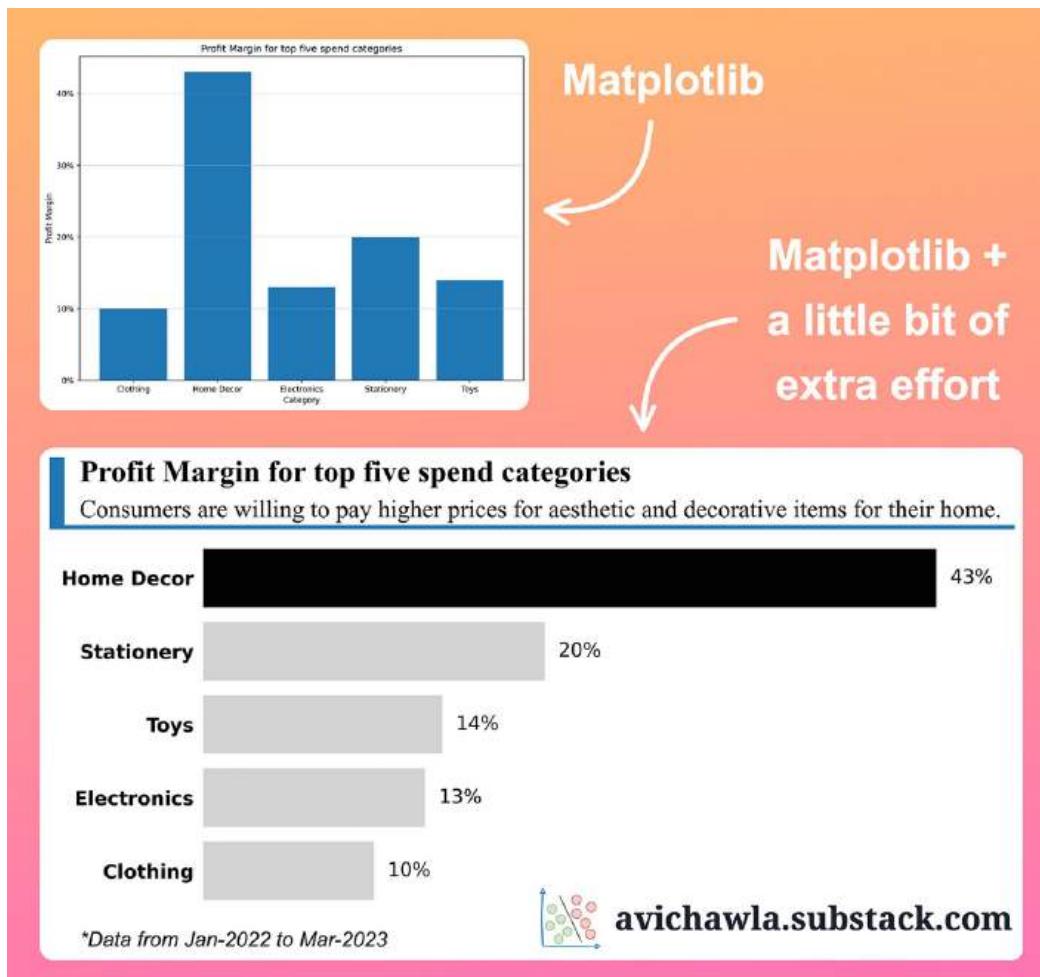
Column	Type	Count	Null %
Name	object	988	0.0%
Kelly Young	object	2	(0.20%)
Renee Davis	object	2	(0.20%)
Patty Jones	object	2	(0.20%)
William Jones	object	2	(0.20%)
Daniel Lee	object	2	(0.20%)
Sarah Smith	object	2	(0.20%)
Anna Thomas	object	2	(0.20%)
Jennifer Gonzales	object	2	(0.20%)
Nicole Garcia	object	2	(0.20%)
Derek Perez	object	2	(0.20%)

Below this, it shows the original "my\_df" DataFrame (1,000 x 9) with the same columns and their types.

Find more info here: [GitHub Repo](#).



# A Little Bit Of Extra Effort Can Hugely Transform Your Storytelling Skills



Matplotlib is pretty underrated when it comes to creating professional-looking plots. Yet, it is totally capable of doing so.

For instance, consider the two plots below.

Yes, both were created using matplotlib. But a bit of formatting makes the second plot much more informative, appealing, and easy to follow.

The title and subtitle significantly aid the story. Also, the footnote offers extra important information, which is nowhere to be seen in the basic plot.

Lastly, the bold bar immediately draws the viewer's attention and conveys the category's importance.

So what's the message here?



Towards being a good data storyteller, ensure that your plot demands minimal effort from the viewer. Thus, don't shy away from putting in that extra effort. This is especially true for professional environments.

At times, it may be also good to ensure that your visualizations convey the right story, even if they are viewed in your absence.



# A Nasty Hidden Feature of Python That Many Programmers Aren't Aware Of

The screenshot shows a Jupyter Notebook cell. At the top, the title "Mutable Default Parameter" is displayed with a curved arrow pointing down to the code. The code defines a function `add_subject` that takes three parameters: `name`, `subject`, and `subjects` (with a default value of `[]`). Inside the function, `subjects.append(subject)` is executed. The function then returns a dictionary with `'name'` and `'subjects'` keys. Below the code, three calls to `add_subject` are shown: `>>> add_subject('Joe', 'Maths')`, `>>> add_subject('Bob', 'Maths')`, and `>>> add_subject('Roy', 'Maths')`. The output of these calls is displayed below, showing that each call appends a new subject to the same list, resulting in a final list of three subjects for each student.

Output:

```
{'name': 'Joe', 'subjects': ['Maths']}  
{'name': 'Bob', 'subjects': ['Maths', 'Maths']}  
{'name': 'Roy', 'subjects': ['Maths', 'Maths', 'Maths']}
```

Appended to the same list

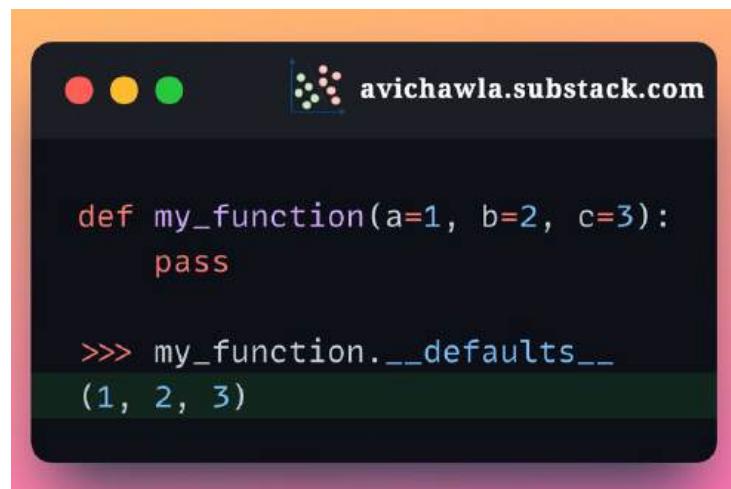
[avichawla.substack.com](http://avichawla.substack.com)

Mutability in Python is possibly one of the most misunderstood and overlooked concepts. The above image demonstrates an example that many Python programmers (especially new ones) struggle to understand.

Can you figure it out? If not, let's understand it.

The default parameters of a function are evaluated right at the time the function is defined. In other words, they are not evaluated each time the function is called (like in C++).

Thus, as soon as a function is defined, the **function object** stores the default parameters in its `__defaults__` attribute. We can verify this below:



```
def my_function(a=1, b=2, c=3):
    pass

>>> my_function.__defaults__
(1, 2, 3)
```

Thus, if you specify a mutable default parameter in a function and mutate it, you unknowingly and unintentionally modify the parameter for all future calls to that function.

This is shown in the demonstration below. Instead of creating a new list at each function call, Python appends the element to the same copy.



Modified  
default  
parameter

```
def add_subject(...):
    ...

>>> add_subject.__defaults__
([],)

>>> add_subject('Joe', 'Maths')
>>> add_subject.__defaults__
(['Maths'],)

>>> add_subject('Bob', 'Maths')
>>> add_subject.__defaults__
(['Maths', 'Maths'])

>>> add_subject('Roy', 'Maths')
>>> add_subject.__defaults__
(['Maths', 'Maths', 'Maths'])
```



## So what can we do to avoid this?

Instead of specifying a mutable default parameter in a function's definition, replace them with None. If the function does not receive a corresponding value during the function call, create the mutable object inside the function.

This is demonstrated below:

**Replace mutable parameter**

```
def add_subject(name, subject, subjects=None):
    if subjects is None:
        # Create if no value was received
        subjects = []

    subjects.append(subject)
    return {'name': name, 'subjects': subjects}

>>> add_subject('Joe', 'Maths')
>>> add_subject('Bob', 'Maths')
>>> add_subject('Roy', 'Maths')
```

Output:

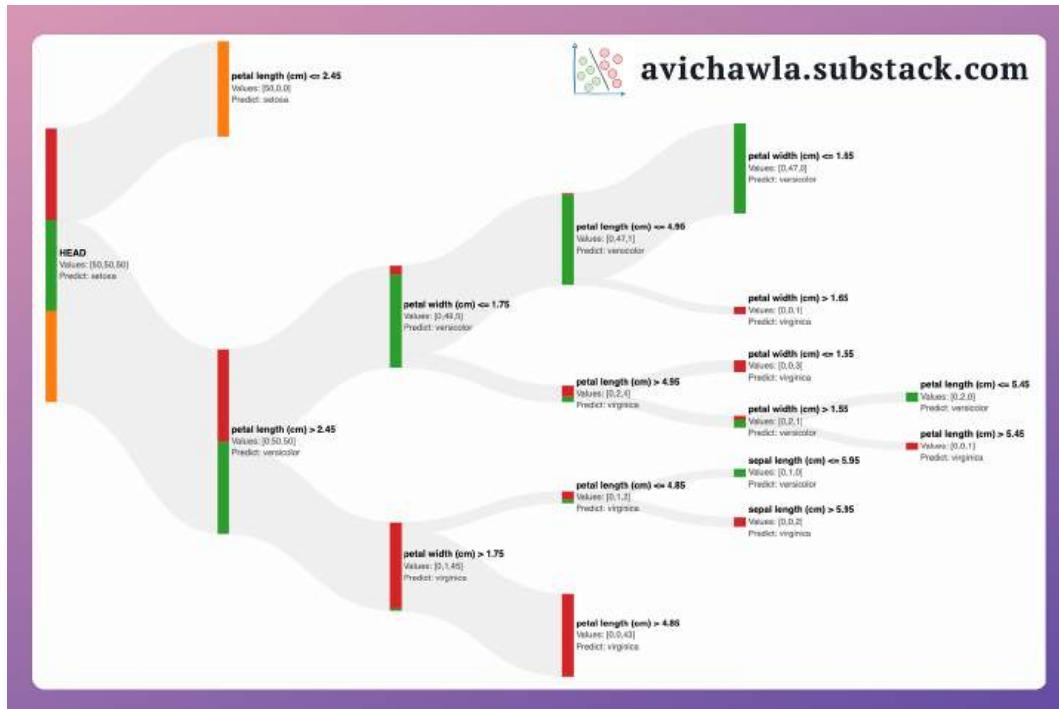
```
{'name': 'Joe', 'subjects': ['Maths'] }
{'name': 'Bob', 'subjects': ['Maths'] }
{'name': 'Roy', 'subjects': ['Maths'] }
```

avichawla.substack.com

As shown above, we create a new list if the function didn't receive any value when it was called. This lets you avoid the unexpected behavior of mutating the same object.



# Interactively Visualise A Decision Tree With A Sankey Diagram



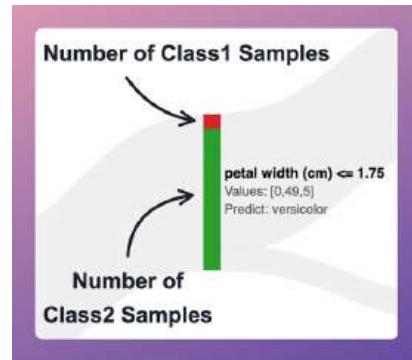
In one of my earlier posts, I explained why sklearn's decision trees always overfit the data with its default parameters (read [here](#) if you wish to recall).

To avoid this, it is always recommended to specify appropriate hyperparameter values. This includes the max depth of the tree, min samples in leaf nodes, etc.

But determining these hyperparameter values is often done using trial-and-error, which can be a bit tedious and time-consuming.

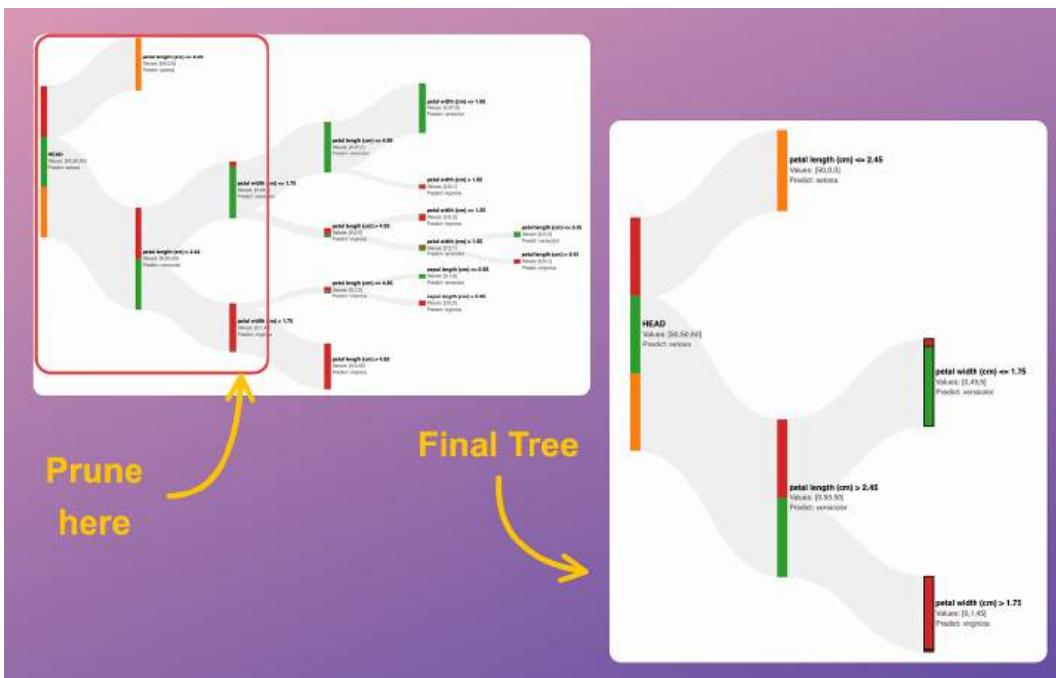
The Sankey diagram above allows you to interactively visualize the predictions of a decision tree at each node.

Also, the number of data points from each class is size-encoded on all nodes, as shown below.



This immediately gives an estimate of the impurity of the node. Based on this, you can visually decide to prune the tree.

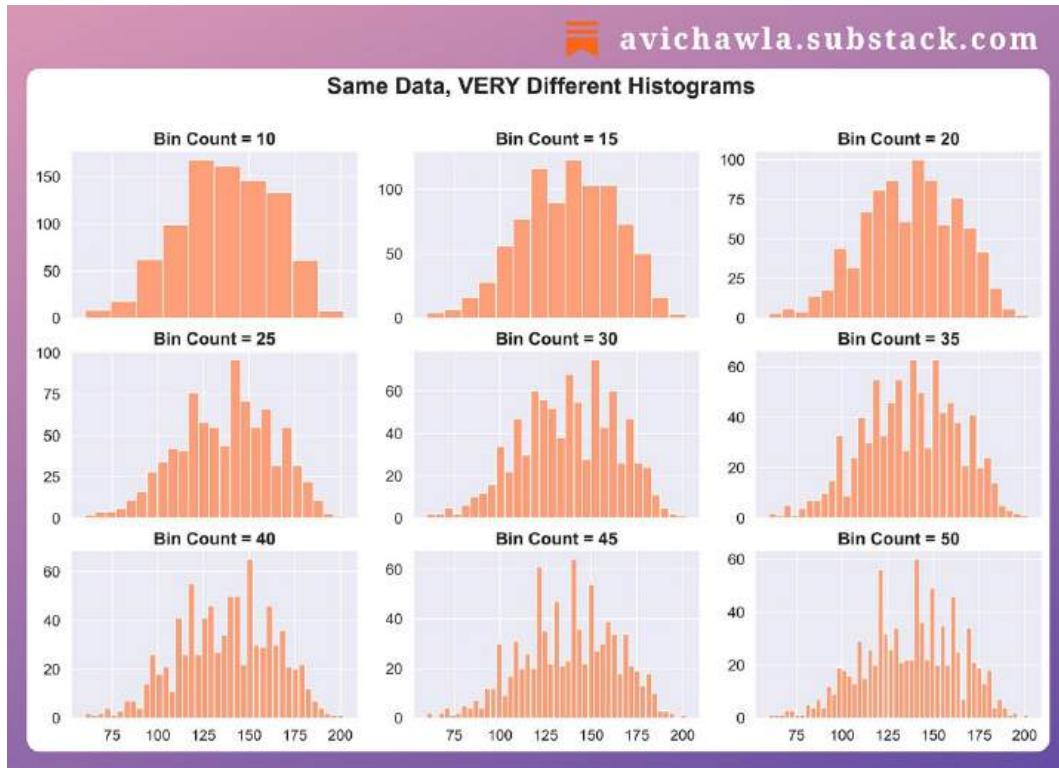
For instance, in the full decision tree shown below, pruning the tree at a depth of two appears to be reasonable.



Once you have obtained a rough estimate for these hyperparameter values, you can train a new decision tree. Next, measure its performance on new data to know if the decision tree is generalizing or not.



# Use Histograms With Caution. They Are Highly Misleading!



Histograms are commonly used for data visualization. But, they can be misleading at times. Here's why.

Histograms divide the data into small bins and represent the frequency of each bin.

Thus, the choice of the number of bins you begin with can significantly impact its shape.

The figure above depicts the histograms obtained on the same data, but by altering the number of bins. Each histogram conveys a different story, even though the underlying data is the same.

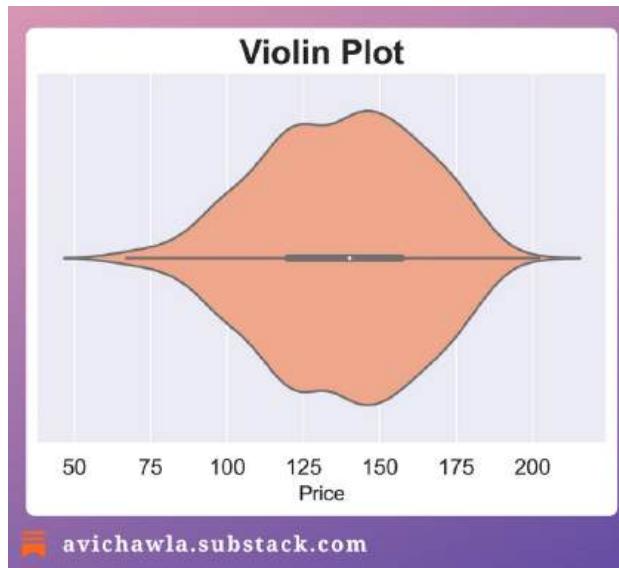
This, at times, can be misleading and may lead you to draw the wrong conclusions.

The takeaway is NOT that histograms should not be used. Instead, look at the underlying distribution too. Here, a violin plot and a KDE plot can help.

**Violin plot**



Similar to box plots, Violin plots also show the distribution of data based on quartiles. However, it also adds a kernel density estimation to display the density of data at different values.



This provides a more detailed view of the distribution, particularly in areas with higher density.

### KDE plot

KDE plots use a smooth curve to represent the data distribution, without the need for binning, as shown below:

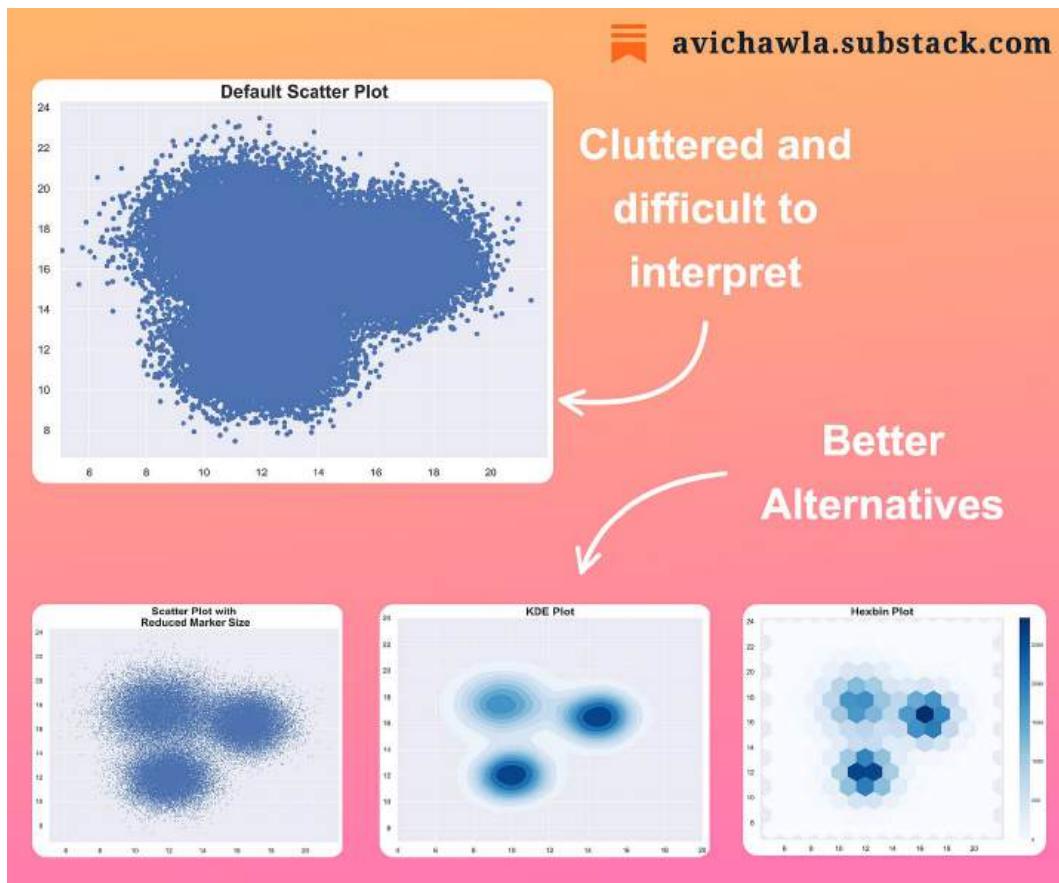


As a departing note, always remember that whenever you condense a dataset, you run the risk of losing important information.

Thus, be mindful of any limitations (and assumptions) of the visualizations you use. Also, consider using multiple methods to ensure that you are seeing the whole picture.



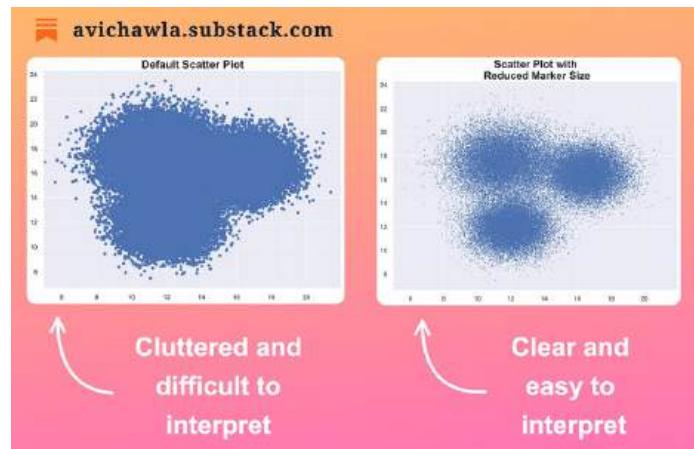
# Three Simple Ways To (Instantly) Make Your Scatter Plots Clutter Free



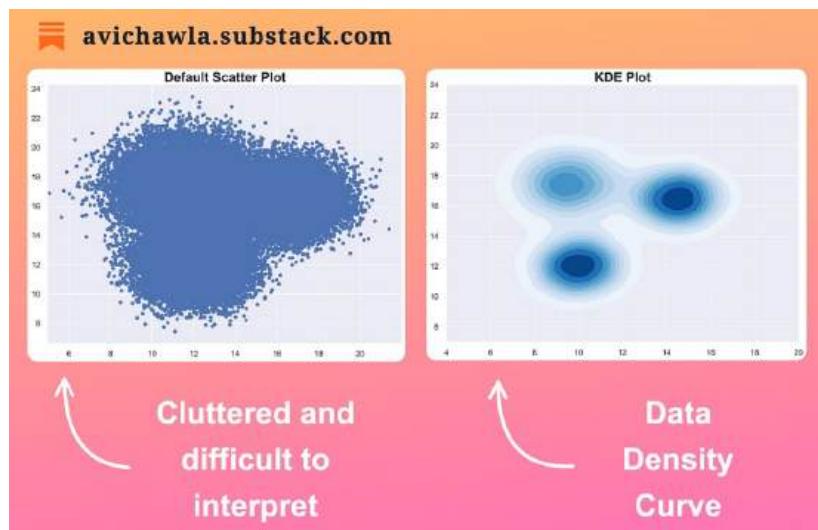
Scatter plots are commonly used in data visualization tasks. But when you have many data points, they often get too dense to interpret.

Here are a few techniques (and alternatives) you can use to make your data more interpretable in such cases.

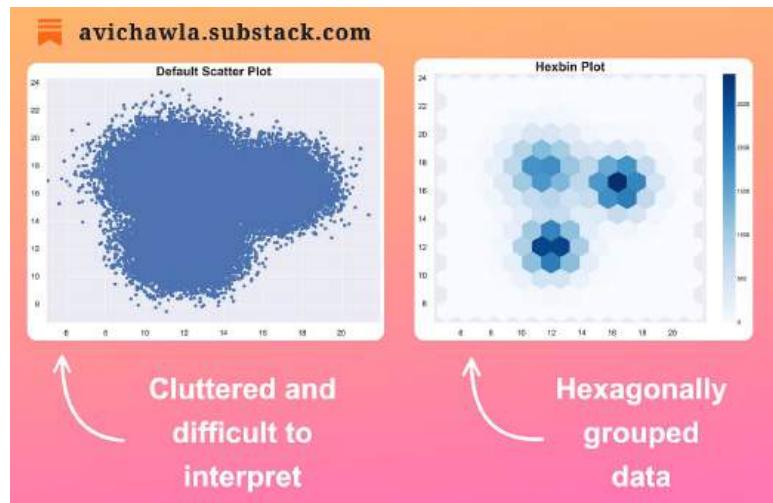
One of the simplest yet effective ways could be to reduce the marker size. This, at times, can instantly offer better clarity over the default plot.



Next, as an alternative to a scatter plot, you can use a density plot, which depicts the data distribution. This makes it easier to identify regions of high and low density, which may not be evident from a scatter plot.

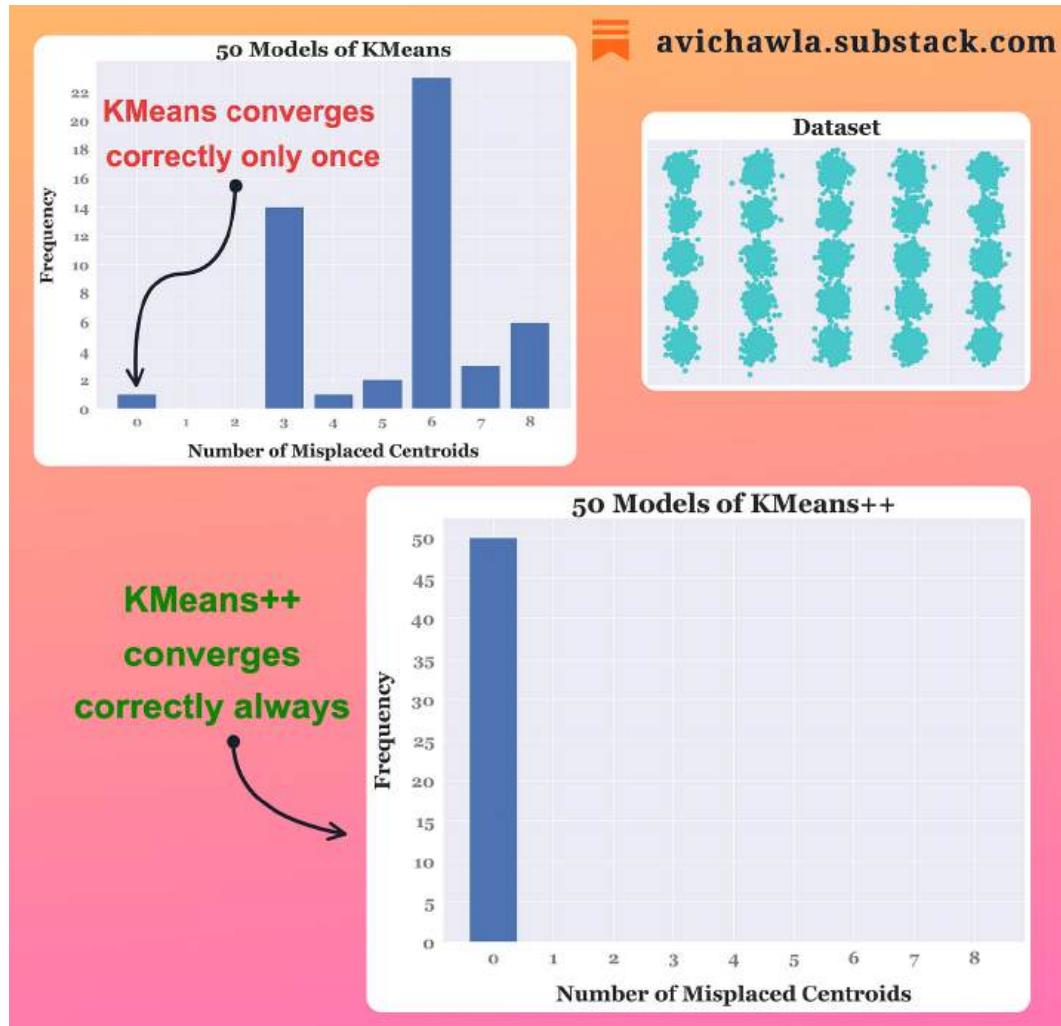


Lastly, another better alternative can be a hexbin plot. It bins the chart into hexagonal regions and assigns a color intensity based on the number of points in that area.





# A (Highly) Important Point to Consider Before You Use KMeans Next Time



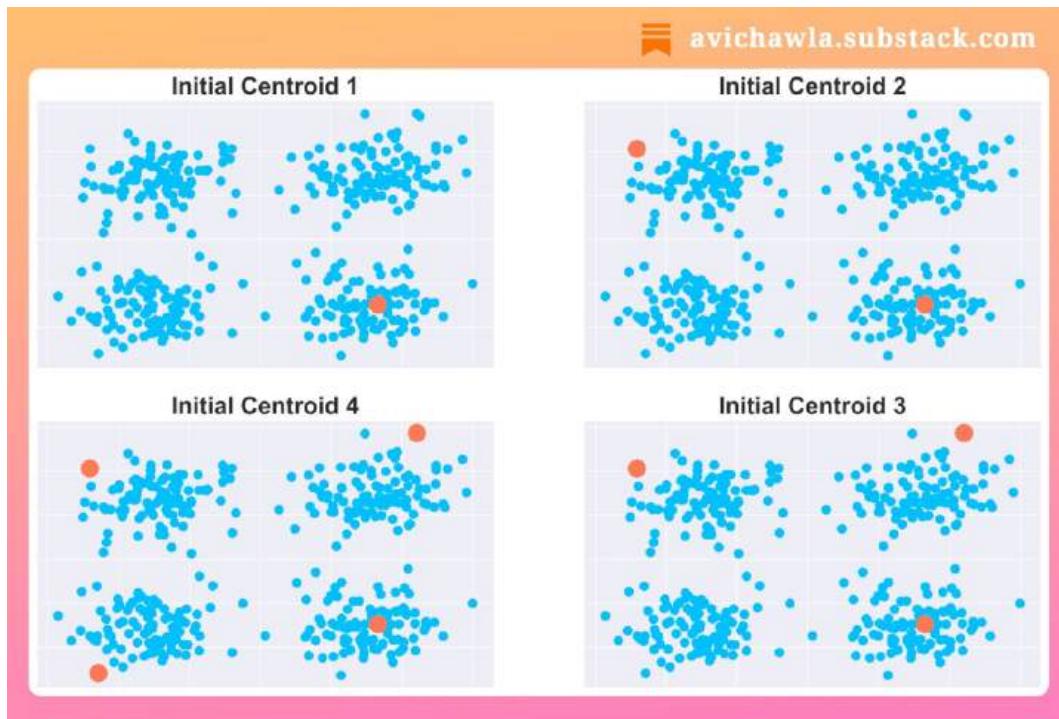
The most important yet often overlooked step of KMeans is its centroid initialization. Here's something to consider before you use it next time.

KMeans selects the initial centroids randomly. As a result, it fails to converge at times. This requires us to repeat clustering several times with different initialization.

Yet, repeated clustering may not guarantee that you will soon end up with the correct clusters. This is especially true when you have many centroids to begin with.

Instead, KMeans++ takes a smarter approach to initialize centroids.

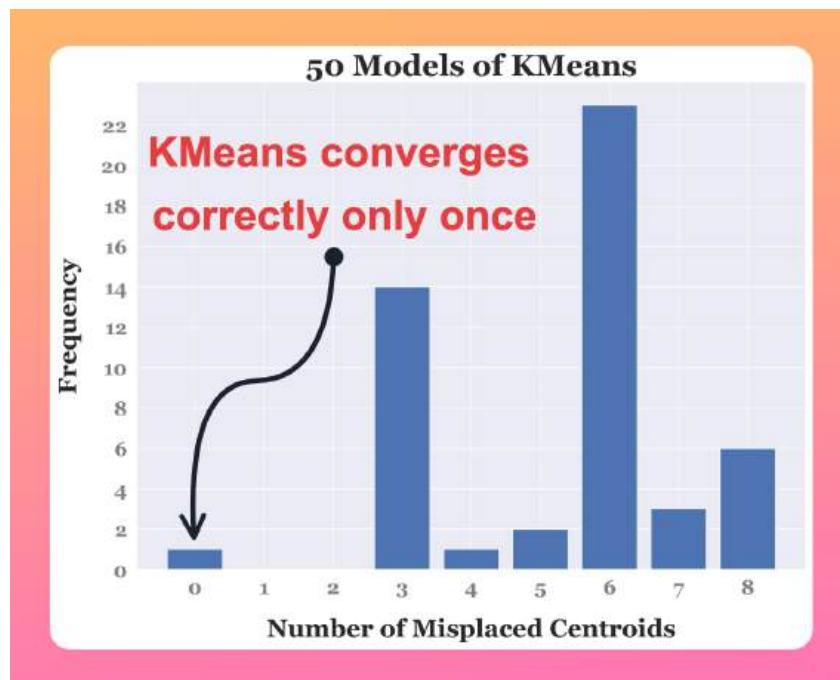
The first centroid is selected randomly. But the next centroid is chosen based on the distance from the first centroid.



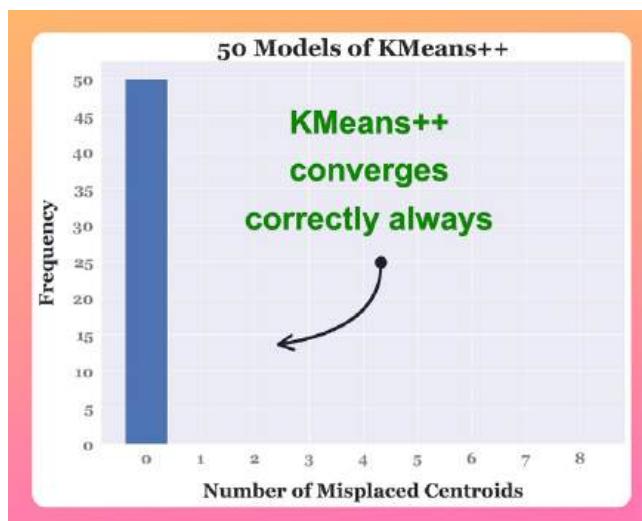
In other words, a point that is away from the first centroid is more likely to be selected as an initial centroid. This way, all the initial centroids are likely to lie in different clusters already, and the algorithm may converge faster and more accurately.

The impact is evident from the bar plots shown below. They depict the frequency of the number of misplaced centroids obtained (analyzed manually) after training 50 different models with KMeans and KMeans++.

On the given dataset, out of the 50 models, KMeans only produced zero misplaced centroids once, which is a success rate of just **2%**.



In contrast, KMeans++ never produced any misplaced centroids.

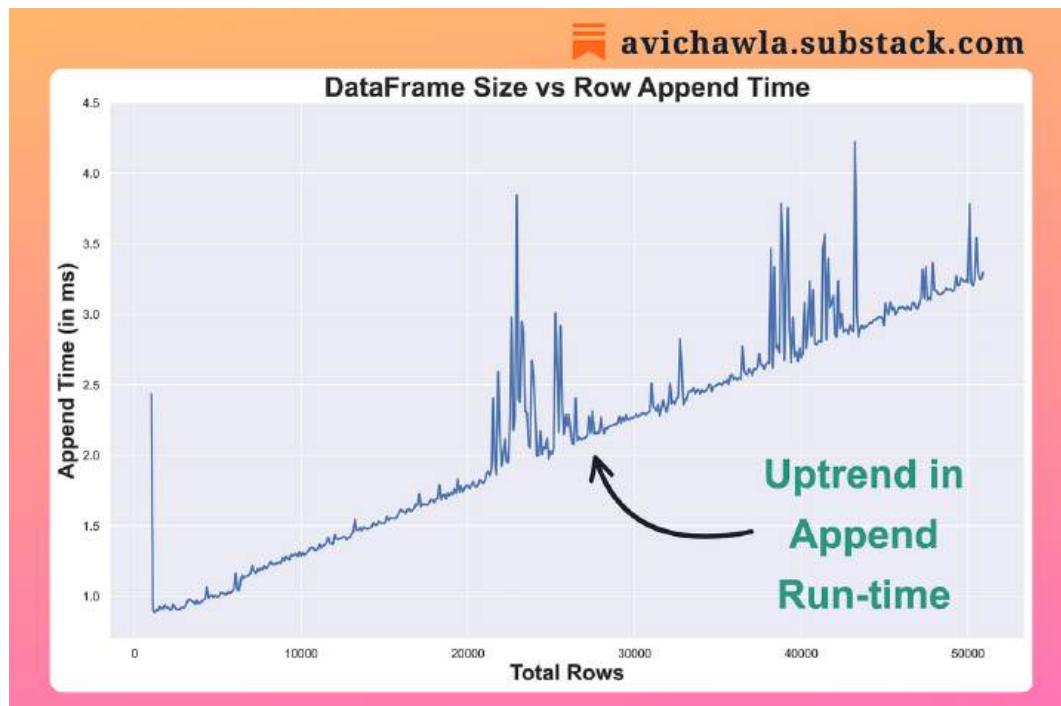


Luckily, if you are using sklearn, you don't need to worry about the initialization step. This is because sklearn, by default, resorts to the KMeans++ approach.

However, if you have a custom implementation, do give it a thought.

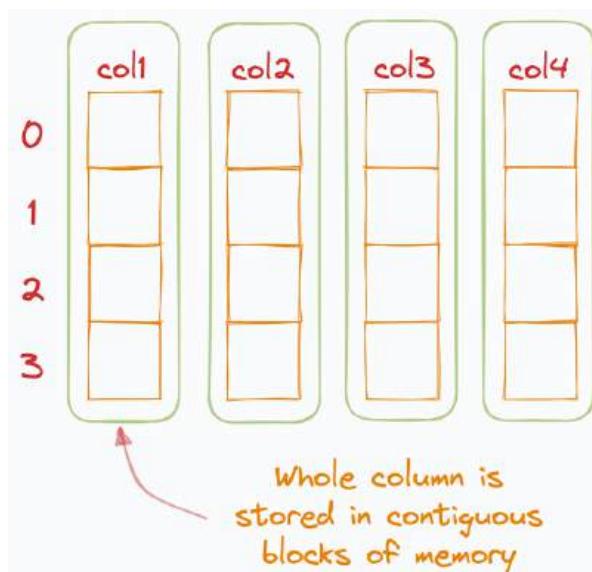


# Why You Should Avoid Appending Rows To A DataFrame



As we append more and more rows to a Pandas DataFrame, the append run-time keeps increasing. Here's why.

A DataFrame is a column-major data structure. Thus, consecutive elements in a column are stored next to each other in memory.





As new rows are added, Pandas always wants to preserve its column-major form.

But while adding new rows, there may not be enough space to accommodate them while also preserving the column-major structure.

In such a case, existing data is moved to a new memory location, where Pandas finds a contiguous block of memory.

Thus, as the size grows, memory reallocation gets more frequent, and the run time keeps increasing.

The reason for spikes in this graph may be because a column taking higher memory was moved to a new location at this point, thereby taking more time to reallocate, or many columns were shifted at once.

### **So what can we do to mitigate this?**

The increase in run-time solely arises because Pandas is trying to maintain its column-major structure.

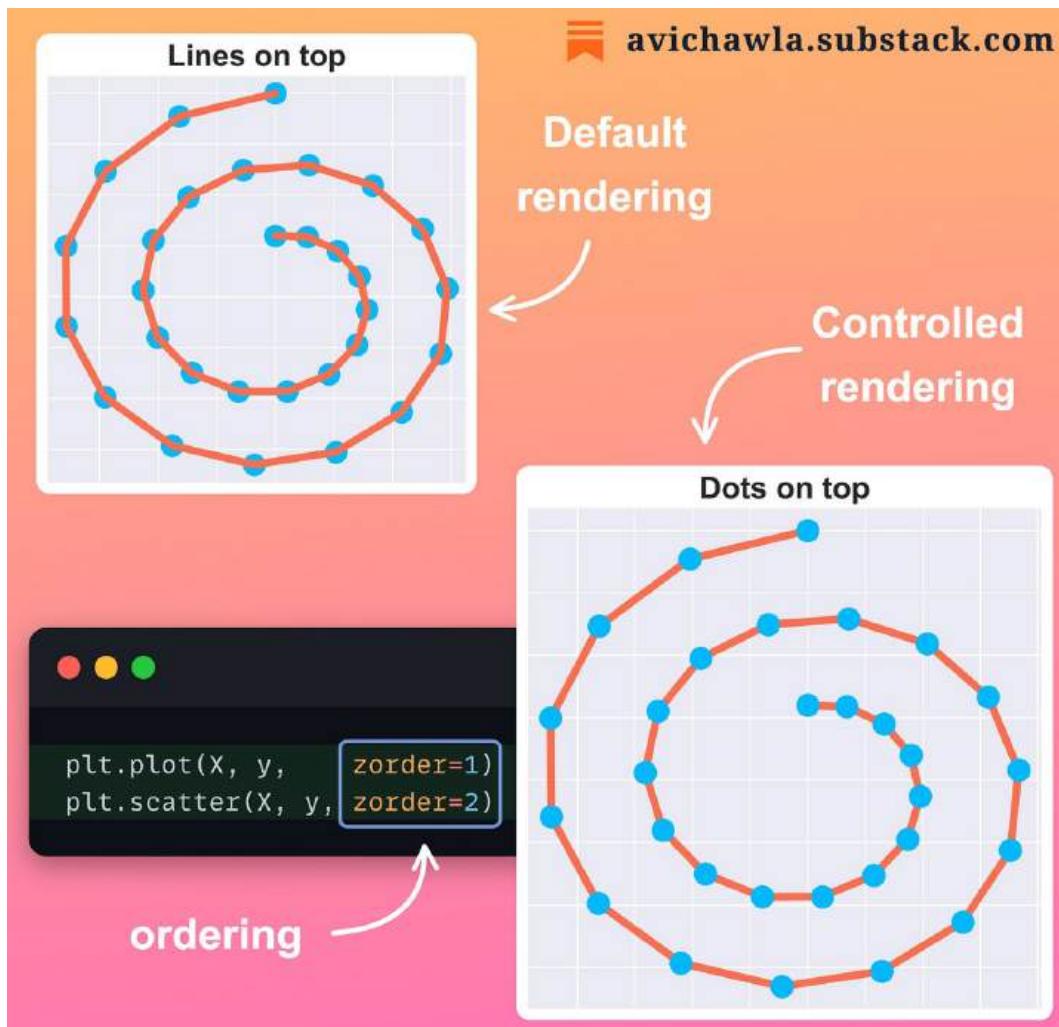
Thus, if you intend to grow a dataframe (row-wise) this frequently, it is better to first convert the dataframe to another data structure, a dictionary or a numpy array, for instance.

Carry out the append operations here, and when you are done, convert it back to a dataframe.

P.S. Adding new columns is not a problem. This is because this operation does not conflict with other columns.



## Matplotlib Has Numerous Hidden Gems. Here's One of Them.



One of the best yet underrated and underutilized potentials of matplotlib is customizability. Here's a pretty interesting thing you can do with it.

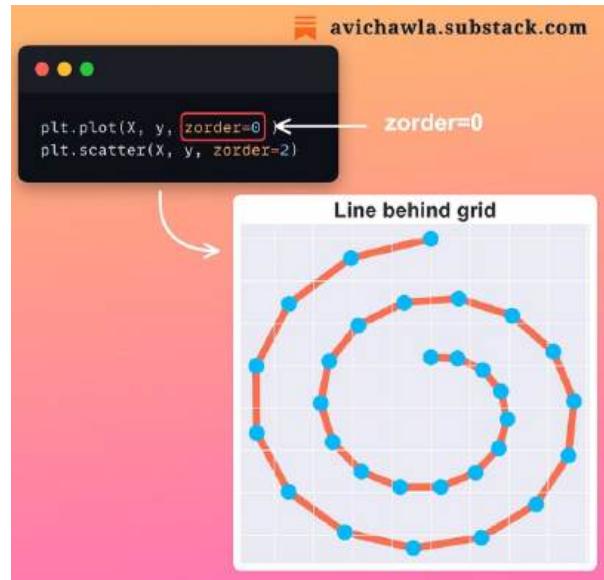
By default, matplotlib renders different types of elements (also called artists), like plots, legend, texts, etc., in a specific order.

But this ordering may not be desirable in all cases, especially when there are overlapping elements in a plot, or the default rendering is hiding some crucial details.

With the `zorder` parameter, you can control this rendering order. As a result, plots with higher `zorder` value appear closer to the viewer and are drawn on top of artists with lower `zorder` values.



Lastly, in the above demonstration, if we specify `zorder=0` for the line plot, we notice that it goes behind the grid lines.



You can find more details about `zorder` here: [Matplotlib docs](#).



# A Counterintuitive Thing About Python Dictionaries

avichawla.substack.com

```
>>> my_dict = {  
    1.0 : 'One (float)',  
    1   : 'One (int)',  
    True : 'One (bool)',  
    '1'  : 'One (string)'  
>>> my_dict  
{1.0 : 'One (bool)', '1' : 'One (string)'}  
Added  
4 keys  
dict only  
has  
2 keys
```

Despite adding 4 distinct keys to a Python dictionary, can you tell why it only preserves two of them?

Here's why.

In Python, dictionaries find a key based on the equivalence of hash (computed using `hash()`), but not identity (computed using `id()`).

In this case, there's no doubt that `1.0`, `1`, and `True` inherently have different datatypes and are also different objects. This is shown below:



```
>>> id(1.0), id(1), id(True)
(153733, 127473, 493931)

>>> type(1.0), type(1), type(True)
(float, int, bool)
```

Yet, as they share the same hash value, the dictionary considers them as the same keys.

```
>>> hash(1.0), hash(1), hash(True)
(1, 1, 1) ## same hash
```

But did you notice that in the demonstration, the final key is 1.0, while the value corresponds to the key True.

```
>>> my_dict
{1.0: 'One (bool)', '1': 'One (string)'}
```

float key                              value of boolean key



This is because, at first, `1.0` is added as a key and its value is '`One (float)`'. Next, while adding the key `1`, python recognizes it as an equivalence of the hash value.

Thus, the value corresponding to `1.0` is overwritten by '`One (int)`', while the key (`1.0`) is kept as is.

Finally, while adding `True`, another hash equivalence is encountered with an existing key of `1.0`. Yet again, the value corresponding to `1.0`, which was updated to '`One (int)`' in the previous step, is overwritten by '`One (bool)`'.

I am sure you may have already guessed why the string key '`1`' is retained.



# Probably The Fastest Way To Execute Your Python Code

```
result = []
for a in range(10000):
    for b in range(10000):
        if (a+b)%11 == 0:
            result.append((a,b))
```

Python

```
$ python big_loop.py
# Run-time: 10.9s
```

100x Faster

Codon

```
$ codon run big_loop.py
# Run-time: 0.11s
```

Many Python programmers are often frustrated with Python's run-time. Here's how you can make your code blazingly fast by changing just one line.

Codon is an open-source, high-performance Python compiler. In contrast to being an interpreter, it compiles your python code to fast machine code.

Thus, post compilation, your code runs at native machine code speed. As a result, typical speedups are often of the order **50x** or more.

According to the official docs, if you know Python, you already know 99% of Codon. There are very minute differences between the two, which you can read here: [Codon docs](#).



Find some more benchmarking results between Python and Codon below:

avichawla.substack.com

fib.py

```
def fib(N):
    """
    Function to find the
    Nth Fibonacci number.

    fib(N) = fib(N-1) + fib(N-2)
    """

    ...
```

pi.py

```
def pi_approx(n_terms):
    """
    Function to find the
    approximate value of pi.

    pi = 4*(1 - 1/3 + 1/5 - 1/7...)
    """

    ...
```

Python

```
$ python fib.py # N=35
# Time: 2.53s
```

```
$ python fib.py # N=45
# Time: 296s
```

```
$ python pi.py # n_terms=10^8
# Time: 14.7s
```

Codon

```
$ codon run fib.py # N=35
# Time: 0.04s (~60x Faster)
```

```
$ codon run fib.py # N=45
# Time: 4.89s (~60x Faster)
```

```
$ codon run pi.py # n_terms=10^8
# Time: 0.35s (~40x Faster)
```



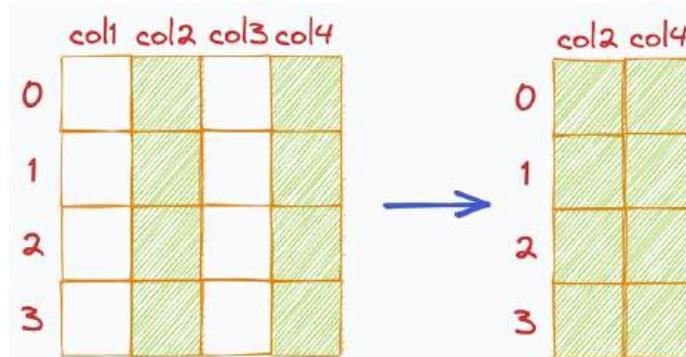
# Are You Sure You Are Using The Correct Pandas Terminologies?



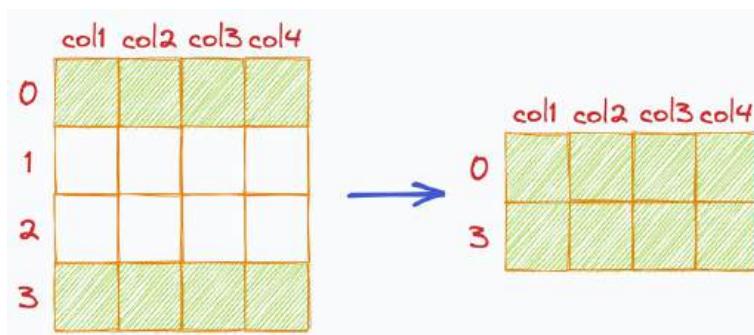
Many Pandas users use the dataframe subsetting terminologies incorrectly. So let's spend a minute to get it straight.

**SUBSETTING** means extracting value(s) from a dataframe. This can be done in four ways:

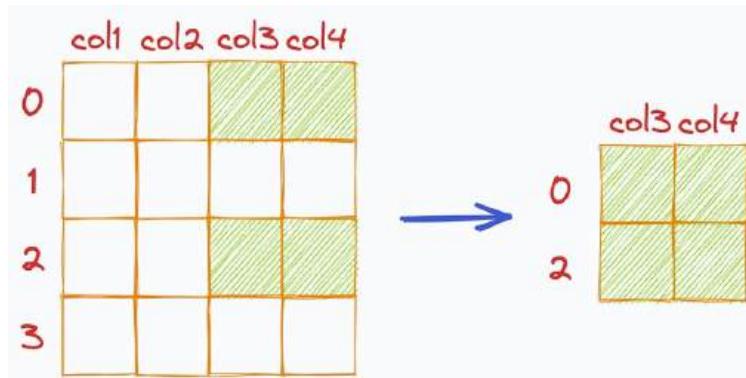
- 1) We call it **SELECTING** when we extract one or more of its **COLUMNS** based on index location or name. The output contains some columns and all rows.



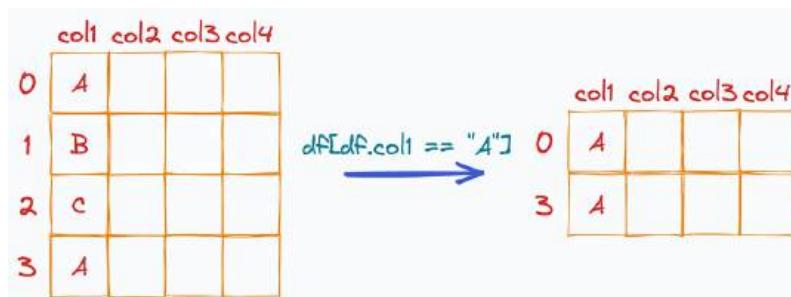
2) We call it **SLICING** when we extract one or more of its **ROWS** based on index location or name. The output contains some rows and all columns.



3) We call it **INDEXING** when we extract both **ROWS** and **COLUMNS** based on index location or name.



4) We call it **FILTERING** when we extract **ROWS** and **COLUMNS** based on conditions.



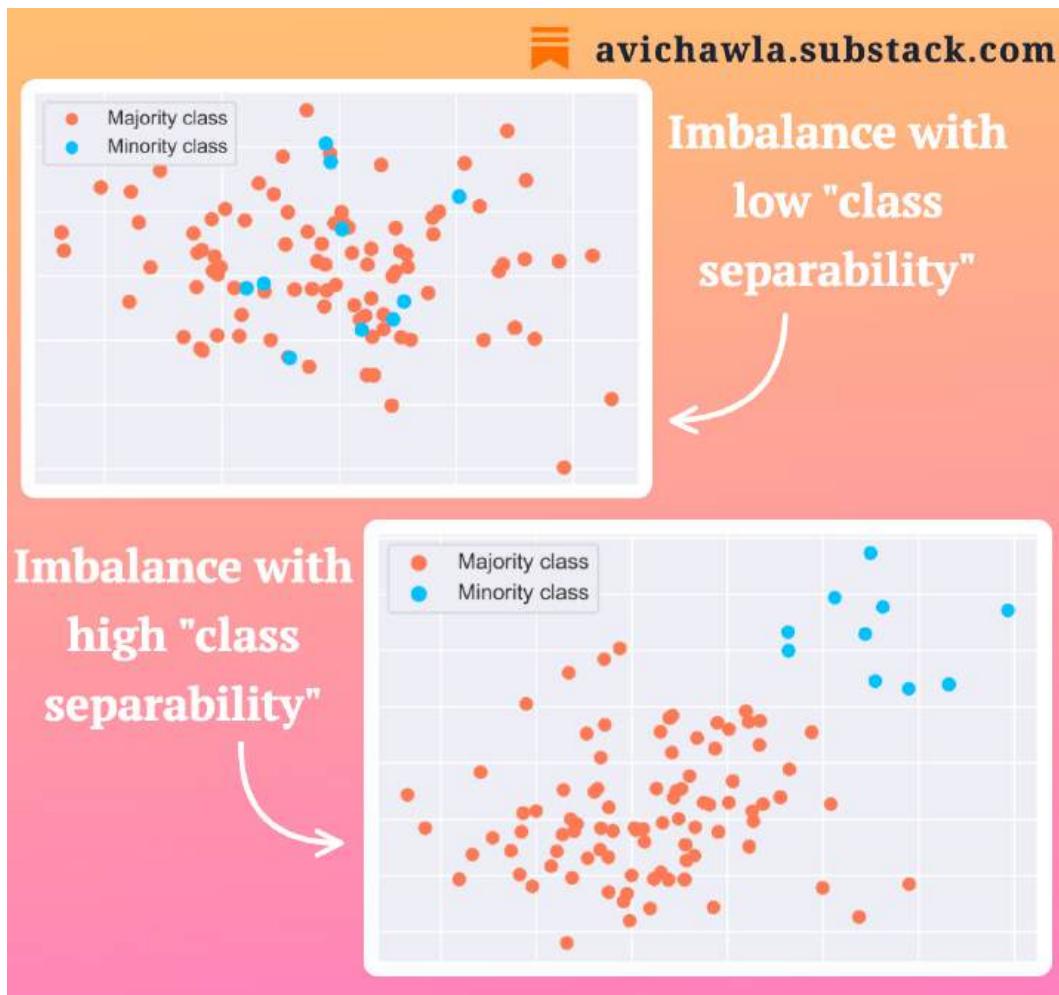


Of course, there are many other ways you can perform these four operations.

Here's a comprehensive Pandas guide I prepared once: [Pandas Map](#). Please refer to the "DF Subset" branch to read about various subsetting methods :)



# Is Class Imbalance Always A Big Problem To Deal With?

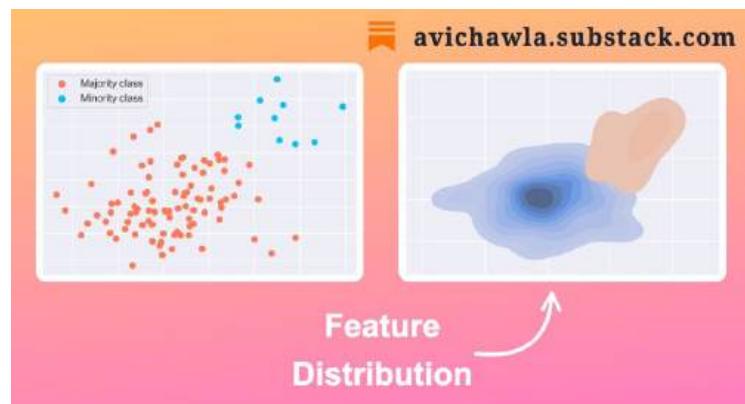


Addressing class imbalance is often a challenge in ML. Yet, it may not always cause a problem. Here's why.

One key factor in determining the impact of imbalance is **class separability**.

As the name suggests, it measures the degree to which two or more classes can be distinguished or separated from each other based on their feature values.

When classes are highly separable, there is little overlap between their feature distributions (as shown below). This makes it easier for a classifier to correctly identify the class of a new instance.

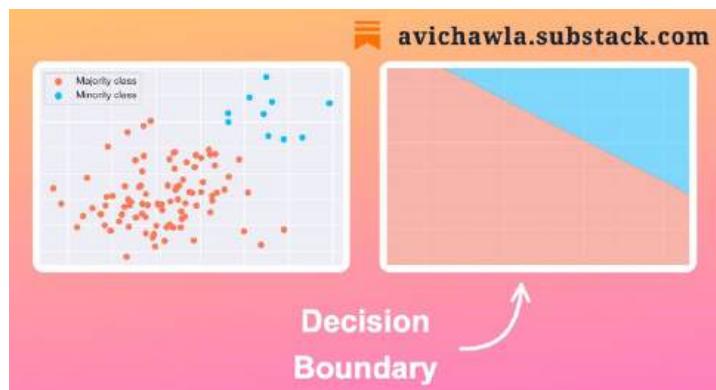


Thus, despite imbalance, even if your data has a high degree of class separability, imbalance may not be a problem per se.

To conclude, consider estimating the class separability before jumping to any sophisticated modeling steps.

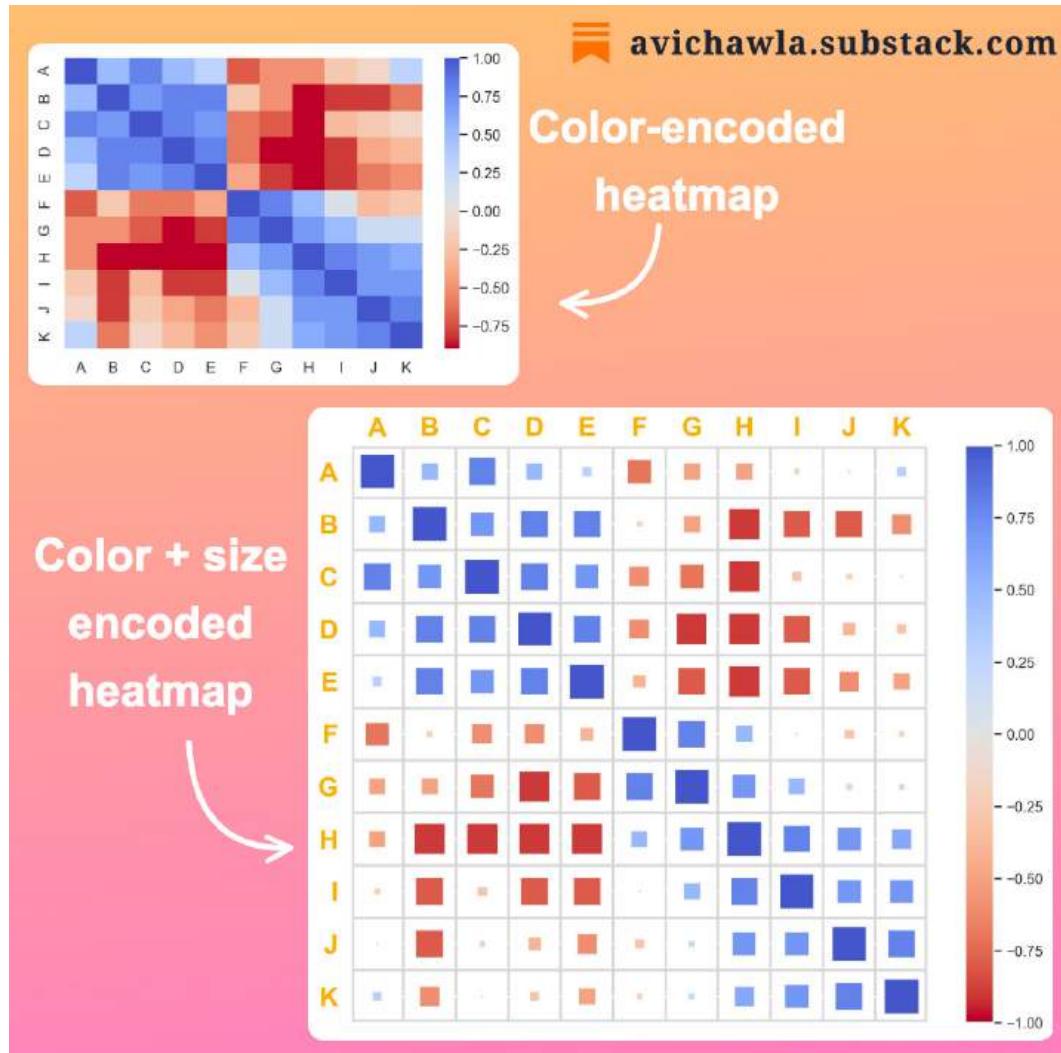
This can be done visually or by evaluating imbalance-specific metrics on simple models.

The figure below depicts the decision boundary learned by a logistic regression model on the class-separable dataset.





# A Simple Trick That Will Make Heatmaps More Elegant



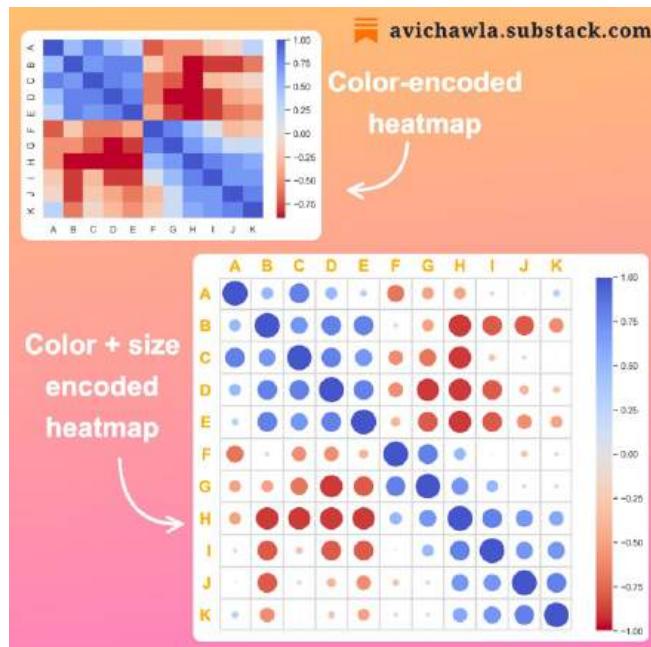
Heatmaps often make data analysis much easier. Yet, they can be further enriched with a simple modification.

A traditional heatmap represents the values using a color scale. Yet, mapping the cell color to numbers is still challenging.

Embedding a size component can be extremely helpful in such cases. In essence, the bigger the size, the higher the absolute value.

This is especially useful to make heatmaps cleaner, as many values nearer to zero will immediately shrink.

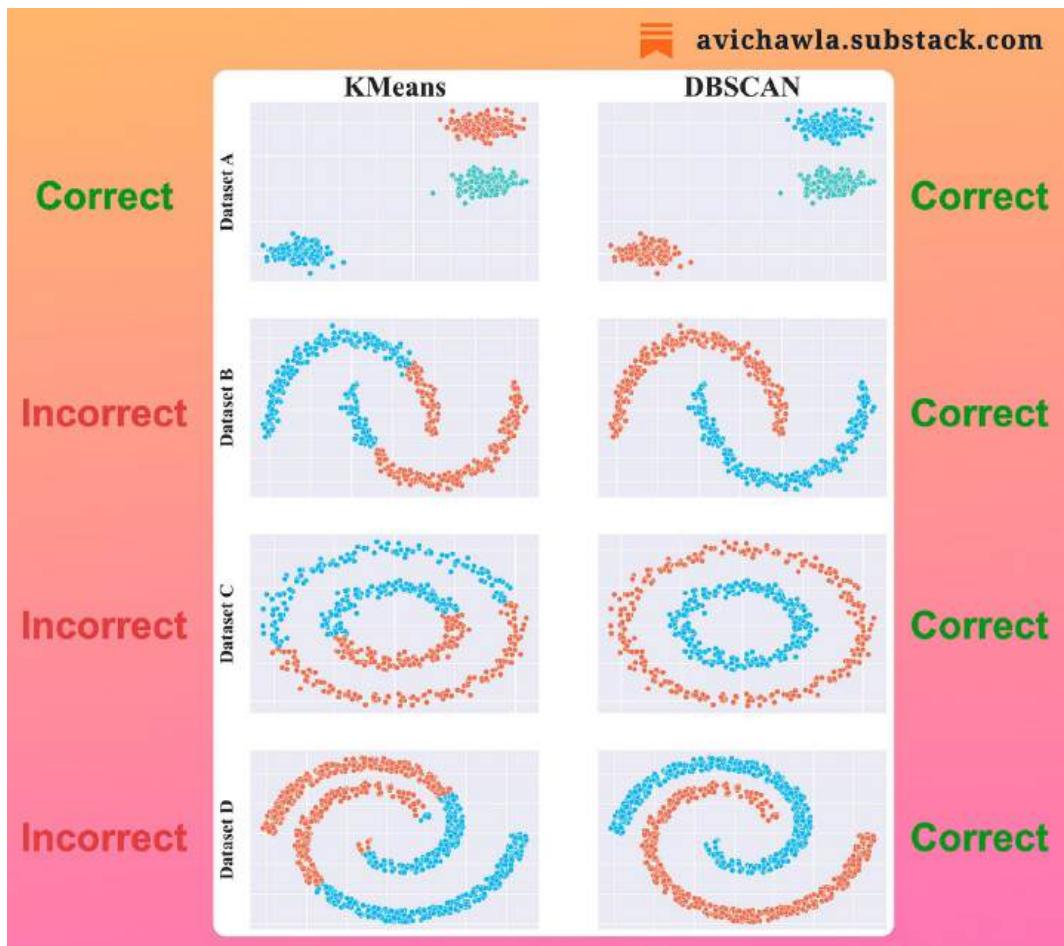
In fact, you can represent the size with any other shape. Below, I created the same heatmap using a circle instead:



Find the code for this post here: [GitHub](https://github.com/avichawla/color_size_encoded_heatmap).



# A Visual Comparison Between Locality and Density-based Clustering



The utility of KMeans is limited to datasets with spherical clusters. Thus, any variation is likely to produce incorrect clustering.

Density-based clustering algorithms, such as DBSCAN, can be a better alternative in such cases.

They cluster data points based on density, making them robust to datasets of varying shapes and sizes.

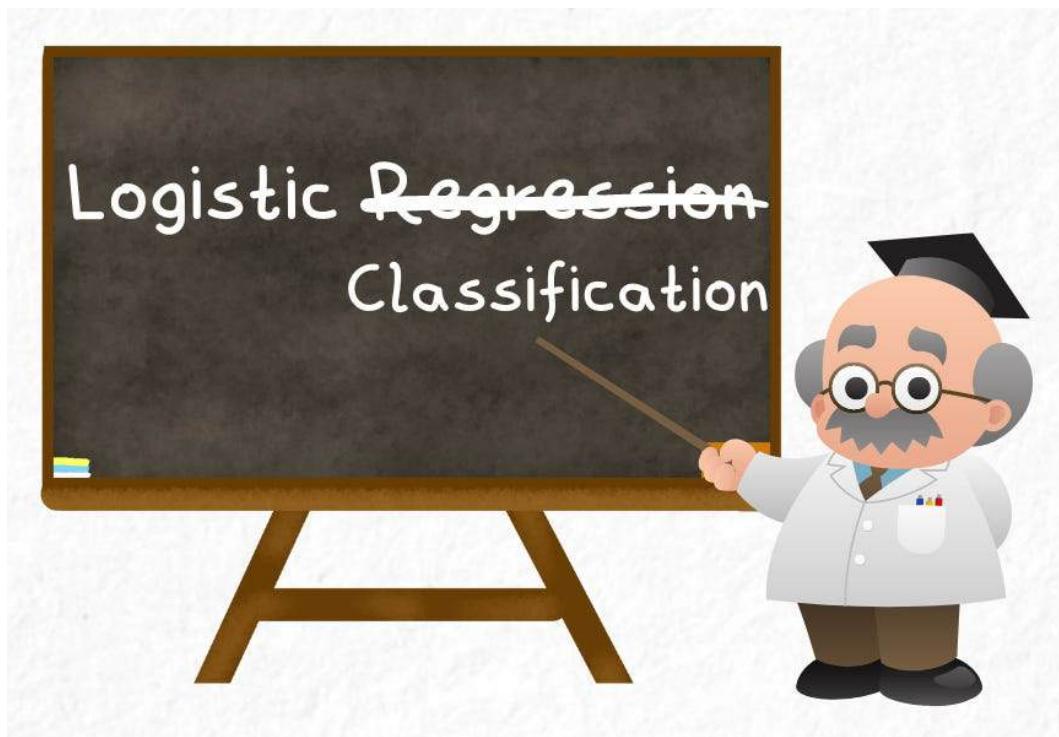
The image depicts a comparison of KMeans vs. DBSCAN on multiple datasets.

As shown, KMeans only works well when the dataset has spherical clusters. But in all other cases, it fails to produce correct clusters.

Find more here: [Sklearn Guide](#).

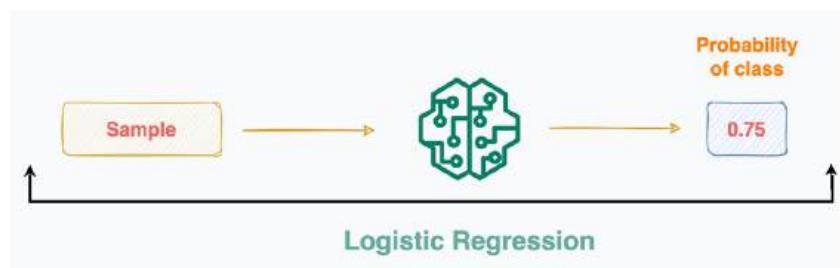


# Why Don't We Call It Logistic Classification Instead?

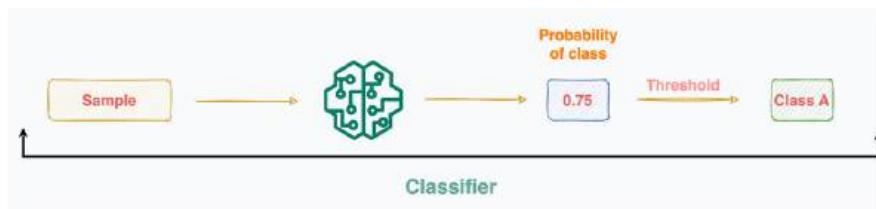


Have you ever wondered why logistic regression is called "regression" when we only use it for classification tasks? Why not call it "logistic classification" instead? Here's why.

Most of us interpret logistic regression as a classification algorithm. However, it is a regression algorithm by nature. This is because it predicts a continuous outcome, which is the probability of a class.



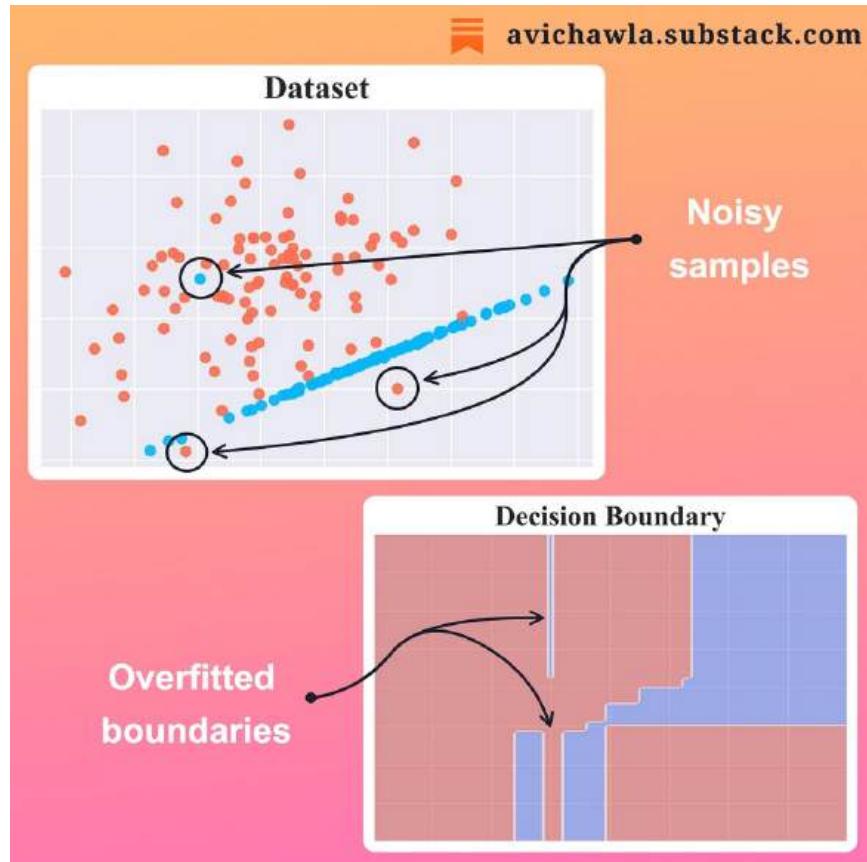
It is only when we apply those thresholds and change the interpretation of its output that the whole pipeline becomes a classifier.



Yet, intrinsically, it is never the algorithm performing the classification. The algorithm always adheres to regression. Instead, it is that extra step of applying probability thresholds that classifies a sample.



# A Typical Thing About Decision Trees Which Many Often Ignore



Although decision trees are simple and intuitive, they always need a bit of extra caution. Here's what you should always remember while training them.

In sklearn's implementation, by default, a decision tree is allowed to grow until all leaves are pure. This leads to overfitting as the model attempts to classify every sample in the training set.

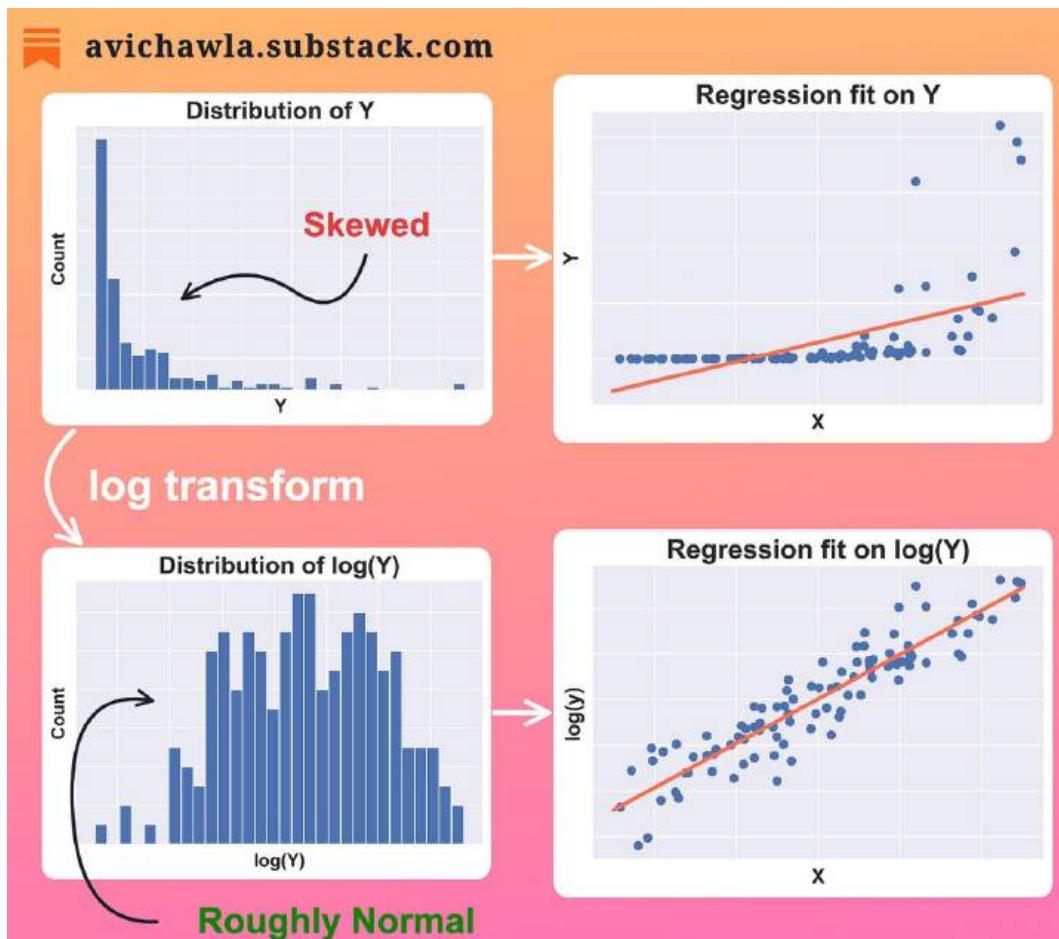
There are various techniques to avoid this, such as pruning and ensembling. Also, make sure that you tune hyperparameters if you use sklearn's implementation.

This was a gentle reminder as many of us often tend to use sklearn's implementations in their default configuration.

It is always a good practice to know what a default implementation is hiding underneath.



# Always Validate Your Output Variable Before Using Linear Regression



The effectiveness of a linear regression model largely depends on how well our data satisfies the algorithm's underlying assumptions.

Linear regression inherently assumes that the residuals (actual-prediction) follow a normal distribution. One way this assumption may get violated is when your output is skewed.

As a result, it will produce an incorrect regression fit.

But the good thing is that it can be corrected. One common way to make the output symmetric before fitting a model is to apply a log transform.

It removes the skewness by evenly spreading out the data, making it look somewhat normal.

One thing to note is that if the output has negative values, a log transform will raise an error. In such cases, one can apply translation transformation first on the output, followed by the log.



# A Counterintuitive Fact About Python Functions

```
# Define a function
>>> def my_func(): pass

# 1) Verify the type of function object
>>> type(my_func)
<class 'function'>

# 2) Add new attributes to function object
>>> my_func.my_attr = 'new_attribute'
>>> my_func.my_attr
'new_attribute'

# 3) Pass as an argument to other functions
>>> def new_func(f): pass
>>> new_func(my_func)

# 4) Access instance-level attributes/methods
>>> my_func.__name__
'my_func'
>>> my_func.__dict__
{'my_attr': 'new_attribute'}
```

Everything in python is an object instantiated from some class. This also includes functions, but accepting this fact often feels counterintuitive at first.

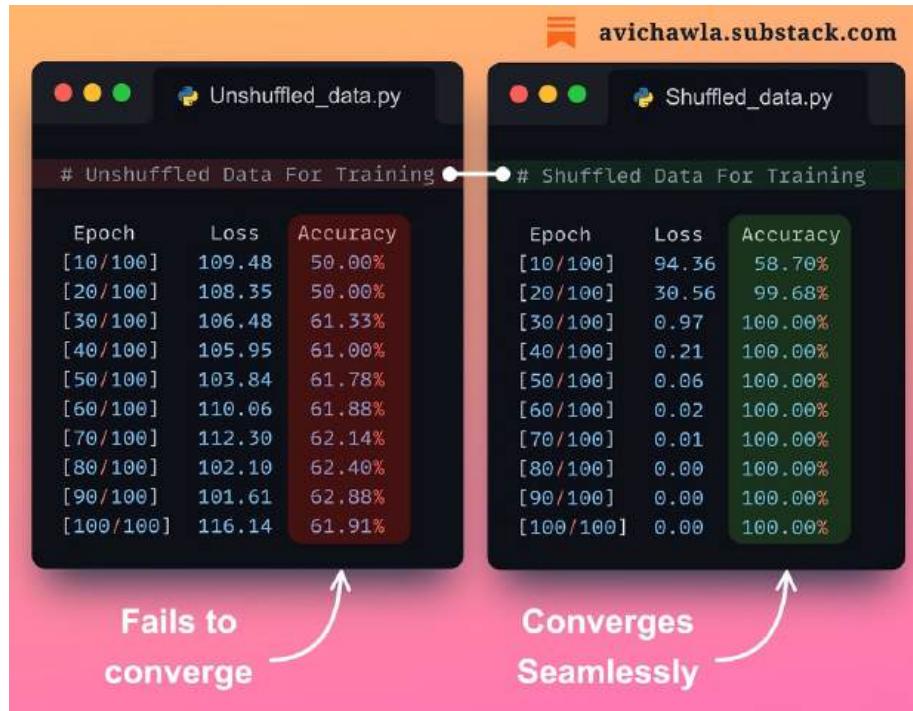
Here are a few ways to verify that python functions are indeed objects.

The friction typically arises due to one's acquaintance with other programming languages like C++ and Java, which work very differently.

However, python is purely an object-oriented programming (OOP) language. You are always using OOP, probably without even realizing it.



# Why Is It Important To Shuffle Your Dataset Before Training An ML Model



ML models may fail to converge for many reasons. Here's one of them which many folks often overlook.

If your data is ordered by labels, this could negatively impact the model's convergence and accuracy. This is a mistake that can typically go unnoticed.

In the above demonstration, I trained two neural nets on the same data. Both networks had the same initial weights, learning rate, and other settings.

However, in one of them, the data was ordered by labels, while in another, it was randomly shuffled.

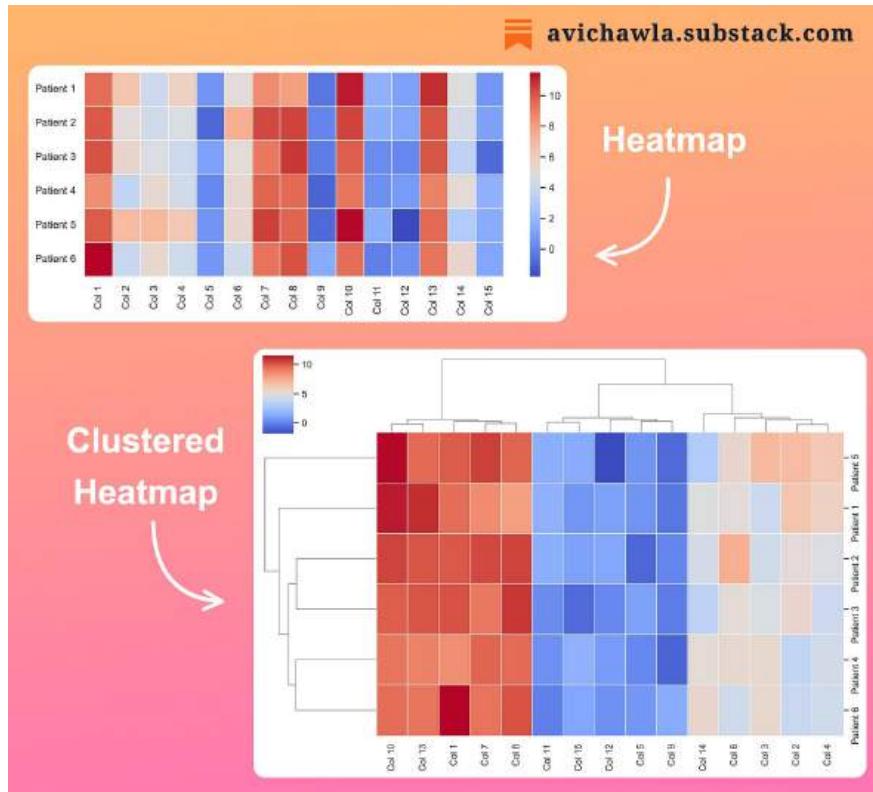
As shown, the model receiving a label-ordered dataset fails to converge. However, shuffling the dataset allows the network to learn from a more representative data sample in each batch. This leads to better generalization and performance.

In general, it's a good practice to shuffle the dataset before training. This prevents the model from identifying any label-specific yet non-existing patterns.

In fact, it is also recommended to alter batch-specific data in every epoch.



# The Limitations Of Heatmap That Are Slowing Down Your Data Analysis



Heatmaps often make data analysis much easier. Yet, they do have some limitations.

A traditional heatmap does not group rows (and features). Instead, its orientation is the same as the input. This makes it difficult to visually determine the similarity between rows (and features).

Clustered heatmaps can be a better choice in such cases. It clusters the rows and features together to help you make better sense of the data.

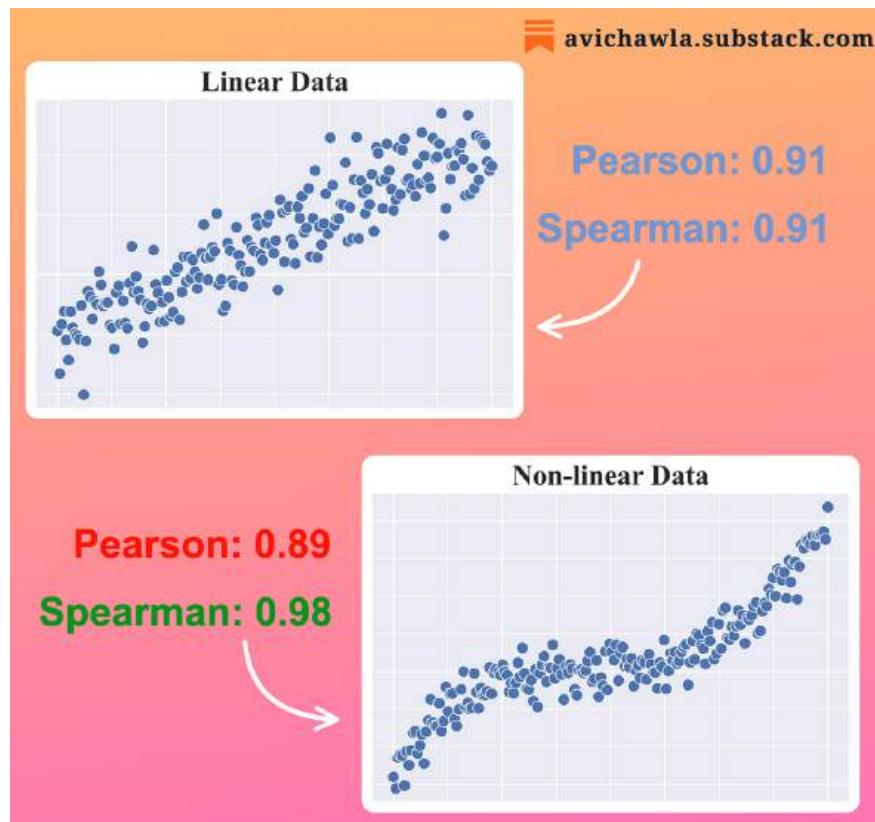
They can be especially useful when dealing with large datasets. While a traditional heatmap will be visually daunting to look at.

However, the groups in a clustered heatmap make it easier to visualize similarities and identify which rows (and features) go with one another.

To create a clustered heatmap, you can use the `sns.clustermap()` method from Seaborn. More info here: [Seaborn docs](#).



# The Limitation Of Pearson Correlation Which Many Often Ignore



Pearson correlation is commonly used to determine the association between two continuous variables. But many often ignore its assumption.

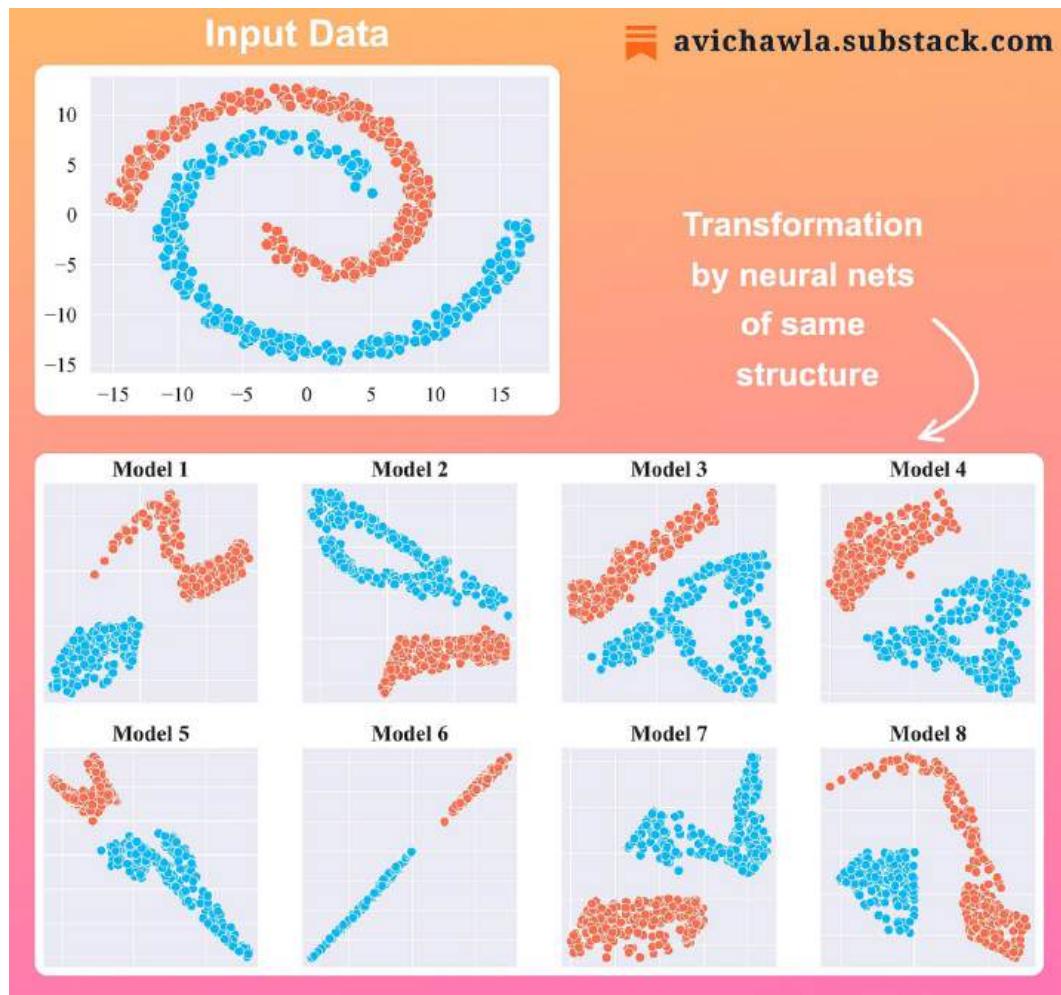
Pearson correlation primarily measures the LINEAR relationship between two variables. As a result, even if two variables have a non-linear but monotonic relationship, Pearson will penalize that.

One great alternative is the Spearman correlation. It primarily assesses the monotonicity between two variables, which may be linear or non-linear.

What's more, Spearman correlation is also useful in situations when your data is ranked or ordinal.



# Why Are We Typically Advised To Set Seeds for Random Generators?



From time to time, we advised to set seeds for random numbers before training an ML model. Here's why.

The weight initialization of a model is done randomly. Thus, any repeated experiment never generates the same set of numbers. This can hinder the reproducibility of your model.

As shown above, the same input data gets transformed in many ways by different neural networks of the same structure.

Thus, before training any model, always ensure that you set seeds so that your experiment is reproducible later.



# An Underrated Technique To Improve Your Data Visualizations



At times, ensuring that your plot conveys the right message may require you to provide additional context. Yet, augmenting extra plots may clutter your whole visualization.

One great way to provide extra info is by adding text annotations to a plot.

In matplotlib, you can use `annotate()`. It adds explanatory texts to your plot, which lets you guide a viewer's attention to specific areas and aid their understanding.

Find more info here: [Matplotlib docs](https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.annotate.html).



# A No-Code Tool to Create Charts and Pivot Tables in Jupyter

The screenshot shows a Jupyter notebook interface. At the top, there's a header with three colored dots (red, yellow, green) and the URL [avichawla.substack.com](http://avichawla.substack.com). Below the header, the notebook title is 'notebook.ipynb'. A code cell contains the following Python code:

```
from pivottablejs import pivot_ui
pivot_ui(df)
```

Below the code cell is a visualization of a pivot table. The visualization has a sidebar on the left with dropdown menus for 'Table', 'Count', and 'Employment\_Status'. The main area shows a grid of data with columns for 'Employee\_City' and 'Employment\_Status' (Full Time, Intern, Totals). The data rows are city names like Aliciafort, Kristaburgh, etc. The visualization is styled with a light gray background and white text.

Employee_City	Full Time	Intern	Totals
Aliciafort	89	24	113
Kristaburgh	77	20	97
New Cindychester	82	24	106
New Russellton	73	20	93
North Melissafurt	62	16	78
Ricardomouth	81	25	106
Wardfort	82	14	96
West Jamesview	94	26	120
Whitakerbury	66	21	87
Whiteside	85	19	104
Totals	791	209	1,000

Here's a quick and easy way to create pivot tables, charts, and group data without writing any code.

PivotTableJS is a drag-n-drop tool for creating pivot tables and interactive charts in Jupyter. What's more, you can also augment pivot tables with heatmaps for enhanced analysis.

Find more info here: [PivotTableJS](#).

**Watch a video version of this post for enhanced understanding: [Video](#).**



# If You Are Not Able To Code A Vectorized Approach, Try This.

df.shape

(100000, 9)

avichawla.substack.com

1) iterrows()

```
%timeit [my_func(row) for index, row in df.iterrows()]
```

2.63 s ± 7.55 ms per loop      **Slowest**

2) apply()

```
%timeit df.apply(my_func, axis = 1)
```

923 ms ± 6 ms per loop      **Slow**

3) itertuples()

```
%timeit [my_func(row) for row in df.itertuples()]
```

87.3 ms ± 486 µs per loop      **Fast**

4) to\_numpy()

```
%timeit np_arr = df.to_numpy(); [my_func(row) for row in np_arr]
```

32.9 ms ± 240 µs per loop      **Fastest**

Although we should never iterate over a dataframe and prefer vectorized code, what if we are not able to come up with a vectorized solution?

In my yesterday's post on why iterating a dataframe is costly, someone posed a pretty genuine question. They asked: "*Let's just say you are forced to iterate. What will be the best way to do so?*"

Firstly, understand that the primary reason behind the slowness of iteration is due to the way a dataframe is stored in memory. (If you wish to recap this, read yesterday's post [here](#).)

Being a column-major data structure, retrieving its rows requires accessing non-contiguous blocks of memory. This increases the run-time drastically.



Yet, if you wish to perform only row-based operations, a quick fix is to convert the dataframe to a NumPy array.

NumPy is faster here because, by default, it stores data in a row-major manner. Thus, its rows are retrieved by accessing contiguous blocks of memory, making it efficient over iterating a dataframe.

That being said, do note that the best way is to write vectorized code always. Use the Pandas-to-NumPy approach only when you are truly struggling with writing vectorized code.



# Why Are We Typically Advised To Never Iterate Over A DataFrame?

The screenshot shows a Jupyter Notebook cell with the following content:

```
df.shape
```

avichawla.substack.com

```
(32768000, 9)
```

**Access column**

```
%timeit df["my_column"]
```

1.73  $\mu$ s ± 546 ns per loop

**Access row**

```
%timeit df.iloc[0]
```

38.4  $\mu$ s ± 1.47  $\mu$ s per loop

A curly brace on the right side groups the two time measurements, with the text "Column access is over 20x faster" written next to it.

From time to time, we are advised to avoid iterating on a Pandas DataFrame. But what is the exact reason behind this? Let me explain.

A DataFrame is a column-major data structure. Thus, consecutive elements in a column are stored next to each other in memory.

As processors are efficient with contiguous blocks of memory, retrieving a column is much faster than a row.

But while iterating, as each row is retrieved by accessing non-contiguous blocks of memory, the run-time increases drastically.

In the image above, retrieving over 32M elements of a column was still over **20x faster** than fetching just nine elements stored in a row.



# Manipulating Mutable Objects In Python Can Get Confusing At Times

```
Method1.py
1 # 1) Define list
2 >>> a = [1,2,3]
3
4 # 2) Assign b to a
5 >>> b = a
6
7 # 3) Modify a
8 >>> a = a + [4,5]
9
10 # 4) Print a
11 >>> a
12 [1, 2, 3, 4, 5] # Modified
13
14 # 5) Print b
15 >>> b
16 [1, 2, 3] # Unchanged

Method2.py
1 # 1) Define list
2 >>> a = [1,2,3]
3
4 # 2) Assign b to a
5 >>> b = a
6
7 # 3) Modify a
8 >>> a += [4,5]
9
10 # 4) Print a
11 >>> a
12 [1, 2, 3, 4, 5] # Modified
13
14 # 5) Print b
15 >>> b
16 [1, 2, 3, 4, 5] # Modified
```

Did you know that with mutable objects, “`a +=`” and “`a = a +`” work differently in Python? Here's why.

Let's consider a list, for instance.

When we use the `=` operator, Python creates a new object in memory and assigns it to the variable.

Thus, all the other variables still reference the previous memory location, which was never updated. This is shown in `Method1.py` above.

But with the `+=` operator, changes are enforced in-place. This means that Python does not create a new object and the same memory location is updated.

Thus, changes are visible through all other variables that reference the same location. This is shown in `Method2.py` above.

We can also verify this by comparing the `id()` pre-assignment and post-assignment.



```
avichawla.substack.com
```

```
Method1.py
```

```
1 # 1) Check ID
2 >>> id(a), id(b)
3 (12345, 12345)
4
5 # 2) Modify a
6 >>> a = a + [4,5]
7
8 # 3) Check ID
9 >>> id(a), id(b)
10 (98765, 12345)
```

**id(a) changed**

```
Method2.py
```

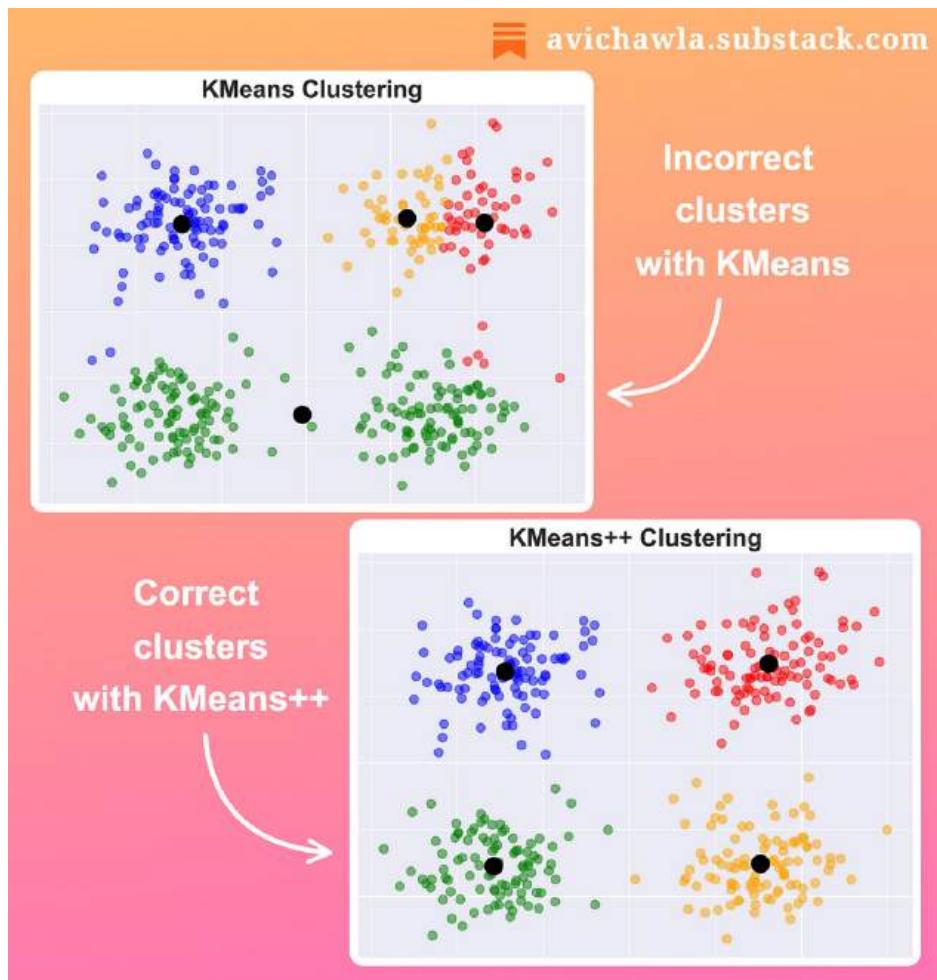
```
1 # 1) Check ID
2 >>> id(a), id(b)
3 (12345, 12345)
4
5 # 2) Modify a
6 >>> a += [4,5]
7
8 # 3) Check ID
9 >>> id(a), id(b)
10 (12345, 12345)
```

**id(a) unchanged**

With “`a = a +`”, the `id` gets changed, indicating that Python created a new object. However, with “`a +=`”, `id` stays the same. This indicates that the same memory location was updated.



# This Small Tweak Can Significantly Boost The Run-time of KMeans



KMeans is a popular but high-run-time clustering algorithm. Here's how a small tweak can significantly improve its run time.

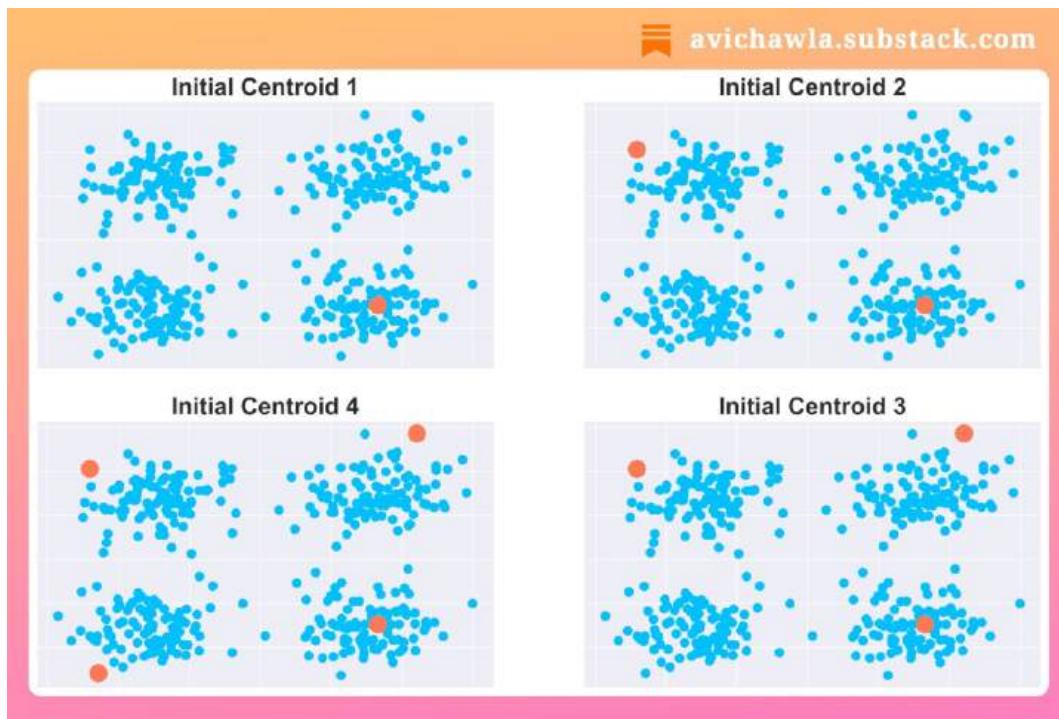
KMeans selects the initial centroids randomly. As a result, it fails to converge at times. This requires us to repeat clustering several times with different initialization.

Instead, KMeans++ takes a smarter approach to initialize centroids. The first centroid is selected randomly. But the next centroid is chosen based on the distance from the first centroid.

In other words, a point that is away from the first centroid is more likely to be selected as an initial centroid. This way, all the initial centroids are likely to lie in different clusters already and the algorithm may converge faster.



The illustration below shows the centroid initialization of KMeans++:





# Most Python Programmers Don't Know This About Python OOP

The screenshot shows a Substack post by avichawla.substack.com. The code defines a `Point2D` class with `__new__` and `__init__` methods. The `__new__` method checks if `x` and `y` are integers, prints "Creating Object!", and returns a new object. If not integers, it raises a `TypeError`. The `__init__` method initializes `x` and `y` and prints "Object Initialized!". Below the code, a terminal session shows creating `p1` with integer values and printing output from both methods. Creating `p2` with float values results in a `TypeError`.

```
class Point2D:
    def __new__(cls, x, y):
        if isinstance(x, int) and isinstance(y, int):
            # Allocate memory and return a new object
            # only when the if-condition is True
            print("Creating Object!")
            return super().__new__(cls) # Return new object
        else:
            raise TypeError("x and y must be integers")

    def __init__(self, x, y):
        self.x = x
        self.y = y
        print("Object Initialized!")

>>> p1 = Point2D(1,2)
'Creating Object!'    # from __new__() method
'Object Initialized!' # from __init__() method

>>> p2 = Point2D(1.5, 2.5)
TypeError: x and y must be integers
```

Most python programmers misunderstand the `__init__()` method. They think that it creates a new object. But that is not true.

When we create an object, it is not the `__init__()` method that allocates memory to it. As the name suggests, `__init__()` only assigns value to an object's attributes.

Instead, Python invokes the `__new__()` method first to create a new object and allocate memory to it. But how is that useful, you may wonder? There are many reasons.

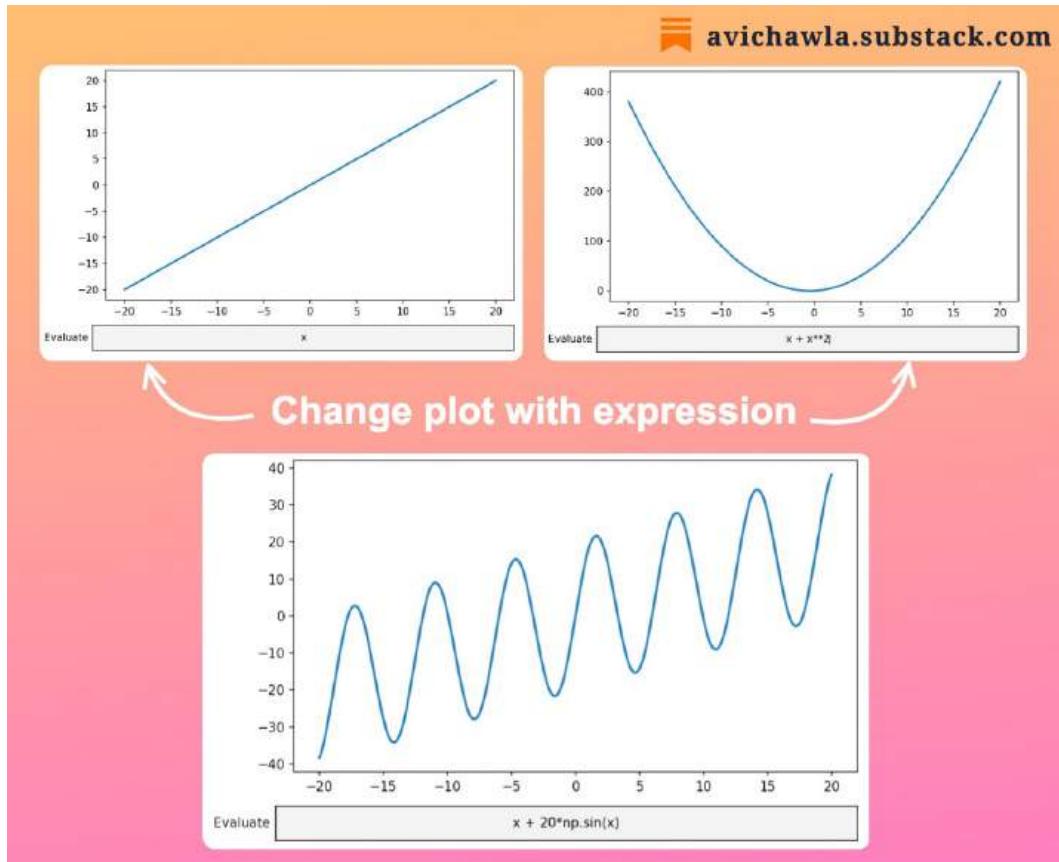
For instance, by implementing the `__new__()` method, you can apply data checks. This ensures that your program allocates memory only when certain conditions are met.



Other common use cases involve defining singleton classes (classes with only one object), creating subclasses of immutable classes such as tuples, etc.



# Who Said Matplotlib Cannot Create Interactive Plots?



👉 Please watch a video version of this post for better understanding: [Video Link](#).

In most cases, Matplotlib is used to create static plots. But very few know that it can create interactive plots too. Here's how.

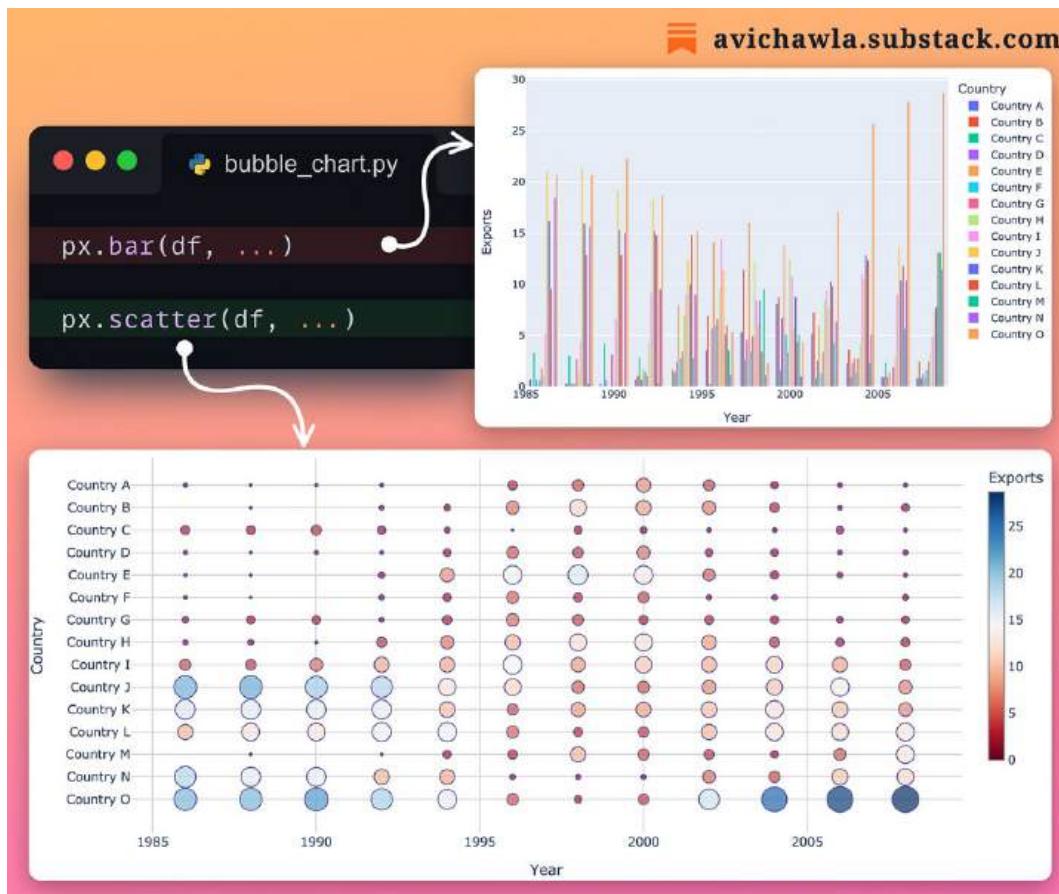
By default, Matplotlib uses the **inline** mode, which renders static plots. However, with the **%matplotlib widget** magic command, you can enable interactive backend for Matplotlib plots.

What's more, its **widgets** module offers many useful widgets. You can integrate them with your plots to make them more elegant.

Find a detailed guide here: [Matplotlib widgets](#).



# Don't Create Messy Bar Plots. Instead, Try Bubble Charts!



Bar plots often get incomprehensible and messy when we have many categories to plot.

A bubble chart can be a better choice in such cases. They are like scatter plots but with one categorical and one continuous axis.

Compared to a bar plot, they are less cluttered and offer better comprehension.

Of course, the choice of plot ultimately depends on the nature of the data and the specific insights you wish to convey.

Which plot do you typically prefer in such situations?



# You Can Add a List As a Dictionary's Key (Technically)!

The image shows a Substack post by [avichawla.substack.com](https://avichawla.substack.com). It contains two screenshots of a terminal window.

**Top Screenshot:** A terminal window showing Python code. The code creates an empty dictionary `my\_dict`, a list `my\_list` with elements [1, 2, 3], and then tries to add `my\_list` as a key to `my\_dict` with value `True`. This results in a `TypeError: unhashable type: 'list'` error. A callout bubble points from the error message to the text "unhashable list".

```
>>> my_dict = {} ## dict
>>> my_list = [1,2,3] ## list
>>> my_dict[my_list] = True
TypeError: unhashable type: 'list'
```

**Bottom Screenshot:** A terminal window showing Python code for a custom list class. The class `MyList` inherits from `list` and overrides the `\_\_hash\_\_` method to return 0. An instance of `MyList` is created with elements [1, 2, 3]. When this instance is added as a key to a dictionary `my\_dict` with value `True`, it works without error. A callout bubble points from the class definition to the text "hashable list".

```
## Inherit list class and implement __hash__ func
class MyList(list):
    def __hash__(self):
        return 0

>>> my_list = MyList([1,2,3])
>>> my_dict[my_list] = True

>>> print(my_dict)
{[1, 2, 3]: True}
```

Python raises an error whenever we add a list as a dictionary's key. But do you know the technical reason behind it? Here you go.

Firstly, understand that everything in Python is an object instantiated from some class. Whenever we add an object as a dict's key, Python invokes the `__hash__` function of that object's class.

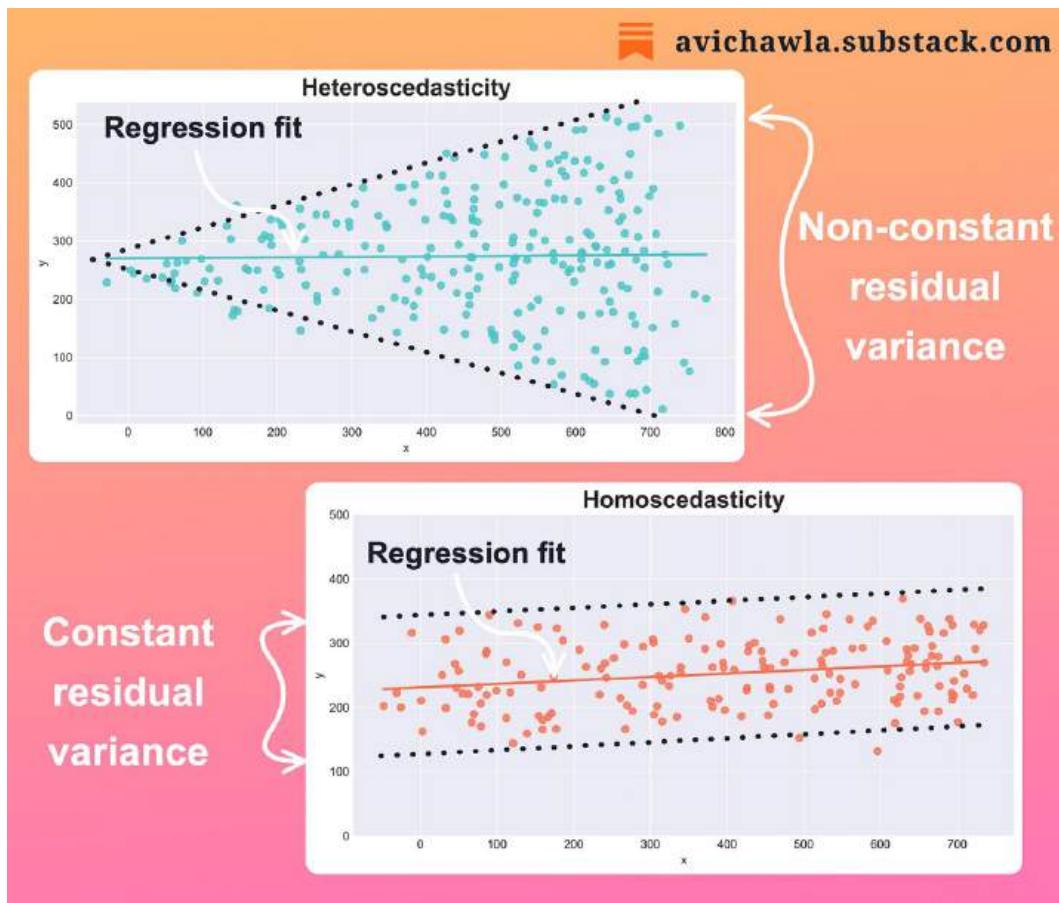
While classes of `int`, `str`, `tuple`, `frozenset`, etc. implement the `__hash__` method, it is missing from the `list` class. That is why we cannot add a list as a dictionary's key.

Thus, technically if we extend the `list` class and add this method, a list can be added as a dictionary's key.

While this makes a list hashable, it isn't recommended as it can lead to unexpected behavior in your code.



# Most ML Folks Often Neglect This While Using Linear Regression



The effectiveness of a linear regression model is determined by how well the data conforms to the algorithm's underlying assumptions.

One highly important, yet often neglected assumption of linear regression is homoscedasticity.

A dataset is homoscedastic if the variability of residuals (=actual-predicted) stays the same across the input range.

In contrast, a dataset is heteroscedastic if the residuals have non-constant variance.

Homoscedasticity is extremely critical for linear regression. This is because it ensures that our regression coefficients are reliable. Moreover, we can trust that the predictions will always stay within the same confidence interval.



## 35 Hidden Python Libraries That Are Absolute Gems



I reviewed 1,000+ Python libraries and discovered these hidden gems I never knew even existed.

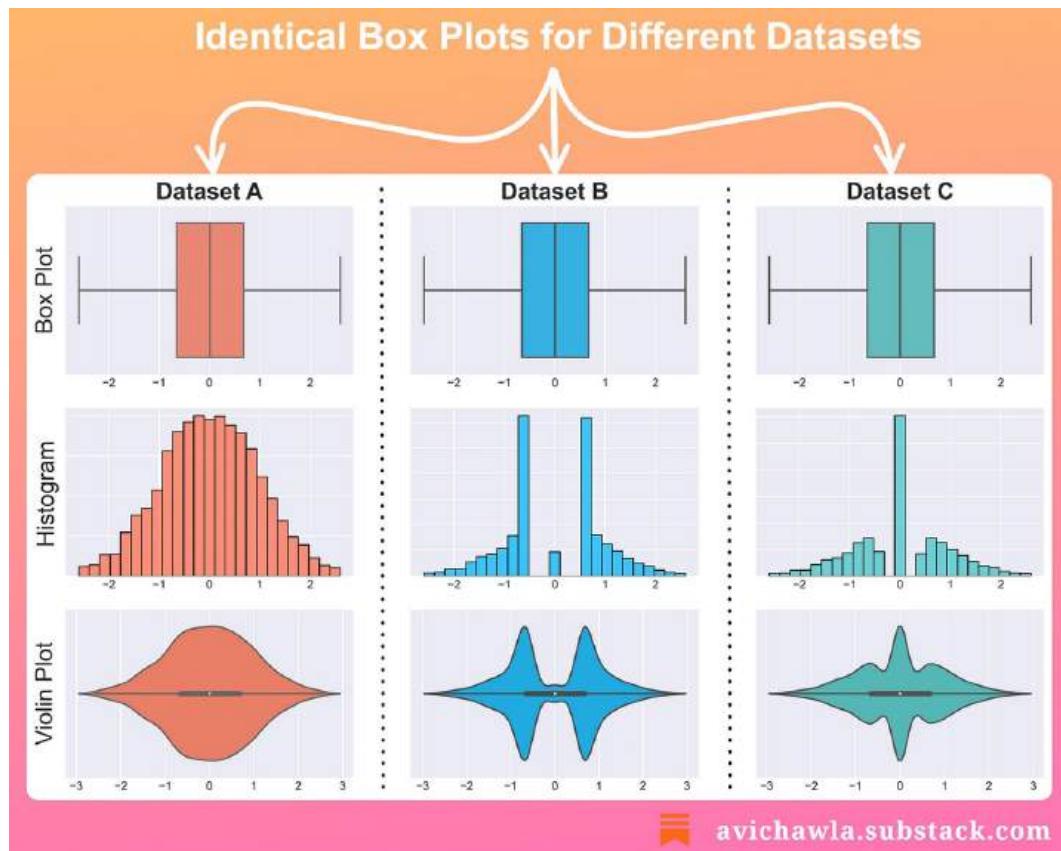
Here are some of them that will make you fall in love with Python and its versatility (even more).

Read this full list here:

<https://avichawla.substack.com/p/35-gem-py-libs>.



# Use Box Plots With Caution! They May Be Misleading.



Box plots are quite common in data analysis. But they can be misleading at times. Here's why.

A box plot is a graphical representation of just five numbers – min, first quartile, median, third quartile, and max.

Thus, two different datasets with similar five values will produce identical box plots. This, at times, can be misleading and one may draw wrong conclusions.

The takeaway is NOT that box plots should not be used. Instead, look at the underlying distribution too. Here, histograms and violin plots can help.

Lastly, always remember that when you condense a dataset, you don't see the whole picture. You are losing essential information.



# An Underrated Technique To Create Better Data Plots



While creating visualizations, there are often certain parts that are particularly important. Yet, they may not be immediately obvious to the viewer.

A good data storyteller will always ensure that the plot guides the viewer's attention to these key areas.

One great way is to zoom in on specific regions of interest in a plot. This ensures that our plot indeed communicates what we intend it to depict.

In matplotlib, you can do so using `indicate_inset_zoom()`. It adds an indicator box, that can be zoomed-in for better communication.

Find more info here: [Matplotlib docs](https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.indicate_inset_zoom.html).



# The Pandas DataFrame Extension Every Data Scientist Has Been Waiting For

The screenshot shows a Jupyter notebook interface with the title "Kanaries/pygwalker". The main area displays two stacked area charts. The top chart shows values from 0 to 50,000 over time, with a legend for season (fall, spring, summer, winter) and work yes or not (0, 1). The bottom chart shows values from 0 to 200,000 over time, also with a similar legend. To the left, a sidebar titled "Data" lists various fields: date, weekday, season, year, holiday, work yes or not, hour, day of year, static, temperature, humidity, wind speed, rainfall, registered, count, and削除 (Delete). The top of the screen has a navigation bar with File, Edit, View, Run, Kernel, Go, Table, Settings, Help, and a search bar.

Watch a video version of this post for better understanding: [Video Link](#).

PyGWalker is an open-source alternative to Tableau that transforms pandas dataframe into a tableau-style user interface for data exploration.

It provides a tableau-like UI in Jupyter, allowing you to analyze data faster and without code.

Find more info here: [PyGWalker](#).



# Supercharge Shell With Python Using Xonsh



Traditional shells have a limitation for python users. At a time, users can either run shell commands or use IPython.

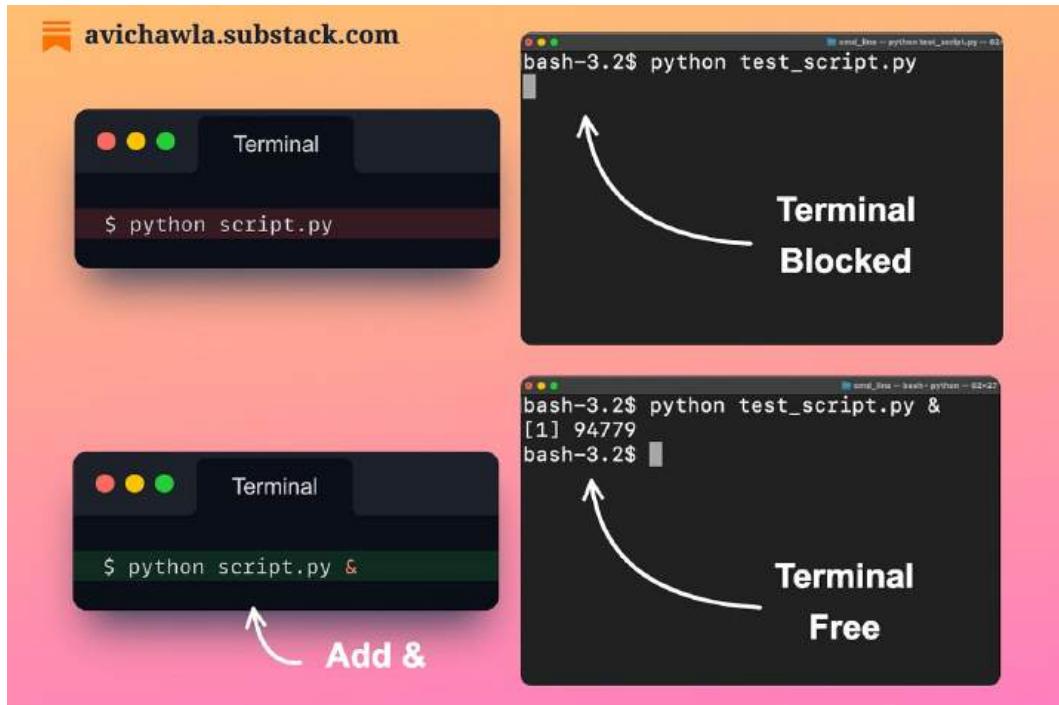
As a result, one has to open multiple terminals or switch back and forth between them in the same terminal.

Instead, try Xonsh. It combines the convenience of a traditional shell with the power of Python. Thus, you can use Python syntax as well as run shell commands in the same shell.

Find more info here: [Xonsh](#).



# Most Command-line Users Don't Know This Cool Trick About Using Terminals



**Watch a video version of this post for better understanding: [Video Link](#).**

After running a command (or script, etc.), most command-line users open a new terminal to run other commands. But that is never required.

Here's how.

When we run a program from the command line, by default, it runs in the foreground. This means you can't use the terminal until the program has been completed.

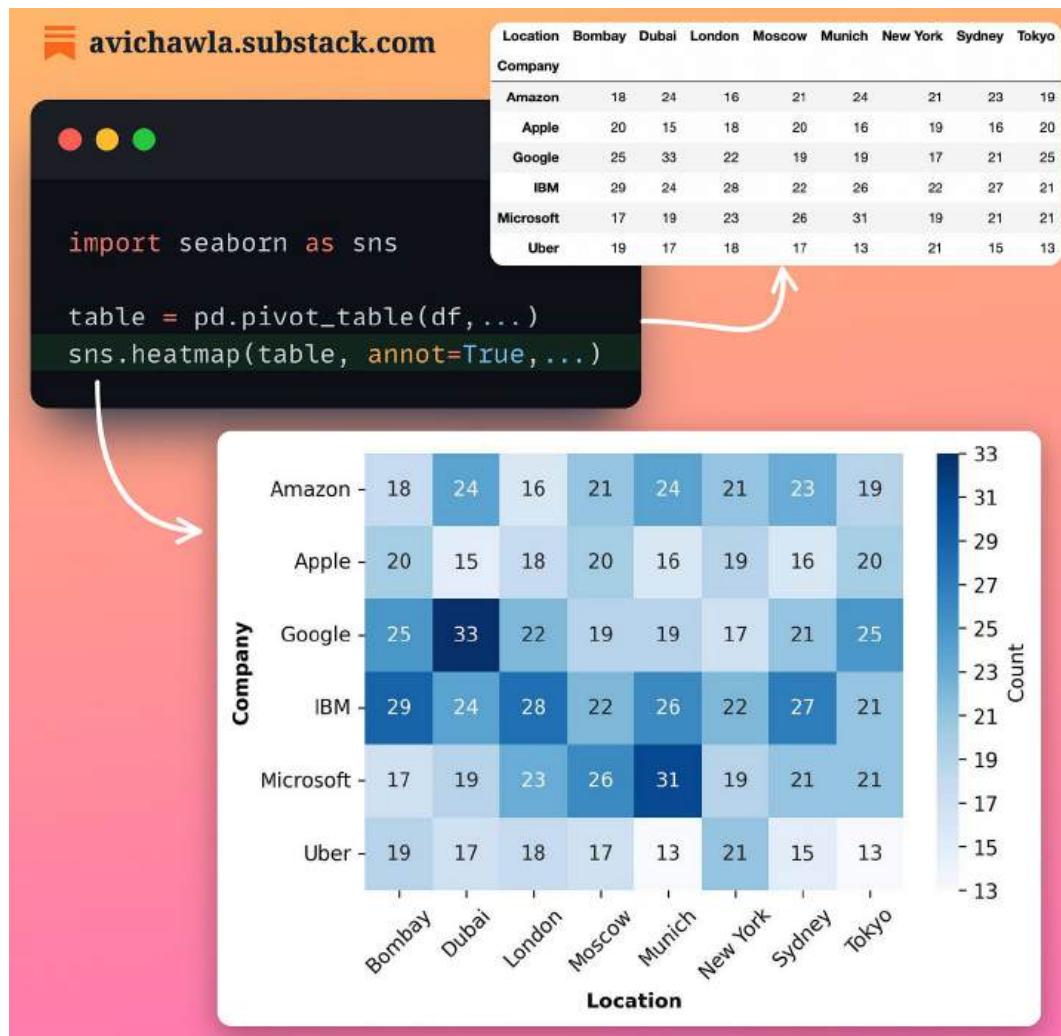
However, if you add '&' at the end of the command, the program will run in the background and instantly free the terminal.

This way, you can use the same terminal to run another command.

To bring the program back to the foreground, use the '**fg**' command.



# A Simple Trick to Make The Most Out of Pivot Tables in Pandas



Pivot tables are pretty common for data exploration. Yet, analyzing raw figures is tedious and challenging. What's more, one may miss out on some crucial insights about the data.

Instead, enrich your pivot tables with heatmaps. The color encodings make it easier to analyze the data and determine patterns.



# Why Python Does Not Offer True OOP Encapsulation

The screenshot shows a Substack article with the URL [avichawla.substack.com](https://avichawla.substack.com). The top part displays Python code defining a class `MyClass` with three types of attributes: public, protected, and private. The bottom part shows the execution of this code in a terminal, demonstrating that both protected and private members are accessible from outside the class.

```
class MyClass:
    def __init__(self):
        self.public_attr      = "I'm public"      # 0 underscores
        self._protected_attr = "I'm protected" # 1 underscore
        self.__private_attr  = "I'm private"   # 2 underscores

my_obj = MyClass()

>>> my_obj.public_attr
"I'm public"

>>> my_obj._protected_attr
"I'm protected"

>>> my_obj._MyClass__private_attr
"I'm private"
```

Annotations on the right side explain the access levels:

- Public member accessible**
- Protected member accessible**
- Private member accessible with name mangling**

Using access modifiers (public, protected, and private) is fundamental to encapsulation in OOP. Yet, Python, in some way, fails to deliver true encapsulation.

By definition, a public member is accessible everywhere. A private member can only be accessed inside the base class. A protected member is accessible inside the base class and child class(es).

But, with Python, there are no such strict enforcements.

Thus, protected members behave exactly like public members. What's more, private members can be accessed outside the class using name mangling.

As a programmer, remember that encapsulation in Python mainly relies on conventions. Thus, it is the responsibility of the programmer to follow them.



# Never Worry About Parsing Errors Again While Reading CSV with Pandas

```
In [1]: !cat file.csv
Name,Amount
Alice,$300
Bob,$1\,000
Charlie,$200
Separator appears
in value

In [2]: pd.read_csv("file.csv")
## ParserError: Error tokenizing data. C error:
## Expected 2 fields in line 3, saw 3

In [3]: import clevercsv
clevercsv.read_dataframe("file.csv")

Out[3]:
   Name  Amount
0   Alice    $300
1     Bob   $1,000
2  Charlie    $200
```

 [avichawla.substack.com](https://avichawla.substack.com)

Pandas isn't smart (yet) to read messy CSV files.

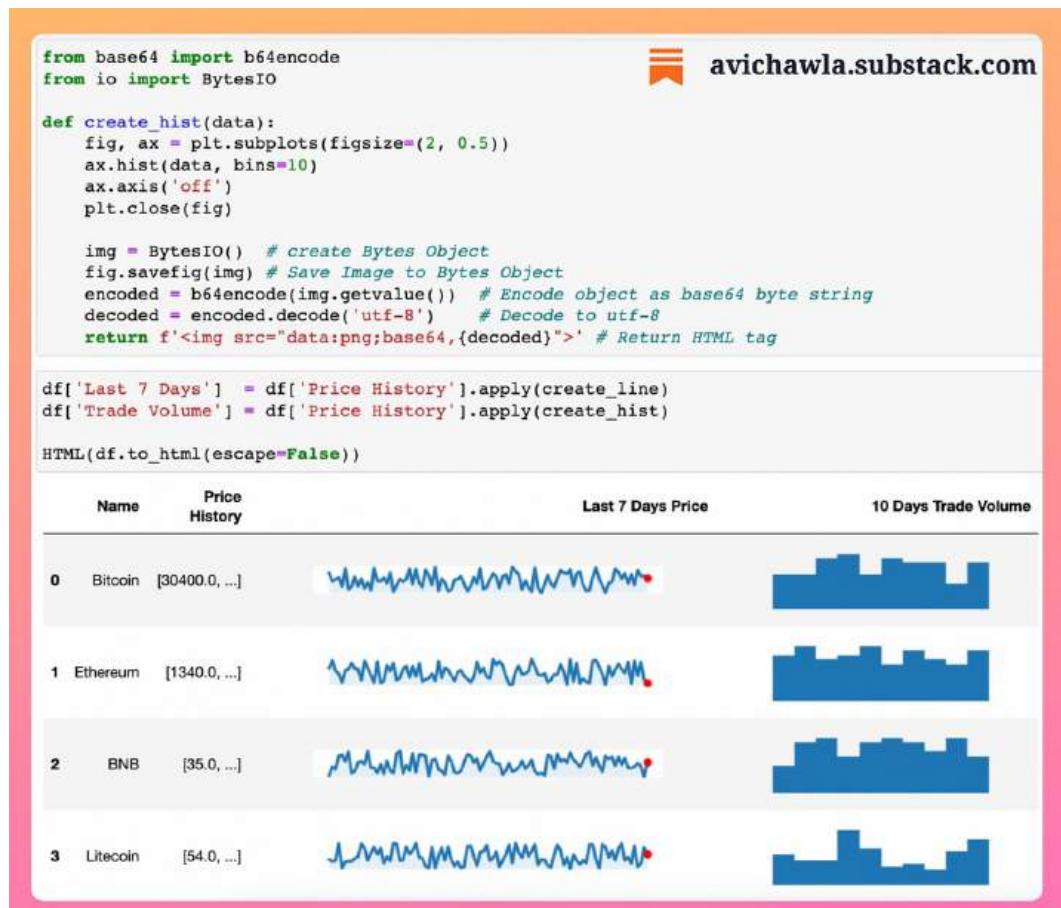
Its `read_csv` method assumes the data source to be in a standard tabular format. Thus, any irregularity in data raises parsing errors, which may require manual intervention.

Instead, try CleverCSV. It detects the format of CSVs and makes it easier to load them, saving you tons of time.

Find more info here: [CleverCSV](#).



# An Interesting and Lesser-Known Way To Create Plots Using Pandas



Whenever you print/display a DataFrame in Jupyter, it is rendered using HTML and CSS. This allows us to format the output just like any other web page.

One interesting way is to embed inline plots which appear as a column of a dataframe.

In the above snippet, we first create a plot as we usually do. Next, we return the <img> HTML tag with its source as the plot. Lastly, we render the dataframe as HTML.

Find the code for this tip here: [Notebook](#).



# Most Python Programmers Don't Know This About Python For-loops

```
for num in range(5):
    print(f"num = {num}")
    num = 10 # modified num

"""
num = 0
num = 1
num = 2
num = 3
num = 4
"""

avichawla.substack.com
```

Often when we use a for-loop in Python, we tend not to modify the loop variable inside the loop.

The impulse typically comes from acquaintance with other programming languages like C++ and Java.

But for-loops don't work that way in Python. Modifying the loop variable has no effect on the iteration.

This is because, before every iteration, Python unpacks the next item provided by iterable (`range(5)`) and assigns it to the loop variable (`num`).

Thus, any changes to the loop variable are replaced by the new value coming from the iterable.



# How To Enable Function Overloading In Python

The diagram illustrates the limitation of Python's native function overloading and how the `multipledispatch` library overcomes it.

**Top Interpreter (Native Function Overloading):**

```
def add(x:int, y:int):
    return x + y

def add(x:int, y:int, z:int):
    return x + y + z

>>> add(1,2)
TypeError: add() missing 1 required positional argument: 'z'
```

**Bottom Interpreter (Using `multipledispatch`):**

```
from multipledispatch import dispatch

@dispatch(int, int)
def add(x, y):
    return x + y

@dispatch(int, int, int)
def add(x, y, z):
    return x + y + z

>>> add(1,2)           >>> add(1,2,3)
3                         6
```

**Text Labels:**

- "python interpreter only considers the latest definition of `add()` function"
- "dispatch decorator enables function overloading"

Python has no native support for function overloading. Yet, there's a quick solution to it.

Function overloading (having multiple functions with the same name but different number/type of parameters) is one of the core ideas behind polymorphism in OOP.

But if you have many functions with the same name, python only considers the latest definition. This restricts writing polymorphic code.

Despite this limitation, the `dispatch` decorator allows you to leverage function overloading.

Find more info here: [Multipledispatch](#).



# Generate Helpful Hints As You Write Your Pandas Code

The screenshot shows a Jupyter Notebook interface with three code cells. The first cell imports Dovpanda. The second cell iterates over a DataFrame using `iterrows()`. A hint box appears, stating: "df.iterrows is not recommended. Essentially it is very similar to iterating the rows of the frames in a loop. In the majority of cases, there are better alternatives that utilize pandas' vector operation". The third cell applies a function to a DataFrame column using `apply(func)`. A similar hint box appears, stating: "df.apply is not recommended. Essentially it is very similar to iterating the rows of the frames in a loop. In the majority of cases, there are better alternatives that utilize pandas' vector operation". The fourth cell concatenates two DataFrames using `pd.concat([df, df])`. A hint box appears, stating: "All dataframes have the same columns and same number of rows. Pay attention, your axis is 0 which concatenates vertically". The fifth cell concatenates two DataFrames using `pd.concat([df, df])` again, with a hint box stating: "After concatenation you have duplicated indices - pay attention". The notebook header shows the URL [avichawla.substack.com](http://avichawla.substack.com).

```
In [4]: import dovpanda
In [5]: iter_df = df.iterrows()
In [6]: df["new_col"] = df.apply(apply_func)
In [7]: merged_df = pd.concat((df, df))
```

When manipulating a dataframe, at times, one may be using unoptimized methods. What's more, errors introduced into the data can easily go unnoticed.

To get hints and directions about your data/code, try Dovpanda. It works as a companion for your Pandas code. As a result, it gives suggestions/warnings about your data manipulation steps.

P.S. When you will import Dovpanda, you will likely get an error. Ignore it and proceed with using Pandas. You will still receive suggestions from Dovpanda.

Find more info here: [Dovpandas](#).



# Speedup NumPy Methods 25x With Bottleneck

The image shows a Mac desktop with two terminal windows side-by-side. The top window has the URL "avichawla.substack.com" in the title bar. It contains the following Python code:

```
import bottleneck as bn
import numpy as np

arr = np.random.random((1000,))
```

The bottom window has two tabs: "numpy.py" and "bottleneck.py". Both tabs show command-line sessions. Arrows point from the NumPy run-times to the corresponding Bottleneck run-times.

NumPy Method	Run-time	Bottleneck Method	Run-time	Performance Boost
<code>&gt;&gt;&gt; np.sum(arr)</code>	<code>## Run-time: 870 µs</code>	<code>&gt;&gt;&gt; bn.nansum(arr)</code>	<code>## Run-time: 33.9 µs (25x Faster)</code>	(25x Faster)
<code>&gt;&gt;&gt; np.mean(arr)</code>	<code>## Run-time: 477 µs</code>	<code>&gt;&gt;&gt; bn.nanmean(arr)</code>	<code>## Run-time: 21 µs (22x Faster)</code>	(22x Faster)
<code>&gt;&gt;&gt; np.std(arr)</code>	<code>## Run-time: 687 µs</code>	<code>&gt;&gt;&gt; bn.nanstd(arr)</code>	<code>## Run-time: 175 µs (4x Faster)</code>	(4x Faster)
<code>&gt;&gt;&gt; np.median(arr)</code>	<code>## Run-time: 1.58 ms</code>	<code>&gt;&gt;&gt; bn.nanmedian(arr)</code>	<code>## Run-time: 0.43 ms (4x Faster)</code>	(4x Faster)
<code>&gt;&gt;&gt; np.max(arr)</code>	<code>## Run-time: 1.26 ms</code>	<code>&gt;&gt;&gt; bn.nanmax(arr)</code>	<code>## Run-time: 0.46 ms (3x Faster)</code>	(3x Faster)

NumPy's methods are already highly optimized for performance. Yet, here's how you can further speed them up.

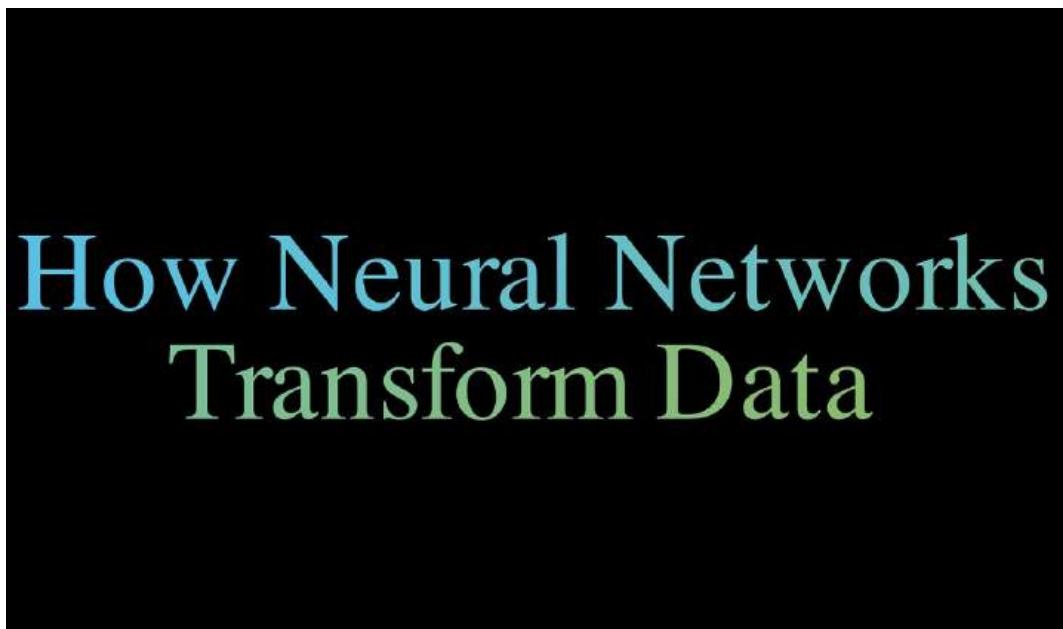
Bottleneck provides a suite of optimized implementations of NumPy methods.

Bottleneck is especially efficient for arrays with NaN values where performance boost can reach up to 100-120x.

Find more info here: [Bottleneck](#).



## Visualizing The Data Transformation of a Neural Network



If you struggle to comprehend how a neural network learns complex non-linear data, I have created an animation that will surely help.

Please find the video here: [\*\*Neural Network Animation\*\*](#).

For linearly inseparable data, the task boils down to projecting the data to a space where it becomes linearly separable.

Now, either you could do this manually by adding relevant features that will transform your data to a linear separable form. Consider concentric circles for instance. Passing a square of  $(x,y)$  coordinates as a feature will do this job.

But in most cases, the transformation is unknown or complex to figure out. Thus, non-linear activation functions are considered the best bet, and a neural network is allowed to figure out this "non-linear to linear transformation" on its own.

As shown in the animation, if we tweak the neural network by adding a 2D layer right before the output, and visualize this transformation, we see that the neural network has learned to linearly separate the data. We add a layer 2D because it is easy to visualize.

This linearly separable data can be easily classified by the last layer. To put it another way, the last layer is analogous to a logistic regression model which is given a linear separable input.

The code for this visualization experiment is available here: [GitHub](#).



# Never Refactor Your Code Manually Again. Instead, Use Sourcery!

**Before Refactoring**

```
def is_special_number(number):
    if number == 7:
        return True
    elif number == 18:
        return True
    else:
        return False
```

Command Line

```
$ sourcery review --in-place my_code.py
```

**After Refactoring**

```
def is_special_number(number):
    return number in [7, 18]
```

[linkedin.com/in/avi-chawla](https://linkedin.com/in/avi-chawla)

Refactoring code is an important step in pipeline development. Yet, manual refactoring takes additional time for testing as one might unknowingly introduce errors.

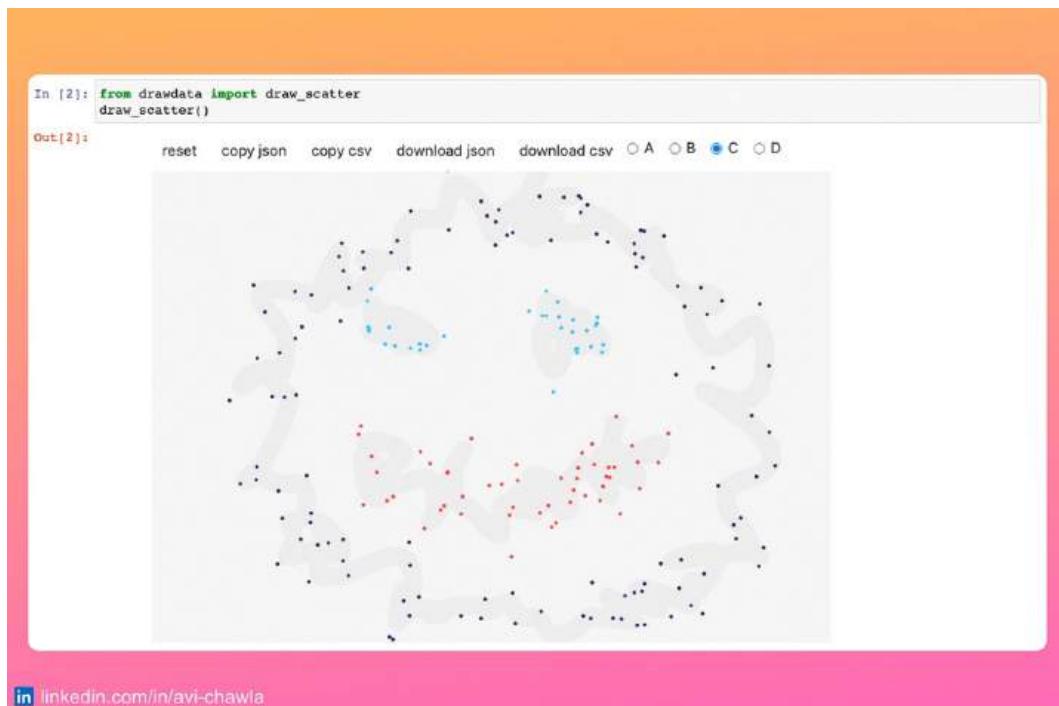
Instead, use Sourcery. It's an automated refactoring tool that makes your code elegant, concise, and Pythonic in no time.

With Sourcery, you can refactor code from the command line, as an IDE plugin in VS Code and PyCharm, pre-commit, etc.

Find more info here: [Sourcery](#).



# Draw The Data You Are Looking For In Seconds



Please watch a video version of this post for better understanding: [Video Link](#).

Often when you want data of some specific shape, programmatically generating it can be a tedious and time-consuming task.

Instead, use drawdata. This allows you to draw any 2D dataset in a notebook and export it. Besides a scatter plot, it can also create histogram and line plot

Find more info here: [Drawdata](#).



# Style Matplotlib Plots To Make Them More Attractive



Matplotlib offers close to 50 different styles to customize the plot's appearance.

To alter the plot's style, select a style from `plt.style.available` and create the plot as you originally would.

Find more info about styling here: [Docs](#).



## Speed-up Parquet I/O of Pandas by 5x

file.parquet:  
32M rows

```
pandas.py
import pandas as pd
df = pd.read_parquet("file.parquet")
# Run-time: 41s
```

fastparquet.py

```
from fastparquet import ParquetFile
pf = ParquetFile('file.parquet')
df = pf.to_pandas()
# Run-time: 8.1s
```

5x Faster

linkedin.com/in/avi-chawla

Dataframes are often stored in parquet files and read using Pandas' `read_parquet()` method.

Rather than using Pandas, which relies on a single-core, use `fastparquet`. It offers immense speedups for I/O on parquet files using parallel processing.

Find more info here: [Docs](#).



## 40 Open-Source Tools to Supercharge Your Pandas Workflow



Pandas receives over [3M downloads per day](#). But 99% of its users are not using it to its full potential.

I discovered these open-source gems that will immensely supercharge your Pandas workflow the moment you start using them.

Read this list here: <https://avichawla.substack.com/p/37-open-source-tools-to-supercharge-pandas>.



# Stop Using The Describe Method in Pandas. Instead, use Skimpy.

The screenshot shows a Jupyter Notebook cell with the following code:

```
from skimpy import skim
skim(df)
```

An arrow points from this code to a detailed data summary table titled "skimpy summary". The table is organized into several sections based on data types:

- Data Summary**:

dataframe	Values
Number of rows	1000
Number of columns	10
- Data Types**:

Column Type	Count
float64	3
category	2
datetime64	2
int64	1
bool	1
string	1
- Categories**:

Categorical Variables
class
location
- number**:

column_name	NA	NA %	mean	sd	p0	p25	p75	p100	hist
length	0	0	0.5	0.36	1.6e-06	0.13	0.86	1	
width	0	0	2	1.9	0.0021	0.6	3	14	
depth	0	0	10	3.2	2	8	12	20	
rnd	120	12	-0.02	1	-2.8	-0.74	0.66	3.7	
- category**:

column_name	NA	NA %	ordered	unique
class	0	0	False	2
location	1	0.1	False	5
- datetime**:

column_name	NA	NA %	first	last	frequency
date	0	0	2018-01-31	2101-04-30	M
date_no_freq	3	0.3	1992-01-05	2023-03-04	None
- string**:

column_name	NA	NA %	words per row	total words
text	6	0.6		5.8
				5800
- bool**:

column_name	true	true rate	hist
booly_col	520	0.52	

At the bottom left is a LinkedIn link: [in](https://linkedin.com/in/avi-chawla) linkedin.com/in/avi-chawla

Supercharge the describe method in Pandas.

Skimpy is a lightweight tool for summarizing Pandas dataframes. In a single line of code, it generates a richer statistical summary than the `describe()` method.

What's more, the summary is grouped by datatypes for efficient analysis. You can use Skimpy from the command line too.

Find more info here: [Docs](#).



# The Right Way to Roll Out Library Updates in Python

**Add decorator**

```
my_library.py
from deprecated import deprecated
@deprecated(reason="old_function will be \
    deprecated in the next \
    release. Use new_function.")
def old_function():
    ...
```

**Prints warning**

```
project.py
old_value = old_function()
DeprecationWarning: Call to deprecated function
old_function. (old_function will be deprecated
in the next release. Use new_function.)
```

[linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

While developing a library, authors may decide to remove some functions/methods/classes. But instantly rolling the update without any prior warning isn't a good practice.

This is because many users may still be using the old methods and they may need time to update their code.

Using the **deprecated** decorator, one can convey a warning to the users about the update. This allows them to update their code before it becomes outdated.

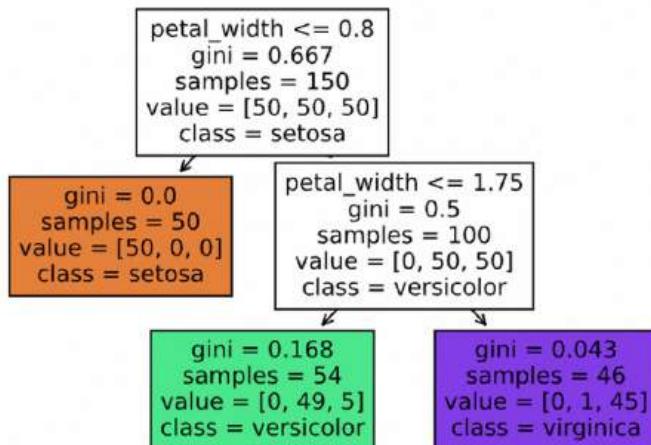
Find more info here: [GitHub](#).



# Simple One-Liners to Preview a Decision Tree Using Sklearn

```
my_tree = DecisionTreeClassifier()  
my_tree.fit(X, y)  
  
from sklearn.tree import plot_tree, export_text  
  
plot_tree(my_tree, feature_names=features,  
          class_names=classes, filled=True)
```

Method 1



```
print(export_text(my_tree, feature_names=features))
```

Method 2

```
--- petal_width <= 0.80  
|--- class: setosa  
--- petal_width >  0.80  
|--- petal_width <= 1.75  
|   |--- class: versicolor  
|   |--- petal_width >  1.75  
|       |--- class: virginica
```

[in](https://www.linkedin.com/in/avi-chawla) linkedin.com/in/avi-chawla

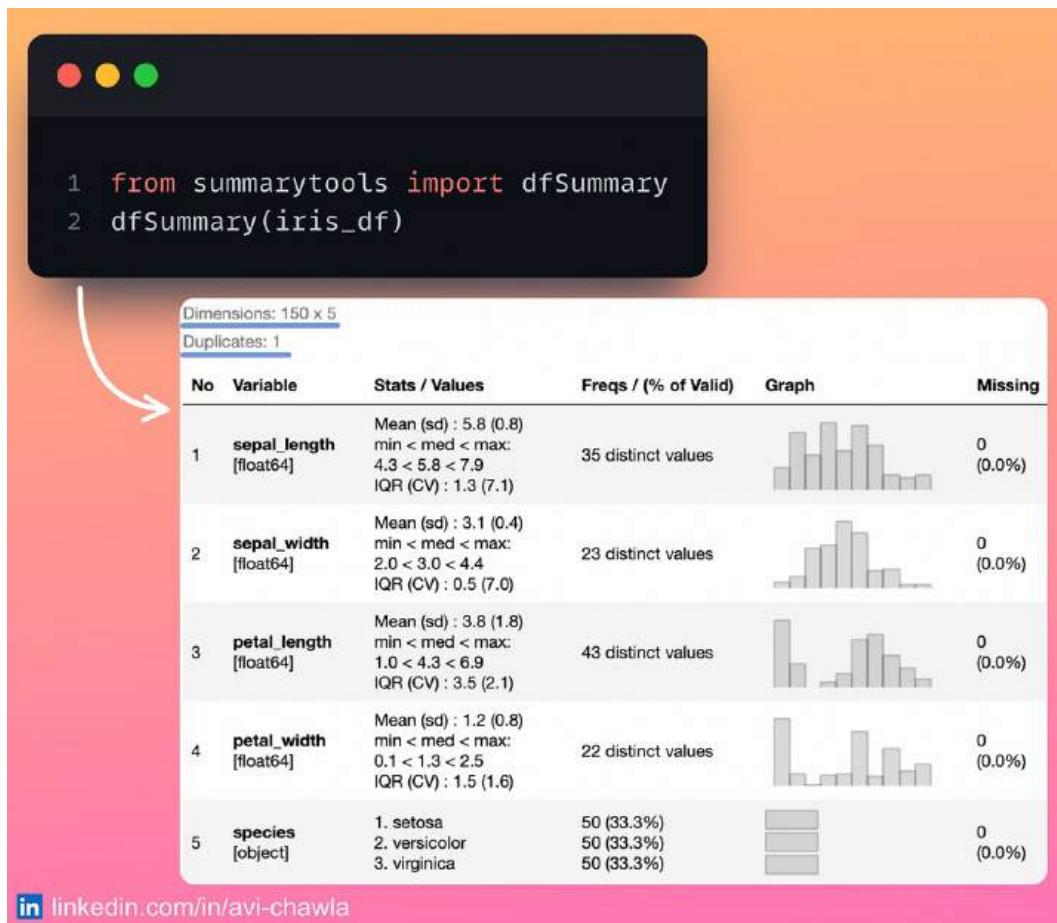
If you want to preview a decision tree, sklearn provides two simple methods to do so.

1. [plot\\_tree](#) creates a graphical representation of a decision tree.
2. [export\\_text](#) builds a text report showing the rules of a decision tree.

This is typically used to understand the rules learned by a decision tree and gaining a better understanding of the behavior of a decision tree model.



# Stop Using The Describe Method in Pandas. Instead, use Summarytools.



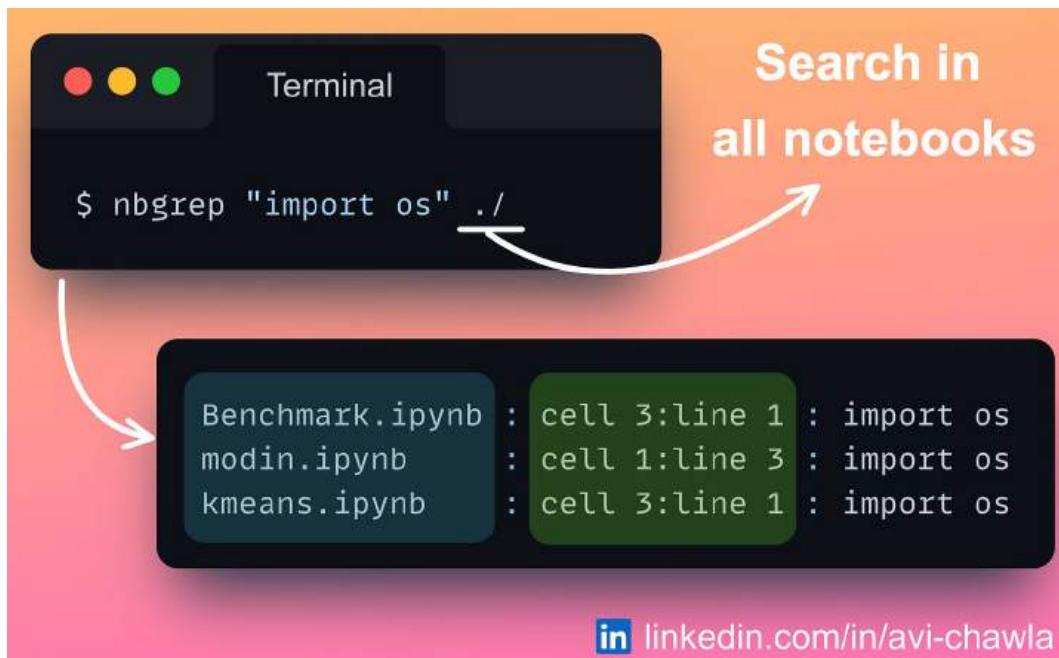
Summarytools is a simple EDA tool that gives a richer summary than **describe()** method. In a single line of code, it generates a standardized and comprehensive data summary.

The summary includes column statistics, frequency, distribution chart, and missing stats.

Find more info here: [Summary Tools](#).



# Never Search Jupyter Notebooks Manually Again To Find Your Code



Have you ever struggled to recall the specific Jupyter notebook in which you wrote some code? Here's a quick trick to save plenty of manual work and time.

**nbcommands** provides a bunch of commands to interact with Jupyter from the terminal.

For instance, you can search for code, preview a few cells, merge notebooks, and many more.

Find more info here: [GitHub](#).



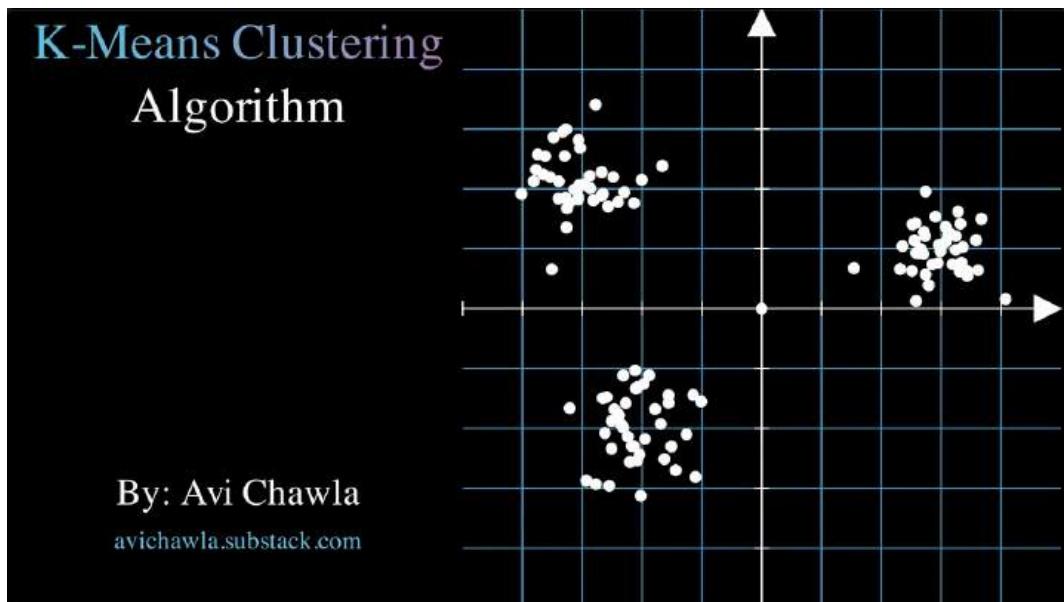
# F-strings Are Much More Versatile Than You Think



Here are 6 lesser-known ways to format/convert a number using f-strings. What is your favorite f-string hack?



# Is This The Best Animated Guide To KMeans Ever?



Have you ever struggled with understanding KMeans? How it works, how are the data points assigned to a centroid, or how do the centroids move?

If yes, let me help.

I created a beautiful animation using Manim to help you build an intuitive understanding of the algorithm.

Please find this video here: [Video Link](#).



# An Effective Yet Underrated Technique To Improve Model Performance

**Original Images**

```
import imgaug.augmenters as iaa

seq = iaa.Sequential([
    iaa.Fliplr(0.5), # horizontal flip
    iaa.Rotate((-40,40)), # Rotate
    ...])

images_aug = seq(images=images)
```

**Augmented Images**

[in](https://www.linkedin.com/in/avi-chawla) linkedin.com/in/avi-chawla

Robust ML models are driven by diverse training data. Here's a simple yet highly effective technique that can help you create a diverse dataset and increase model performance.

One way to increase data diversity is using data augmentation.

The idea is to create new samples by transforming the available samples. This can prevent overfitting, improve performance, and build robust models.

For images, you can use imgaug (linked in comments). It provides a variety of augmentation techniques such as flipping, rotating, scaling, adding noise to images, and many more.

Find more info: [ImgAug](#).



# Create Data Plots Right From The Terminal

```
>>> from bashplotlib.histogram import plot_hist
>>> np_arr = np.random.normal(size=1000)
>>> plot_hist(np_arr, bincount=50)

54|          0
51|      oo oo
48|      000 000 0
45|      000 000 0
43|      0000 000 0
40|      0000000000 00
37|      00 000000000000
34|      00 000000000000
31|      000000000000000
29|      000000000000000
26|      00000000000000000
23|      00000000000000000
20|      00000000000000000
17|      00000000000000000 0
15|      00000000000000000 0
12|      00000000000000000 0
 9|      00000000000000000
 6|      00 00000000000000000 0
 3|      o   o 00000000000000000
 1|      o   o 00000000000000000 0
```

 [linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

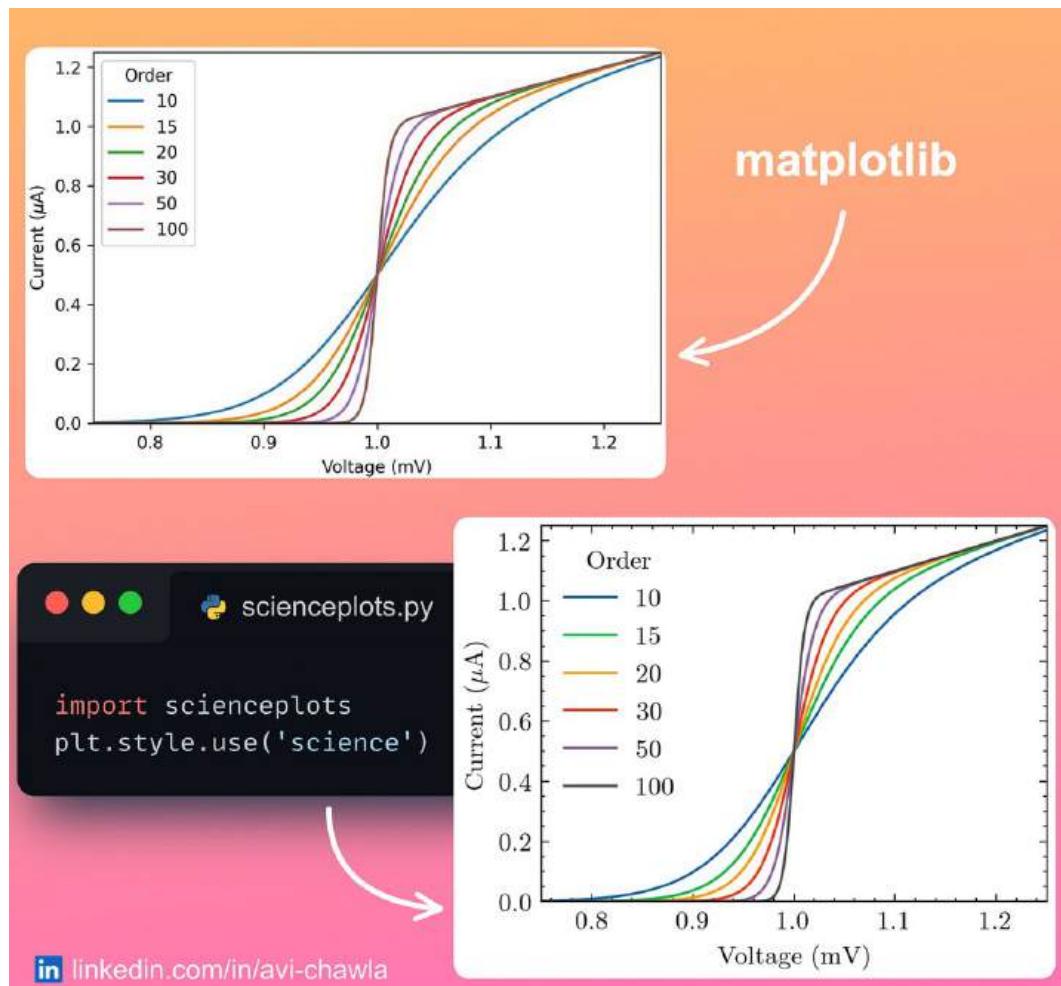
Visualizing data can get tough when you don't have access to a GUI. But here's what can help.

Bashplotlib offers a quick and easy way to make basic plots right from the terminal. Being pure python, you can quickly install it anywhere using pip and visualize your data.

Find more info here: [Bashplotlib](#).



# Make Your Matplotlib Plots More Professional



The default matplotlib plots are pretty basic in style and thus, may not be the apt choice always. Here's how you can make them appealing.

To create professional-looking and attractive plots for presentations, reports, or scientific papers, try Science Plots.

Adding just two lines of code completely transforms the plot's appearance.

Find more info here: [GitHub](#).



## 37 Hidden Python Libraries That Are Absolute Gems



I reviewed 1,000+ Python libraries and discovered these hidden gems I never knew even existed.

Here are some of them that will make you fall in love with Python' and its versatility (even more).

Read this list here: <https://avichawla.substack.com/p/gem-libraries>.



# Preview Your README File Locally In GitHub Style

The image shows a terminal window on a Mac OS X desktop. The title bar says "Terminal". Inside, the command "\$ grip -b" is typed. A white arrow points from the text "Preview README in browser" on the left side of the image towards the terminal window. To the right of the terminal is a screenshot of a web browser displaying a README page titled "README.md". The page includes social sharing links (GitHub, Medium, Substack, LinkedIn) and a preview of the content which features various data science icons like Python, neural networks, and charts. Below the preview, the URL "avichawla.substack.com" is visible. Further down, the section "Daily Dose of Data Science" is shown with a brief description and a link to download the repository via Git.

Preview README in browser

in [linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

Please watch a video version for better understanding: [Video Link](#).

Have you ever wanted to preview a README file before committing it to GitHub? Here's how to do it.

Grip is a command-line tool that allows you to render a README file as it will appear on GitHub. This is extremely useful as sometimes one may want to preview the file before pushing it to GitHub.

What's more, editing the README instantly reflects in the browser without any page refresh.

Read more: [Grip](#).



# Pandas and NumPy Return Different Values for Standard Deviation. Why?

```
Std-dev.py
```

```
import numpy as np
import pandas as pd

X = np.arange(20)
df = pd.DataFrame(X)

print(f"NumPy : {np.std(X)}")
print(f"Pandas: {df.std()}")
```

std-dev using  
Pandas and  
NumPy

different output

NumPy : 5.766  
Pandas: 5.916

[in](https://linkedin.com/in/avi-chawla) linkedin.com/in/avi-chawla

Pandas assumes that the data is a sample of the population and that the obtained result can be biased towards the sample.

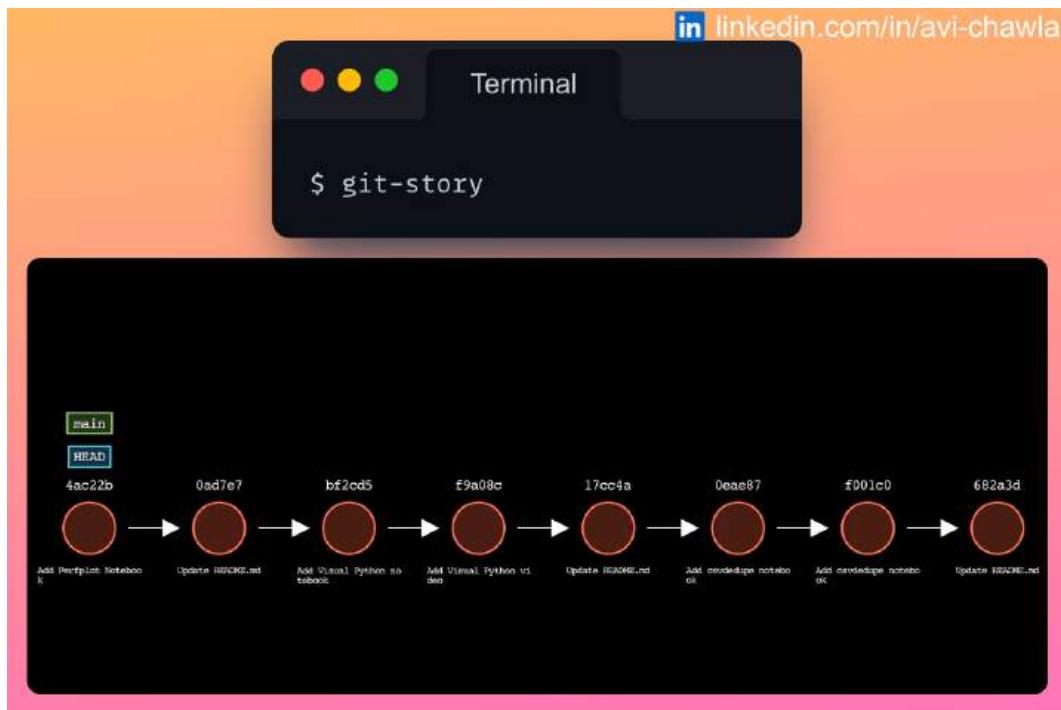
Thus, to generate an unbiased estimate, it uses  $(n-1)$  as the dividing factor instead of  $n$ . In statistics, this is also known as Bessel's correction.

NumPy, however, does not make any such correction.

Find more info here: [Bessel's correction](#).



# Visualize Commit History of Git Repo With Beautiful Animations



As the size of your project grows, it can get difficult to comprehend the Git tree.

Git-story is a command line tool to create elegant animations for your git repository.

It generates a video that depicts the commits, branches, merges, HEAD commit, and many more. Find more info in the comments.

Please watch a video version of this post here: [Video](#).

Read more: [Git-story](#).



# Perfplot: Measure, Visualize and Compare Run-time With Ease



Here's an elegant way to measure the run-time of various Python functions.

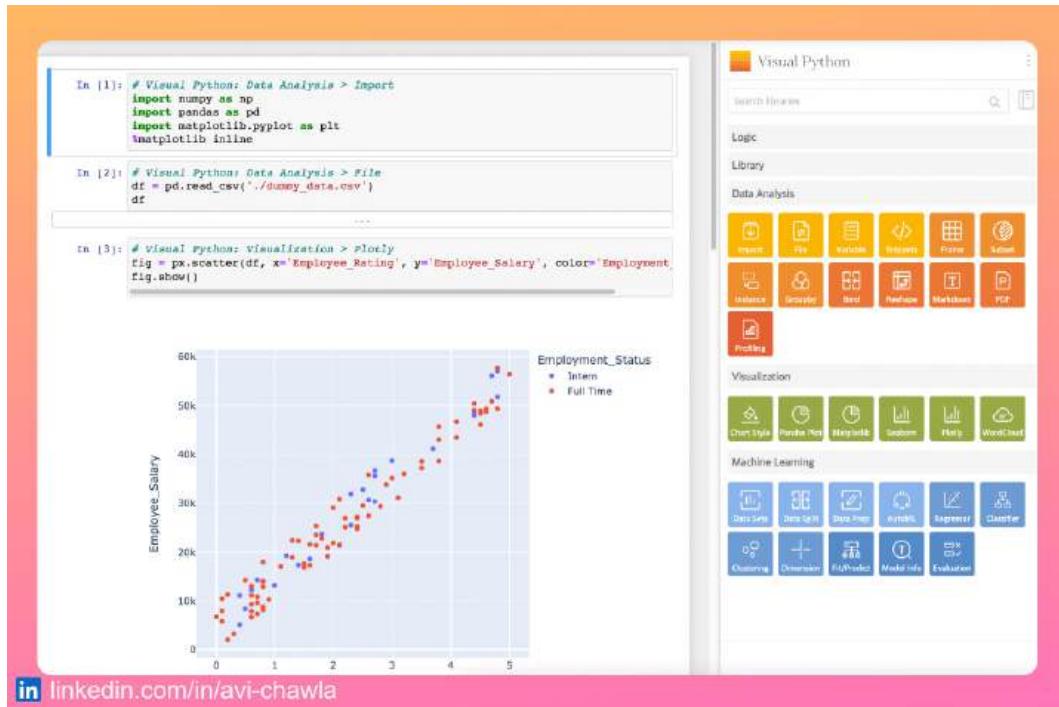
Perfplot is a tool designed for quick run-time comparisons of many functions/algorithms.

It extends Python's `timeit` package and allows you to quickly visualize the run-time in a clear and informative way.

Find more info: [Perfplot](#).



# This GUI Tool Can Possibly Save You Hours Of Manual Work



Please watch a video version of this post for better understanding: [Link](#).

This is indeed one of the coolest and most useful Jupyter notebook-based data science tools.

Visual Python is a GUI-based python code generator. Using this, you can easily eliminate writing code for many repetitive tasks. This includes importing libraries, I/O, Pandas operations, plotting, etc.

Moreover, with the click of a couple of buttons, you can import the code for many ML-based utilities. This covers sklearn models, evaluation metrics, data splitting functions, and many more.

Read more: [Visual Python](#).



# How Would You Identify Fuzzy Duplicates In A Data With Million Records?

The slide illustrates the process of identifying fuzzy duplicates in a dataset. It starts with a table of 8 rows of address data, where several rows represent different variations of the same person's name (e.g., Daniel Lopez vs. Daniel NaN). Brackets on the right side group these into three clusters. Below this, a 'Command Line' section shows a command to run the `csvdedupe` tool on a file named `input.csv`. The final part of the slide shows the output of this command: a new table where each row is assigned a 'Cluster ID'. Rows 0 and 1 are grouped together under Cluster ID 0, while rows 2 and 3 are grouped under Cluster ID 1, and so on. A callout points to the first two rows in the clustered table, which are highlighted with blue circles around their cluster IDs.

	First_Name	Last_Name	Address	Phone
0	Daniel	Lopez	719 Greene St. East Rhonda	9371184929
1	Daniel	Nan	719 Green Street East Rhoda	93711-84929
2	Alan	Martin	982 Carol Harbors Apart.	7481919235
3	Alan Martin	NaN	982 Carol Apartments	748-191-9235
4	Philip	Owens	2578 Banks Ford	869-6922x9581
5	Shannon	White	USCGC Molina	(150)082-7982
6	Julia	Anderson	09162 Mason Mnts.	698-1590x3236
7	Juliya	Anderrson	9162 Mason Street Mountain	69815903236

	Cluster ID	First_Name	Last_Name	Address	Phone
0	0	Daniel	Lopez	719 Greene St. East Rhonda	9371184929
1	0	Daniel	nan	719 Green Street East Rhoda	93711-84929
2	1	Alan	Martin	982 Carol Harbors Apart.	7481919235
3	1	Alan Martin	nan	982 Carol Apartments	748-191-9235
4	2	Philip	Owens	2578 Banks Ford	869-6922x9581
5	3	Shannon	White	USCGC Molina	(150)082-7982
6	4	Julia	Anderson	09162 Mason Mnts.	698-1590x3236
7	4	Juliya	Anderrson	9162 Mason Street Mountain	69815903236

Imagine you have over a million records with fuzzy duplicates. How would you identify potential duplicates?

The naive approach of comparing every pair of records is infeasible in such cases. That's over  $10^{12}$  comparisons ( $n^2$ ). Assuming a speed of 10,000 comparisons per second, it will take roughly 3 years to complete.

The `csvdedupe` tool (linked in comments) solves this by cleverly reducing the comparisons. For instance, comparing the name "Daniel" to "Philip" or "Shannon" to "Julia" makes no sense. They are guaranteed to be distinct records.



Thus, it groups the data into smaller buckets based on rules. One rule could be to group all records with the same first three letters in the name.

This way, it drastically reduces the number of comparisons with great accuracy.

Read more: [csvdedupe](#).



# Stop Previewing Raw DataFrames. Instead, Use DataTables.

The screenshot shows a Jupyter Notebook cell with the following code:

```
In [1]: import pandas as pd  
from jupyter_datatables import init_datatables_mode  
  
In [2]: init_datatables_mode()  
  
In [3]: pd.read_csv("employee_dataset.csv")
```

Below the code, there is a search bar with the value "andrade". The main area displays a preview of the "employee\_dataset.csv" file using the Jupyter-DataTables extension. The preview includes:

- Buttons for Print, CSV, Show 10 entries, and a search input field.
- A table header with columns: Name, Company\_Name, Employee\_Job\_Title, Employee\_City, and Employee\_Country.
- Below the header, there are four small bar charts representing the distribution of data for each column.
- A data table showing 10 rows of sample data from 69 total entries. The columns correspond to the headers above.
- Pagination controls at the bottom showing page 1 of 7.
- A note at the bottom left: "Showing 1 to 10 of 69 entries (filtered from 1,000 total entries)".
- An output message: "Out[3]: Sample size: 1,000 out of 1,000".
- A LinkedIn profile link: "in linkedin.com/in/avi-chawla".

After loading any dataframe in Jupyter, we preview it. But it hardly tells anything about the data.

One has to dig deeper by analyzing it, which involves simple yet repetitive code.

Instead, use [Jupyter-DataTables](#).

It supercharges the default preview of a DataFrame with many common operations. This includes sorting, filtering, exporting, plotting column distribution, printing data types, and pagination.

Please view a video version here for better understanding: [Post Link](#).



# 🚀 A Single Line That Will Make Your Python Code Faster

```
def func_without_numba():
    result = []
    for a in range(10000):
        for b in range(10000):
            if (a+b)%11 == 0:
                result.append((a,b))

func_without_numba()
# Run-time: 8.34 sec
```

~33x Faster

```
from numba import njit

@njit
def func_with_numba():
    # same code

func_with_numba()
# Run-time: 0.25 sec
```

[in linkedin.com/in/avi-chawla](https://linkedin.com/in/avi-chawla)

If you are frustrated with Python's run-time, here's how a single line can make your code blazingly fast.

Numba is a just-in-time (JIT) compiler for Python. This means that it takes your existing python code and generates a fast machine code (at run-time).

Thus, post compilation, your code runs at native machine code speed. Numba works best on code that uses NumPy arrays and functions, and loops.

Get Started: [Numba Guide](#).



## Prettify Word Clouds In Python

The screenshot shows a Jupyter Notebook interface. On the left, a code cell contains the following Python code:

```
from wordcloud import WordCloud

wc = WordCloud().generate(text)
```

On the right, the resulting word cloud image is displayed. The words are colored in various shades of blue, green, and yellow, centered around the word "Python".

The screenshot shows a Jupyter Notebook interface. On the left, a code cell contains the following Python code:

```
from PIL import Image

mask = Image.open("pylogo.png")

wc = WordCloud(mask=mask,
               contour_width=4,
               ...)
wc.generate(text)
```

On the right, the resulting word cloud image is displayed, shaped like the Python logo. The words are colored in various shades of blue, green, and yellow, centered around the word "Python".

[in linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

If you use word clouds often, here's a quick way to make them prettier.

In Python, you can easily alter the shape and color of a word cloud. By supplying a mask image, the resultant world cloud will take its shape and appear fancier.

Find more info here: [Notebook Link](#).



# How to Encode Categorical Features With Many Categories?

```
import category_encoders as ce

enc = ce.BinaryEncoder(cols=['class'])

enc.fit_transform(data["class"])

class_0    class_1    class_2
0          0          0          1
1          0          1          0
2          0          1          1
3          1          0          0
4          0          0          1
```

	gender	class
0	Male	A
1	Female	B
2	Male	C
3	Female	D
4	Female	A

} **Binary**

We often encode categorical columns with one-hot encoding. But the feature matrix becomes sparse and unmanageable with many categories.

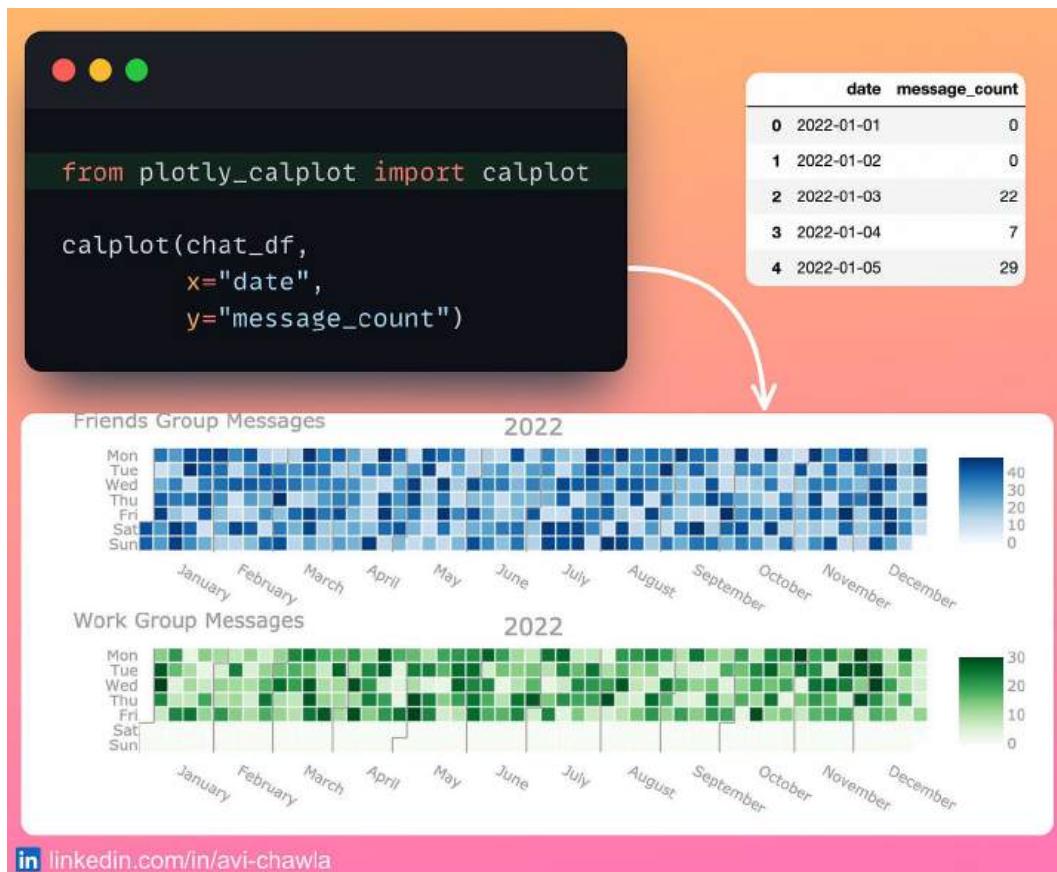
The category-encoders library provides a suite of encoders specifically for categorical variables. This makes it effortless to experiment with various encoding techniques.

For instance, I used its binary encoder above to represent a categorical column in binary format.

Read more: [Documentation](#).



# Calendar Map As A Richer Alternative to Line Plot



Ever seen one of those calendar heat maps? Here's how you can create one in two lines of Python code.

A calendar map offers an elegant way to visualize daily data. At times, they are better at depicting weekly/monthly seasonality in data instead of line plots. For instance, imagine creating a line plot for "Work Group Messages" above.

To create one, you can use "plotly\_calplot". Its input should be a DataFrame. A row represents the value corresponding to a date.

Read more: [Plotly Calplot](#).



# 10 Automated EDA Tools That Will Save You Hours Of (Tedious) Work

# 10 Automated EDA

# Tools That Will

# Save You Hours

# Of (Tedious) Work

Most steps in a data analysis task stay the same across projects. Yet, manually digging into the data is tedious and time-consuming, which inhibits productivity.

Here are 10 EDA tools that automate these repetitive steps and profile your data in seconds.

**Please find this full document in my LinkedIn post: [Post Link](#).**



# Why KMeans May Not Be The Apt Clustering Algorithm Always

in [linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

KMeans is a popular clustering algorithm. Yet, its limitations make it inapplicable in many cases.

For instance, KMeans clusters the points purely based on locality from centroids. Thus, it can create wrong clusters when data points have arbitrary shapes.

Among the many possible alternatives is DBSCAN, which is a density-based clustering algorithm. Thus, it can identify clusters of arbitrary shape and size.

This makes it robust to data with non-spherical clusters and varying densities. Find more info in the comments.

Find more here: [Sklearn Guide](#).



# Converting Python To LaTeX Has Possibly Never Been So Simple

```
import latexify
import math

@latexify.function      Add decorator
def roots(a, b, c):
    return (-b + math.sqrt(b**2 - 4*a*c)) / (2*a)
```

roots

$$\text{roots}(a, b, c) = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

```
@latexify.function
def fib(n):
    if n<2:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

fib

$$\text{fib}(n) = \begin{cases} 1, & \text{if } n < 2 \\ \text{fib}(n - 1) + \text{fib}(n - 2), & \text{otherwise} \end{cases}$$

[in linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

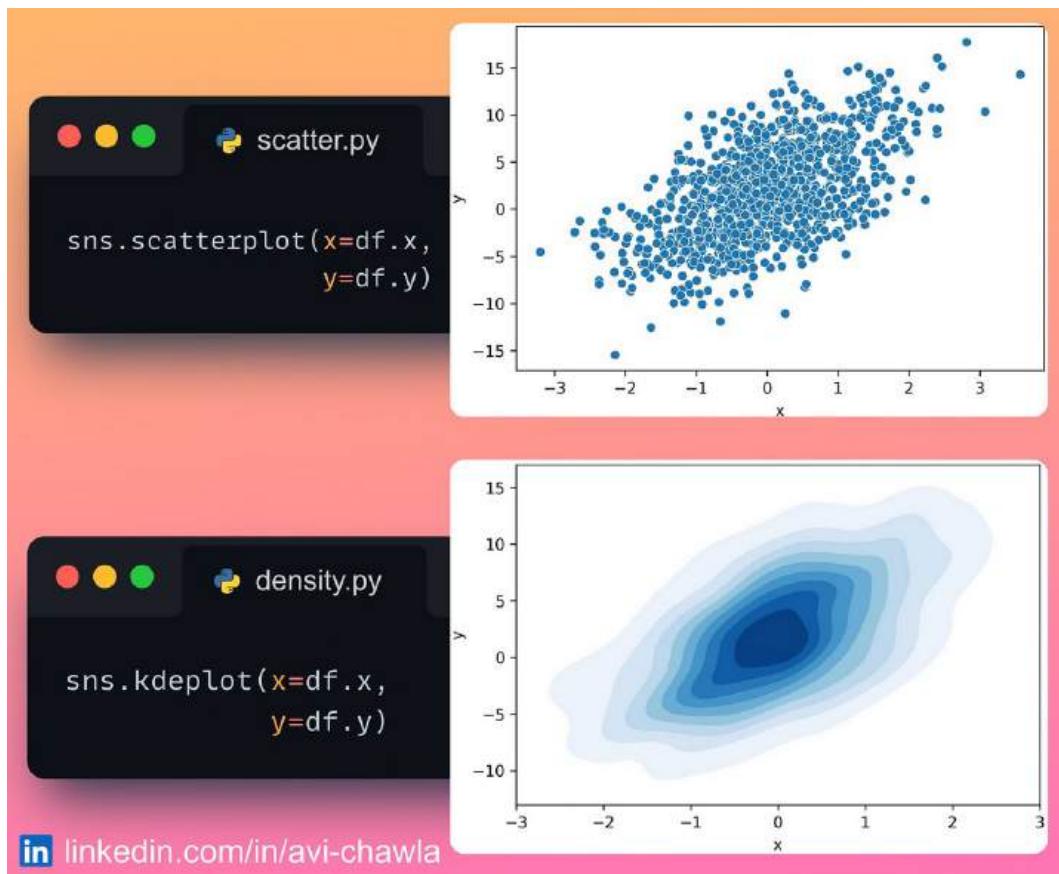
If you want to display python code and its output as LaTeX, try `latexify_py`. With this, you can print python code as a LaTeX expression and make your code more interpretable.

What's more, it can also generate LaTeX code for python code. This saves plenty of time and effort of manually writing the expressions in LaTeX.

Find more info here: [Repository](#).



# Density Plot As A Richer Alternative to Scatter Plot



Scatter plots are extremely useful for visualizing two sets of numerical variables. But when you have, say, thousands of data points, scatter plots can get too dense to interpret.

A density plot can be a good choice in such cases. It depicts the distribution of points using colors (or contours). This makes it easy to identify regions of high and low density.

Moreover, it can easily reveal clusters of data points that might not be obvious in a scatter plot.

Read more: [Docs](#).



# 30 Python Libraries to (Hugely) Boost Your Data Science Productivity

# 30 Python Libraries to (Hugely) Boost Your Data Science Productivity

Here's a collection of 30 essential open-source data science libraries. Each has its own use case and enormous potential to skyrocket your data science skills.

I would love to know the ones you use.

Please find this full document in my LinkedIn post: [Post Link](#).



# Sklearn One-liner to Generate Synthetic Data

```
dummy_data.py
```

```
from sklearn.datasets import make_classification

## create data
X, y = make_classification(n_samples=50,
                           n_features=4,
                           n_classes=2)

>>> print(X)
array([[-0.36,  1.01,  0.19, -1.18],
       [-0.29,  1.21,  0.22, -1.92],
       ...
       [-2.12,  1.82,  0.59,  3.18]])

>>> print(y)
array([0, 1, ..., 0, 1])
```

in [linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

Often for testing/building a data pipeline, we may need some dummy data.

With Sklearn, you can easily create a dummy dataset for regression, classification, and clustering tasks.

More info here: [Sklearn Docs.](#)



# Label Your Data With The Click Of A Button

```
In [3]: from ipyannotate import annotate
         from ipyannotate.buttons import ValueButton as Button

In [5]: annotation = annotate(images_data,
                             buttons=[Button('Dog'), # label buttons list
                                       Button('Cat')])
annotation
```

Dog      Cat



```
In [6]: labels = [task.value for task in annotation.tasks] # get labels
labels
Out[6]: ['Cat', 'Dog', 'Dog', 'Cat']
```

[in linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

Often with unlabeled data, one may have to spend some time annotating/labeling it.

To do this quickly in a jupyter notebook, use **ipyannotate**. With this, you can annotate your data by simply clicking the corresponding button.

Read more: [ipyannotate](#).

Watch a video version of this post on LinkedIn: [Post Link](#).



# Analyze A Pandas DataFrame Without Code

The screenshot shows a Jupyter Notebook environment. In the top code cell (In [1]), the user has run:

```
import pandas as pd  
from pandasql import load
```

In the second cell (In [2]), they have run:

```
df = pd.read_csv("Dummy_Dataset.csv")
```

And in the third cell (In [4]), they have run:

```
show(8C)
```

The main area displays a Pandas DataFrame with 23 rows and 7 columns. The columns are: Index, Name, Company\_Name, Employee\_Job\_Title, Employee\_City, Employee\_Country, and Employee\_Status. The data includes various names like Michael Clark, Etain Smith, Leslie Donovan, Rhylee King, Joshua Patterson, Cheyenne Tornes, Jonathan Chen, Scott Powell, Theresa Doyle, Kristan Harrington, Joseph Hunt, Tracy King, Julie Moss, Tanya Cross, Christopher Berry, Rebecca Jimenez, Adam Simpson, Austin Smith, John Edwards, Theresa Espinosa, Lisa Moore, Scott Escobar, Christopher Calahan, and Tara Sheward. The Employee\_City column lists locations such as Ricardo, Western Sarawak, Singapore, New Russellton, Blue, Ricardo, Tokela, Weston, Kotsabang, New Cindychester, Equatorial Guinea, Ricardo, Palau, Ricardo, Cape Verde, Ricardo, United States of America, Ricardo, Equatorial Guinea, Ricardo, Nepal, North Melosofurt, Ghana, Ricardo, Botswana, Ricardo, Norway, Ricardo, Bangladesh, Ricardo, Honduras, Ricardo, Thailand, Ricardo, Gibraltar, Ricardo, Guyana, Ricardo, East Point and Mayfield, Ricardo, Kirzaz Republic, Ricardo, Mali, and Ricardo, Jamaica.

At the bottom left, there is a LinkedIn link: [linkedin.com/in/avi-chawla](https://linkedin.com/in/avi-chawla).

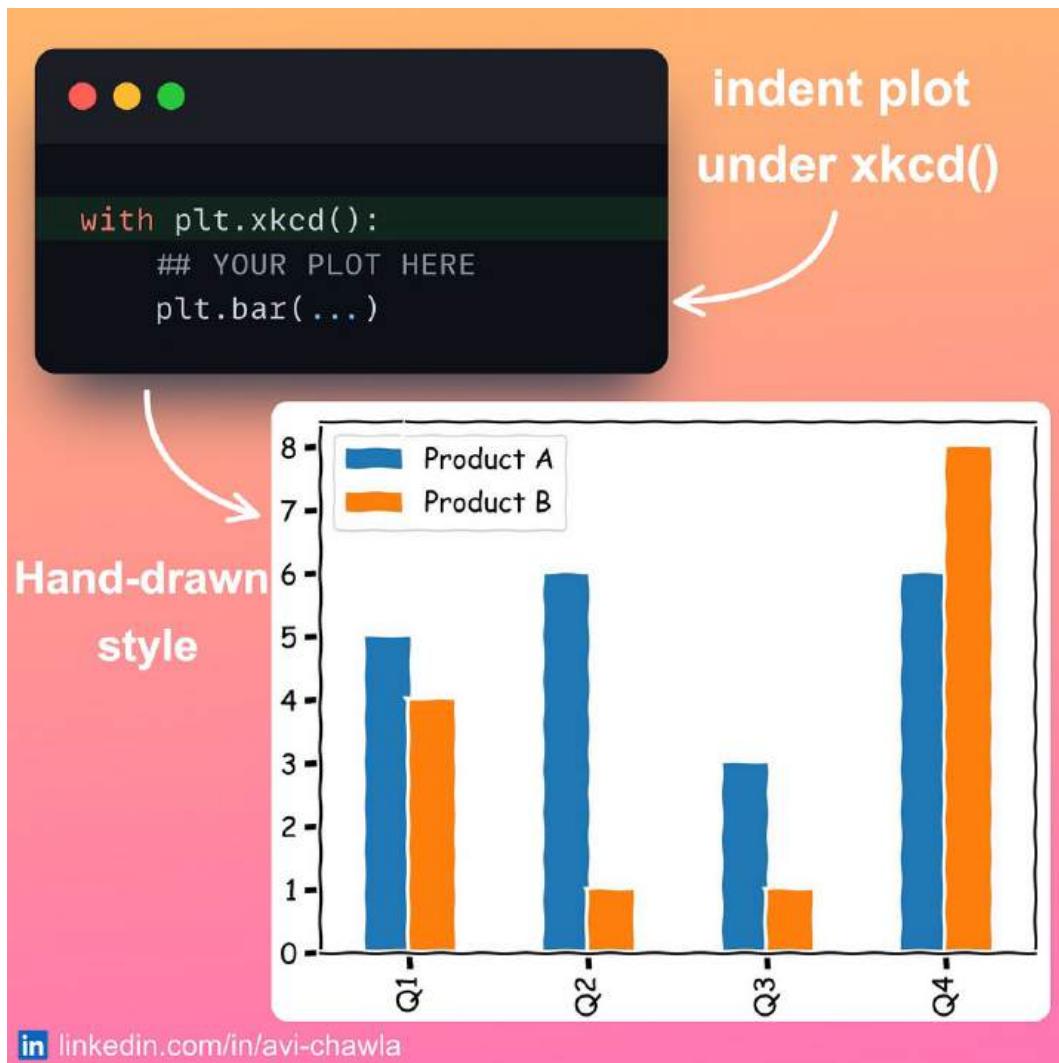
If you want to analyze your dataframe in a GUI-based application, try Pandas GUI. It provides an elegant GUI for viewing, filtering, sorting, describing tabular datasets, etc.

What's more, using its intuitive drag-and-drop functionality, you can easily create a variety of plots and export them as code.

Watch a video version of this post on LinkedIn: [Post Link](#).



# Python One-Liner To Create Sketchy Hand-drawn Plots



[xkcd](#) comic is known for its informal and humorous style, as well as its stick figures and simple drawings.

Creating such visually appealing hand-drawn plots is pretty simple using matplotlib. Just indent the code in a `plt.xkcd()` context to display them in comic style.

Do note that this style is just used to improve the aesthetics of a plot through hand-drawn effects. However, it is not recommended for formal presentations, publications, etc.

Read more: [Docs](#).



# 70x Faster Pandas By Changing Just One Line of Code

The image shows two terminal windows side-by-side. The left window, titled 'Pandas.py', contains Python code for reading a 2M row CSV file and concatenating 20 copies of it. The right window, titled 'Modin.py', contains similar code but imports modin.pandas instead of pandas. Arrows point from the 'Dataset' size and the 'Import Statement' modification to their respective counterparts in each window.

**Pandas.py**

```
import pandas as pd

data = "file.csv" ## 2M Rows

df = pd.read_csv(data)
## 3.6 sec

pd.concat([df for _ in range(20)])
## 7.1 sec
```

**7 GB Dataset**

**Modin.py**

```
import modin.pandas as pd

data = "file.csv" ## 2M Rows

df = pd.read_csv(data)
## 1.3 sec (2.75x Faster)

pd.concat([df for _ in range(20)])
## 0.1 sec (70x Faster)
```

**Modify Import Statement**

in [linkedin.com/in/avi-chawla](https://linkedin.com/in/avi-chawla)

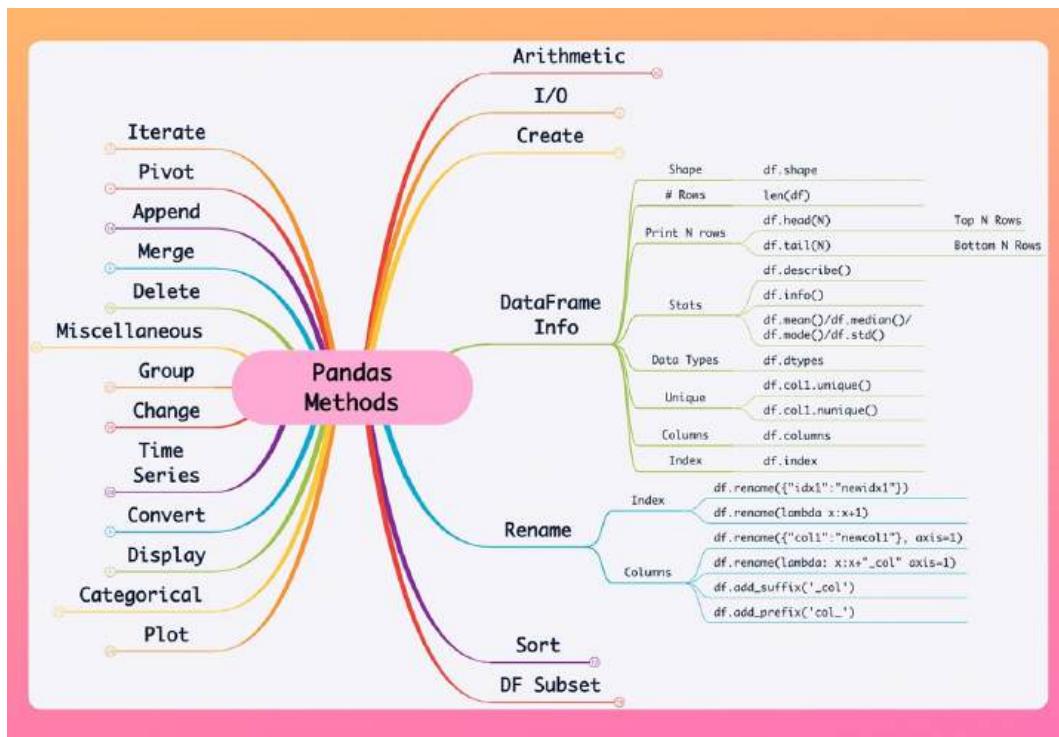
It is challenging to work on large datasets in Pandas. This, at times, requires plenty of optimization and can get tedious as the dataset grows further.

Instead, try Modin. It delivers instant improvements with no extra effort. Change the import statement and use it like the Pandas API, with significant speedups. Find more info in the comments.

Read more: [Modin Guide](#).



# An Interactive Guide To Master Pandas In One Go



Here's a mind map illustrating Pandas Methods on one page. How many do you know :)

- ◆ Load/Save
- ◆ DataFrame info
- ◆ Filter
- ◆ Merge
- ◆ Time-series
- ◆ Plot
- ◆ and many more, in a single map.

Find the full diagram here: [Pandas Mind Map](#).



# Make Dot Notation More Powerful in Python

```
myclass.py
```

```
class Square:
    def __init__(self, length):
        self._side = length

    @property → Getter
    def side(self):
        return self._side

    @side.setter → Setter
    def side(self, length):
        if length<0:
            raise ValueError("Side cannot be negative")
        else:
            self._side = length
```

Raises errors during assignment

```
>>> s = Square(10)
>>> s.side # Getter
10
>>> s.side = -2 # Setter (with dot)
ValueError: Side cannot be negative
```

[linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

Dot notation offers a simple and elegant way to access and modify the attributes of an instance.

Yet, it is a good programming practice to use the getter and setter method for such purposes. This is because it offers more control over how attributes are accessed/changed.

To leverage both in Python, use the **@property** decorator. As a result, you can use the dot notation and still have explicit control over how attributes are accessed/set.



## The Coolest Jupyter Notebook Hack

The diagram illustrates three methods to access the output of a previously run Jupyter cell:

- 1) Use the underscore followed by the output-cell-index: `In [3]: _2` results in `Out[3]: array([1, 2, 3])`.
- 2) Use the `Out` or `_oh` dict and specify the output-cell-index as the key: `In [4]: Out[2]` results in `Out[4]: array([1, 2, 3])`.
- 3) Use the `_oh` dict and specify the output-cell-index as the key: `In [5]: _oh[2]` results in `Out[5]: array([1, 2, 3])`.

in [linkedin.com/in/avi-chawla](https://linkedin.com/in/avi-chawla)

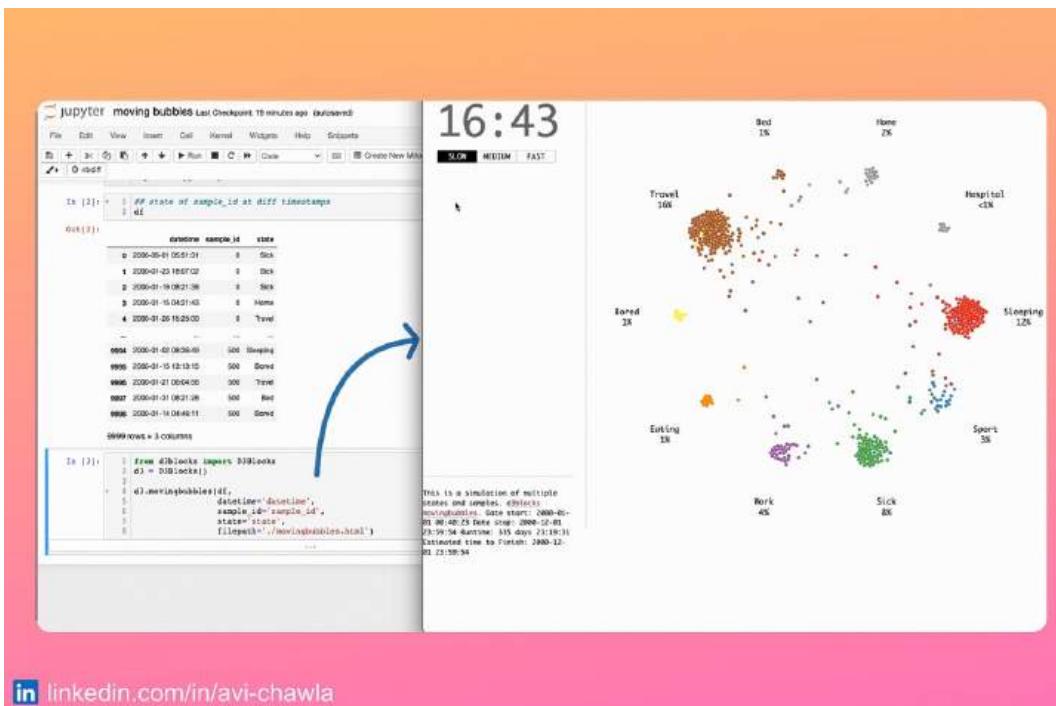
```
In [1]: import numpy as np
In [2]: np.array([1,2,3])
Out[2]: array([1, 2, 3])
In [3]: _2
Out[3]: array([1, 2, 3])
In [4]: Out[2]
Out[4]: array([1, 2, 3])
In [5]: _oh[2]
Out[5]: array([1, 2, 3])
```

Have you ever forgotten to assign the results to a variable in Jupyter? Rather than recomputing the result by rerunning the cell, here are three ways to retrieve the output.

- 1) Use the underscore followed by the output-cell-index.
- 2/3) Use the `Out` or `_oh` dict and specify the output-cell-index as the key.



# Create a Moving Bubbles Chart in Python



Ever seen one of those moving points charts? Here's how you can create one in Python in just three lines of code.

A Moving Bubbles chart is an elegant way to depict the movements of entities across time. Using this, we can easily determine when clusters appear in our data and at what state(s).

To create one, you can use "[d3blocks](#)". Its input should be a DataFrame. A row represents the state of a sample at a particular timestamp.



# Skorch: Use Scikit-learn API on PyTorch Models

```
class MyModel(nn.Module):
    def __init__(self):
        ## Define Network

    def forward(self, x):
        ## Forward Pass
```

```
from skorch import NeuralNetClassifier

model = NeuralNetClassifier(
    MyModel,
    lr=0.1,
    criterion=nn.MSELoss
)

model.fit(X, y)
preds = model.predict(X)
```

Define Pytorch model

Use Scikit-learn API on model

in [linkedin.com/in/avi-chawla](https://linkedin.com/in/avi-chawla)

skorch is a high-level library for PyTorch that provides full Scikit-learn compatibility. In other words, it combines the power of PyTorch with the elegance of sklearn.

Thus, you can train PyTorch models in a way similar to Scikit-learn, using functions such as fit, predict, score, etc.

Using skorch, you can also put a PyTorch model in the sklearn pipeline, and many more.

Overall, it aims at being as flexible as PyTorch while having a clean interface as sklearn.

Read more: [Documentation](#).



# Reduce Memory Usage Of A Pandas DataFrame By 90%

The screenshot shows a Jupyter Notebook interface with two code cells and a data preview.

**Code Cell 1:**

```
## df.shape: (10^7, 2)  
  
=> df.A.dtype  
dtype('int64')  
## Range: [-2^63, 2^63-1]  
  
=> df.A.min(), df.A.max()  
(1, 100)  
  
=> df.A.memory_usage()  
76.3 MB
```

**Code Cell 2:**

```
df["A"] = df.A.astype(np.int8)  
## Range: [-128, 127]  
  
=> df.A.memory_usage()  
9.5 MB # (~90% Lower)
```

**Data Preview:**

	A	B
0	38	46
1	28	58
2	47	82
3	88	87
4	13	78

**Annotations:**

- Three arrows point from the text "Supported range larger than required" to the line "# Range: [-2^63, 2^63-1]" in the first code cell.
- An arrow points from the text "Convert to smaller datatype" to the line "df["A"] = df.A.astype(np.int8)" in the second code cell.

**LinkedIn Profile:** [linkedin.com/in/avi-chawla](https://linkedin.com/in/avi-chawla)

By default, Pandas always assigns the highest memory datatype to its columns. For instance, an integer-valued column always gets the int64 datatype, irrespective of its range.

To reduce memory usage, represent it using an optimized datatype, which is enough to span the range of values in your columns.

Read [this blog](#) for more info. It details many techniques to optimize the memory usage of a Pandas DataFrame.



# An Elegant Way To Perform Shutdown Tasks in Python

The image shows a Mac OS X desktop with two windows. The top window is a code editor titled 'my\_file.py' containing Python code. The bottom window is a terminal window titled 'Terminal' showing the output of running the script.

**Code Editor (my\_file.py):**

```
import atexit  
  
@atexit.register  
def final_function():  
    print("COMPLETED EXECUTION!")  
  
for i in range(5):  
    print(f"num = {i}")
```

**Terminal Output:**

```
$ python my_file.py  
num = 0  
num = 1  
num = 2  
num = 3  
num = 4  
COMPLETED EXECUTION!
```

**Annotations:**

- A red arrow points from the text "The decorator invokes the function at the end" to the '@atexit.register' line in the code editor.
- A red arrow points from the text "Add decorator to method" to the 'final\_function()' definition in the code editor.

**LinkedIn Profile:** [linkedin.com/in/avi-chawla](https://linkedin.com/in/avi-chawla)

Often towards the end of a program's execution, we run a few basic tasks such as saving objects, printing logs, etc.

To invoke a method right before the interpreter is shutting down, decorate it with the **@atexit.register** decorator.

The good thing is that it works even if the program gets terminated unexpectedly. Thus, you can use this method to save the state of the program or print any necessary details before it stops.

Read more: [Documentation](#).



# Visualizing Google Search Trends of 2022 using Python



If your data has many groups, visualizing their distribution together can create cluttered plots. This makes it difficult to visualize the underlying patterns.

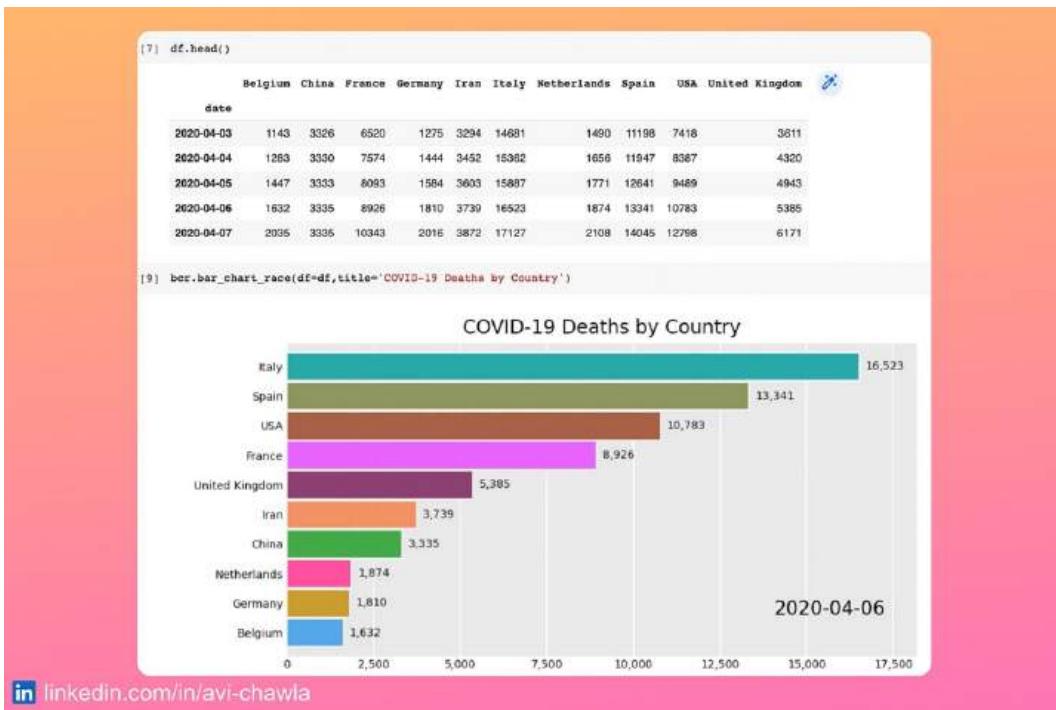
Instead, consider plotting the distribution across individual groups using FacetGrid. This allows you to compare the distributions of multiple groups side by side and see how they vary.

As shown above, a FacetGrid allows us to clearly see how different search terms trended across 2022.

P.S. I used the [year-in-search-trends](#) repository to fetch the trend data.



# Create A Racing Bar Chart In Python



Ever seen one of those racing bar charts? Here's how you can create one in Python in just two lines of code.

A racing bar chart is typically used to depict the progress of multiple values over time.

To create one, you can use the "**bar-chart-race**" library.

Its input should be a Pandas DataFrame where every row represents a single timestamp. The column holds the corresponding values for a particular category.

Read more: [Documentation](#).



# Speed-up Pandas Apply 5x with NumPy

The screenshot shows two code cells side-by-side. The top cell, titled "Pandas Apply", contains Python code for defining a function `assign\_class` and applying it to a DataFrame column. The bottom cell, titled "NumPy Select", contains Python code for using `np.select` to achieve the same result. A pink arrow points from the "Default" section of the NumPy code to the "Pandas Apply" cell, indicating that the Pandas approach is slower.

**Pandas Apply**

```
def assign_class(num):
    if num<10:
        return "Class A"
    if num<50:
        return "Class B"
    return "Class C"

df.A.apply(assign_class)
## 1.02 s ± 20.5 ms per loop
```

**NumPy Select**

```
condlist = [ df["A"]<10 , df["A"]<50 ]
resultlist = [ "Class A" , "Class B" ]

np.select(condlist, resultlist, "Class C")
## 0.20 s ± 7.14 ms per loop
```

**~5x Faster**

**Default**

in [linkedin.com/in/avi-chawla](https://linkedin.com/in/avi-chawla)

While creating conditional columns in Pandas, we tend to use the **apply()** method almost all the time.

However, **apply()** in Pandas is nothing but a glorified for-loop. As a result, it misses the whole point of vectorization.

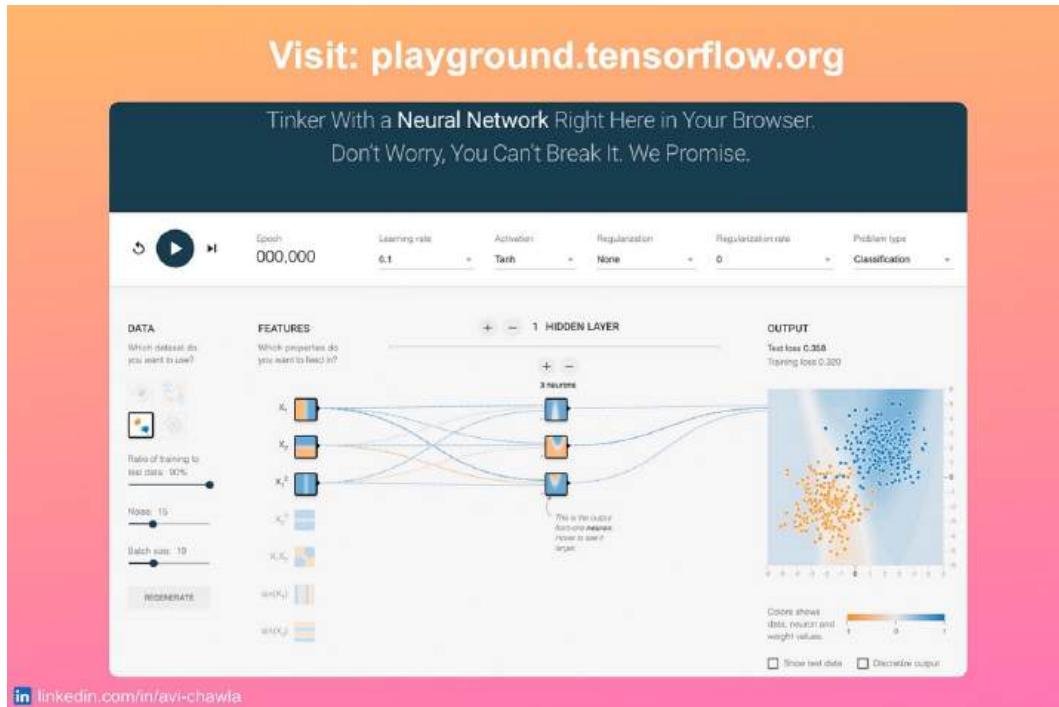
Instead, you should use the **np.select()** method to create conditional columns. It does the same job but is extremely fast.

The conditions and the corresponding results are passed as the first two arguments. The last argument is the default result.

Read more here: [NumPy docs](#).



# A No-Code Online Tool To Explore and Understand Neural Networks



Neural networks can be intimidating for beginners. Also, experimenting programmatically does not provide enough intuitive understanding about them.

Instead, try TensorFlow Playground. Its elegant UI allows you to build, train and visualize neural networks without any code.

With a few clicks, one can see how neural networks work and how different hyperparameters affect their performance. This makes it especially useful for beginners.

Try here: [Tensorflow Playground](http://playground.tensorflow.org).



# What Are Class Methods and When To Use Them?

Define classmethod

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    @classmethod
    def from_square(cls, size):
        return Rectangle(size, size)
```

create object using classmethod →

```
rect = Rectangle.from_square(5)

print(rect.width) # Output: 5
print(rect.height) # Output: 5
```

[linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

Class methods, as the name suggests, are bound to the class and not the instances of a class. They are especially useful for providing an alternative interface for creating instances.

Moreover, they can be also used to define utility functions that are related to the class rather than its instances.

For instance, one can define a class method that returns a list of all instances of the class. Another use could be to calculate a class-level statistic based on the instances.

To define a class method in Python, use the **@classmethod** decorator. As a result, this method can be called directly using the name of the class.



# Make Sklearn KMeans 20x times faster

The image shows two terminal windows side-by-side. The top window is titled 'sklearn.py' and contains Python code for training a KMeans model on a dataset 'x\_train'. The bottom window is titled 'faiss.py' and contains Python code for training a KMeans model using the Faiss library. A large green arrow points from the 'faiss.py' window to the 'sklearn.py' window, with the text '≈20x Faster' written above the arrow. The 'faiss.py' window also includes a LinkedIn link: [linkedin.com/in/avi-chawla](https://linkedin.com/in/avi-chawla).

```
from sklearn.cluster import KMeans
kmeans = KMeans(8).fit(x_train)
# Training Time: 162s
```

x\_train shape: (500000, 1024)

```
import faiss
kmeans = faiss.Kmeans(d=1024, k=8)
kmeans.train(x_train)
# Training Time: 7.8s
```

≈20x Faster

in [linkedin.com/in/avi-chawla](https://linkedin.com/in/avi-chawla)

The KMeans algorithm is commonly used to cluster unlabeled data. But with large datasets, scikit-learn takes plenty of time to train and predict.

To speed-up KMeans, use Faiss by Facebook AI Research. It provides faster nearest-neighbor search and clustering.

Faiss uses "Inverted Index", an optimized data structure to store and index the data points. This makes performing clustering extremely efficient.

Additionally, Faiss provides parallelization and GPU support, which further improves the performance of its clustering algorithms.

Read more: [GitHub](#).



## Speed-up NumPy 20x with Numexpr

```
import numpy as np
import numexpr as ne

a = np.random.random(10**7)
b = np.random.random(10**7)

%timeit np.cos(a) + np.sin(b)
142 ms ± 257 µs per loop

%timeit ne.evaluate("cos(a) + sin(b)")
32.5 ms ± 229 µs per loop ~5x Faster
```

[in linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

Numpy already offers fast and optimized vectorized operations. Yet, it does not support parallelism. This provides further scope for improving the run-time of NumPy.

To do so, use Numexpr. It allows you to speed up numerical computations with multi-threading and just-in-time compilation.

Depending upon the complexity of the expression, the speed-ups can range from 0.95x and 20x. Typically, it is expected to be 2x-5x.

Read more: [Documentation](#).



# A Lesser-Known Feature of Apply Method In Pandas

The screenshot shows a Jupyter Notebook interface with four code cells and a data frame preview.

- Code Cell 1:** A Python function definition:

```
def min_max(row):
    return max(row), min(row)
```
- Data Frame Preview:** A preview of a DataFrame with columns A, B, and C, and two rows (0, 1).

	A	B	C
0	1	3	2
1	4	6	3
- Code Cell 2:** A command to apply the function to the DataFrame along axis 1:

```
>>> df.apply(min_max, axis = 1)
0 (3, 1)
1 (6, 3)
```

A callout arrow points from this cell to the text "Pandas Series of Tuple".
- Code Cell 3:** A command to apply the function with the result\_type argument set to "expand":

```
>>> df.apply(min_max, axis = 1,
             result_type="expand")
```

	0	1
0	3	1
1	6	3

A callout arrow points from this cell to the text "Pandas DataFrame".
- Bottom Left:** A LinkedIn profile link: [linkedin.com/in/avi-chawla](https://linkedin.com/in/avi-chawla)

After applying a method on a DataFrame, we often return multiple values as a tuple. This requires additional steps to project it back as separate columns.

Instead, with the `result_type` argument, you can control the shape and output type. As desired, the output can be either a DataFrame or a Series.



# An Elegant Way To Perform Matrix Multiplication

The image shows a Mac OS X desktop with a terminal window open. A white arrow points from the top snippet to the bottom one.

```
import numpy as np
```

```
x = np.matmul(a, np.matmul(b, c))
```

```
x = a @ b @ c
```

[in](https://www.linkedin.com/in/avi-chawla) [linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

Matrix multiplication is a common operation in machine learning. Yet, chaining repeated multiplications using **matmul** function makes the code cluttered and unreadable.

If you are using NumPy, you can instead use the **@** operator to do the same.



# Create Pandas DataFrame from Dataclass

The diagram illustrates the creation of a Pandas DataFrame from a list of Dataclass objects. It consists of two code snippets in dark-themed windows, each with three circular window controls (red, yellow, green) at the top.

**Top Window (define dataclass):**

```
from dataclasses import dataclass

@dataclass
class Point:
    x_loc:int
    y_loc:int
```

**Bottom Window (list of dataclass objects):**

```
points = [Point(5, 5),
          Point(1, 4),
          Point(2, 3)]

pd.DataFrame(points)
"""
      x_loc  y_loc
0      5      5
1      1      4
2      2      3
"""
```

**Annotations:**

- A white arrow points from the text "list of dataclass objects" to the first line of the bottom window's code: "points = [Point(5, 5),".
- A white arrow points from the text "define dataclass" to the first line of the top window's code: "from dataclasses import dataclass".

**LinkedIn Profile:**

in [linkedin.com/in/avi-chawla](https://linkedin.com/in/avi-chawla)

A Pandas DataFrame is often created from a Python list, dictionary, by reading files, etc. However, did you know you can also create a DataFrame from a Dataclass?

The image demonstrates how you can create a DataFrame from a list of dataclass objects.



# Hide Attributes While Printing A Dataclass Object

The image shows two code snippets in a Python code editor. The top snippet illustrates the default behavior where all attributes are printed. The bottom snippet shows how to use the `repr=False` field parameter to hide specific attributes from the print output.

**Top Snippet (Default Behavior):**

```
from dataclasses import dataclass

@dataclass
class Student:
    name:str
    key:str

Jane = Student("Jane", "27HD")

print(Jane)
Student(name='Jane', key='27HD')
```

A callout arrow points from the text "Prints all attributes" to the printed output "Student(name='Jane', key='27HD')".

**Bottom Snippet (Hiding Attributes):**

```
from dataclasses import dataclass, field

@dataclass
class Student:
    name:str
    key:str = field(repr = False)

Jane = Student("Jane", "27HD")

print(Jane)
Student(name='Jane')
```

A callout arrow points from the text "Attribute hidden in print" to the line "key:str = field(repr = False)".

[linkedin.com/in/avi-chawla](https://linkedin.com/in/avi-chawla)

By default, a dataclass prints all the attributes of an object declared during its initialization.

But if you want to hide some specific attributes, declare `repr=False` in its field, as shown above.



## List : Tuple :: Set : ?

The terminal window shows two snippets of Python code. The top snippet, titled 'set.py', contains:

```
my_set = {1, 2, 3}

my_dict = {my_set: "A set"}
## TypeError: unhashable type: 'set'
```

A red arrow points from the error message to the text 'A set cannot be added as a key' located to the right of the terminal window.

The bottom snippet, titled 'frozenset.py', contains:

```
## frozenset
my_set = frozenset({1, 2, 3})

my_dict = {my_set: "A frozen set"}

my_dict[my_set]
"A frozen set"
```

To the left of the bottom terminal window, the text 'Use frozenset' is displayed in white on a pink background.

[linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

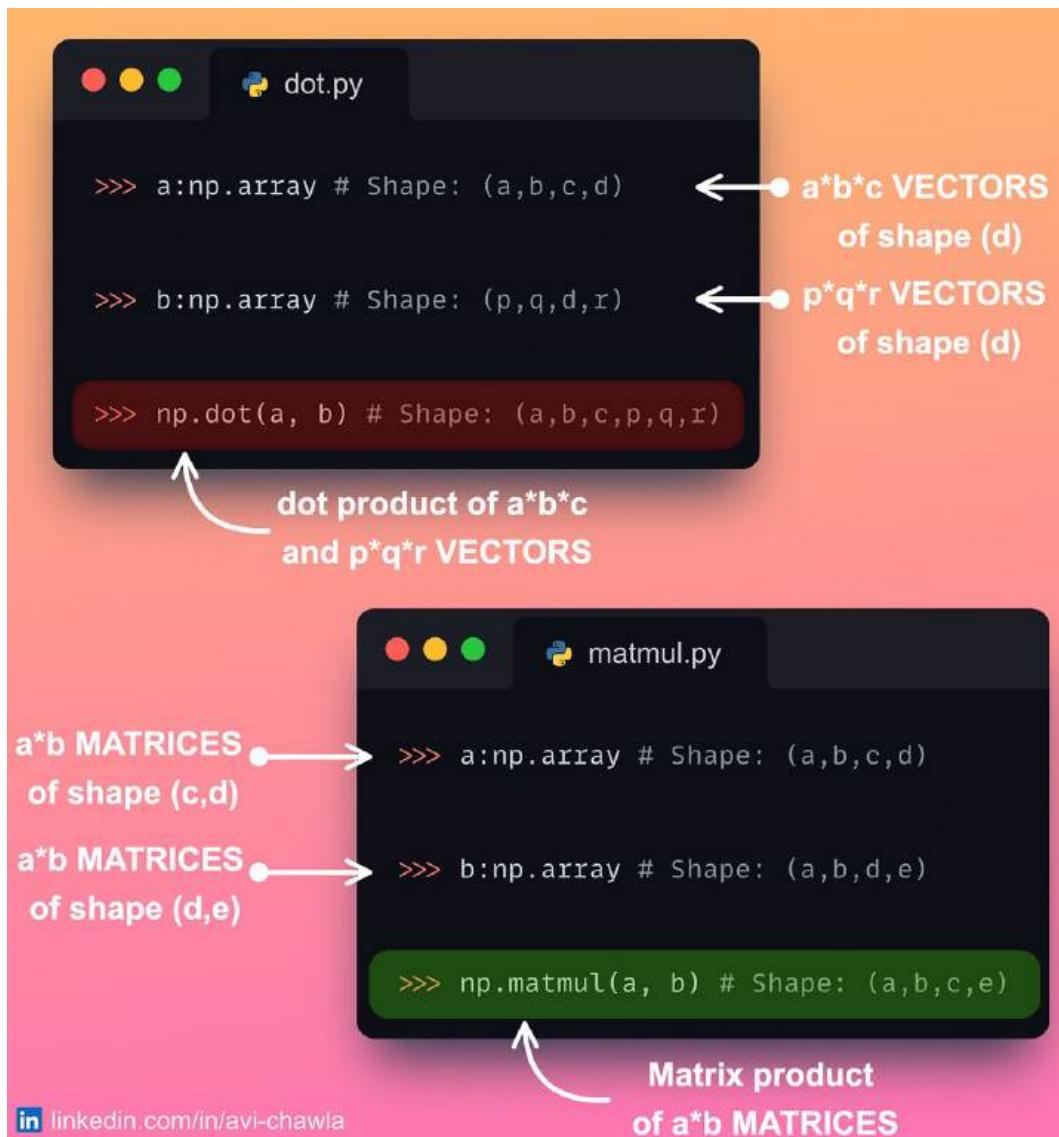
Dictionaries in Python require their keys to be immutable. As a result, a set cannot be used as keys as it is mutable.

Yet, if you want to use a set, consider declaring it as a frozenset.

It is an immutable set, meaning its elements cannot be changed after it is created. Therefore, they can be safely used as a dictionary's key.



# Difference Between Dot and Matmul in NumPy



The **np.matmul()** and **np.dot()** methods produce the same output for 2D (and 1D) arrays. This makes many believe that they are the same and can be used interchangeably, but that is not true.

The **np.dot()** method revolves around individual vectors (or 1D arrays). Thus, it computes the dot product of ALL vector pairs in the two inputs.

The **np.matmul()** method, as the name suggests, is meant for matrices. Thus, it computes the matrix multiplication of corresponding matrices in the two inputs.



# Run SQL in Jupyter To Analyze A Pandas DataFrame

The image shows two Jupyter notebook cells side-by-side against a background gradient from orange to pink.

**Pandas:** The top cell is titled "Filter-Pandas.ipynb". It contains the Python code: `df[df.city == "New Delhi"]`.

**DuckDB:** The bottom cell is titled "Filter-SQL.ipynb". It contains the DuckDB SQL code: `%%sql select * from df where city = 'New Delhi';`.

A LinkedIn link icon followed by the URL [linkedin.com/in/avi-chawla](https://linkedin.com/in/avi-chawla) is located at the bottom left of the screenshot.

Pandas already provides a wide range of functionalities to analyze tabular data. Yet, there might be situations when one feels comfortable using SQL over Python.

Using DuckDB, you can analyze a Pandas DataFrame with SQL syntax in Jupyter, without any significant run-time difference.

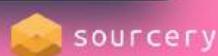
Read the guide here to get started: [Docs](#).



# Automated Code Refactoring With Sourcery



[in](https://linkedin.com/in/avi-chawla) linkedin.com/in/avi-chawla



Refactoring codebase is an important yet time-consuming task. Moreover, at times, one might unknowingly introduce errors during refactoring.

This takes additional time for testing and gets tedious with more refactoring, especially when the codebase is big.

Rather than following this approach, use [Sourcery](#). It's an automated refactoring tool that makes your code elegant, concise, and Pythonic in no time.

With Sourcery, you can refactor code in many ways. For instance, you can refactor scripts through the command line, as an IDE plugin in VS Code and PyCharm, etc.

Read my full blog on Sourcery here: [Medium](#).



# \_\_Post\_\_init\_\_: Add Attributes To A Dataclass Object Post Initialization

```
from dataclasses import dataclass

@dataclass
class StudentMarks:
    student_id:str
    marks:float

    def __post_init__(self):
        if self.marks>30:
            self.grade = "Pass"
        else:
            self.grade = "Fail"

Peter = StudentMarks("B20", 43)

print(Peter.grade) # Pass
```

[linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

After initializing a class object, we often create derived attributes from existing variables.

To do this in dataclasses, you can use the `__post_init__` method. As the name suggests, this method is invoked right after the `__init__` method.

This is useful if you need to perform additional setups on your dataclass instance.



# Simplify Your Functions With Partial Functions

Fix some parameters →

```
def quadratic(x, a, b, c):
    return a*(x**2) + b*x + c
```

```
py partial_function.py

from functools import partial
quadratic_c1 = partial(quadratic, c=1)

quadratic_c1(x=1, a=4, b=5)
# Output: 10
```

[in](https://www.linkedin.com/in/avi-chawla) linkedin.com/in/avi-chawla

When your function takes many arguments, it can be a good idea to simplify it by using partial functions.

They let you create a new version of the function with some of the arguments fixed to specific values.

This can be useful for simplifying your code and making it more readable and concise. Moreover, it also helps you avoid repeating yourself while invoking functions.



# When You Should Not Use the `head()` Method In Pandas

`df.sort_values(by="Marks",  
ascending=False).head(3)`

	Name	Marks
1	Jane	100
2	Mark	97
0	Peter	95

**Ignores  
repeated  
values**

`df`

	Name	Marks
0	Peter	95
1	Jane	100
2	Mark	97
3	David	95

`df.nlargest(n=3,  
columns="Marks",  
keep="all")`

	Name	Marks
1	Jane	100
2	Mark	97
0	Peter	95
3	David	95

[in](https://linkedin.com/in/avi-chawla) linkedin.com/in/avi-chawla

One often retrieves the top **k** rows of a sorted Pandas DataFrame by using `head()` method. However, there's a flaw in this approach.

If your data has repeated values, `head()` will not consider that and just return the first **k** rows.

If you want to consider repeated values, use `nlargest` (or `nsmallest`) instead. Here, you can specify the desired behavior for duplicate values using the `keep` parameter.



# DotMap: A Better Alternative to Python Dictionary

**Add using dot notation**

```
from dotmap import DotMap
students = DotMap()
students.john.id = "12A"
students.john.english = 45
students.mary.id = "12B"
students.mary.english = 49
students.dave.id = "12C"
students.dave.english = 34
```

**Pretty Print**

```
people pprint()
{'dave': {'english': 34, 'id': '12C'},
 'john': {'english': 45, 'id': '12A'},
 'mary': {'english': 49, 'id': '12B'}}
```

[in linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

Python dictionaries are great, but they have many limitations.

It is difficult to create dynamic hierarchical data. Also, they don't offer the widely adopted dot notation to access values.

Instead, use DotMap. It behaves like a Python dictionary but also addresses the above limitations.

What's more, it also has a built-in pretty print method to display it as a dict/JSON for debugging large objects.

Read more: [GitHub](#).



# Prevent Wild Imports With `__all__` in Python

The image shows two Python code editors side-by-side. The left editor, titled 'my\_functions.py', contains the following code:

```
__all__ = ["func1", "func2"]

def func1():
    return "Function 1"

def func2():
    return "Function 2"

def func3():
    return "Function 3"
```

A red box highlights the line `__all__ = ["func1", "func2"]`. A white arrow points from this box to the text 'Specify \_\_all\_\_' on the right. Another red box highlights the imports in the second editor. A white arrow points from this box to the text 'Only func1 and func2 imported' on the left.

The right editor, titled 'module.py', contains the following code:

```
from my_functions import *

>>> func1()
"Function 1"

>>> func2()
"Function 2"

>>> func3()
NameError: name 'func3' is not defined
```

A red box highlights the line `from my_functions import *`. A white arrow points from this box to the text 'Only func1 and func2 imported' on the left.

At the bottom left of the image is a LinkedIn profile link: [linkedin.com/in/avi-chawla](https://linkedin.com/in/avi-chawla).

Wild imports (`from module import *`) are considered a bad programming practice. Yet, here's how you can prevent it if someone irresponsibly does that while using your code.

In your module, you can define the importable functions/classes/variables in `__all__`. As a result, whenever someone will do a wild import, Python will only import the symbols specified here.

This can be also useful to convey what symbols in your module are intended to be private.



## Three Lesser-known Tips For Reading a CSV File Using Pandas

The collage consists of three rectangular windows, each showing a snippet of Python code for reading a CSV file:

- Top Window:** Shows the code `pd.read_csv("data.csv", nrows = 10)`. To its right, the text "Read only first 10 rows" is displayed.
- Middle Window:** Shows the code `pd.read_csv("data.csv", usecols = ["A", "C"])`. To its left, the text "Read specific columns" is displayed.
- Bottom Window:** Shows two snippets of code: `pd.read_csv("data.csv", skiprows = 10)` and `pd.read_csv("data.csv", skiprows = [1, 5])`. To the right of the first snippet, the text "Skip first 10 rows" is shown with an arrow pointing from the code. To the right of the second snippet, the text "Skip 1st row and 5th row" is shown with an arrow pointing from the code.

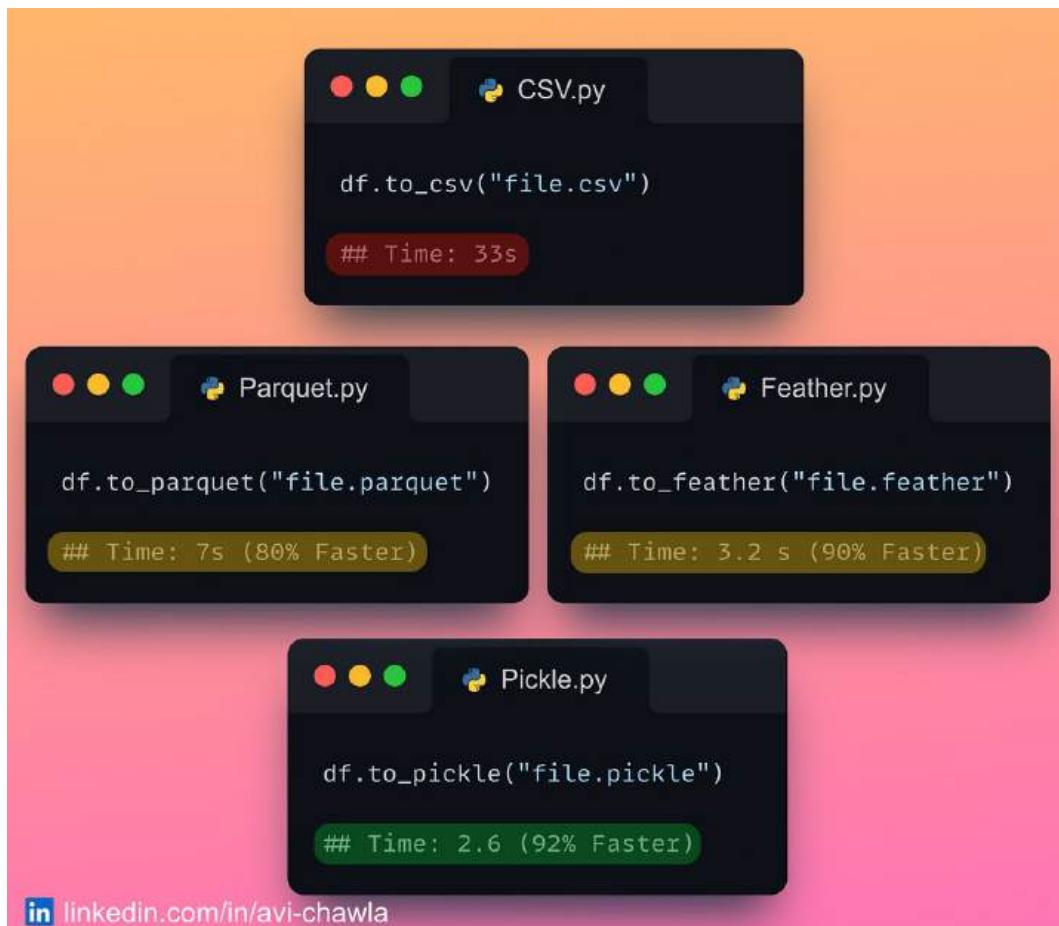
At the bottom left of the collage, there is a small LinkedIn icon followed by the URL [linkedin.com/in/avi-chawla](https://linkedin.com/in/avi-chawla).

Here are three extremely useful yet lesser-known tips for reading a CSV file with Pandas:

1. If you want to read only the first few rows of the file, specify the **nrows** parameter.
2. To load a few specific columns, specify the **usecols** parameter.
3. If you want to skip some rows while reading, pass the **skiprows** parameter.



# The Best File Format To Store A Pandas DataFrame



In the image above, you can find the run-time comparison of storing a Pandas DataFrame in various file formats.

Although CSVs are a widely adopted format, it is the slowest format in this list.

Thus, CSVs should be avoided unless you want to open the data outside Python (in Excel, for instance).

Read more in my blog: [Medium](#).



# Debugging Made Easy With PySnooper

The screenshot shows two terminal windows. The top window displays Python code using the `pysnooper` library to debug a function. The bottom window shows the resulting debugging output.

**Code:**

```
py-snooper.py
1 import pysnooper
2
3 @pysnooper.snoop()
4 def add_sub(a, b):
5
6     add = a+b
7     sub = a-b
8
9     return (add, sub)
10
11 add_sub(9, 5)
```

**Add Decorator:** A callout points to the line `@pysnooper.snoop()`.

**Debugging Output:** A callout points to the terminal output below.

**Terminal Output:**

```
$ python py-snooper.py
Starting var:.. a = 9
Starting var:.. b = 5
call          4 def add_sub(a, b):
line          6     add = a+b
New var:..... add = 14
line          7     sub = a-b
New var:..... sub = 4
line          9     return (add, sub)
Return value:.. (14, 4)
```

[linkedin.com/in/avi-chawla](https://linkedin.com/in/avi-chawla)

Rather than using many print statements to debug your python code, try PySnooper.

With just a single line of code, you can easily track the variables at each step of your code's execution.

Read more: [Repository](#).



# Lesser-Known Feature of the Merge Method in Pandas

```
pd.merge(name_df, rewards_df,
         on = "Cust_ID",
         how = "outer",
         indicator = True)
```

Cust_ID	Name	Rewards	_merge
1	Joe	NaN	left_only
2	Mark	50	both
3	Peter	20	both
4	NaN	70	right_only

[in.linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

While merging DataFrames in Pandas, keeping track of the source of each row in the output can be extremely useful.

You can do this using the **indicator** argument of the **merge()** method. As a result, it augments an additional column in the merged output, which tells the source of each row.



# The Best Way to Use Apply() in Pandas



The image above shows a run-time comparison of popular open-source libraries that provide parallelization support for Pandas.

You can find the links to these libraries [here](#). Also, if you know any other similar libraries built on top of Pandas, do post them in the comments or reply to this email.



# Deep Learning Network Debugging Made Easy

```
import tsensor

W = # Shape: (n_neurons, hidden_size)
b = # Shape: (n_neurons, 1)
X = # Shape: (n_batches, batch_size, hidden_size)

with tsensor.explain():
    for i in range(n_batches):
        batch = X[i,:,:]
        Y = torch.matmul(W, batch.T) + b
```

**Output**

batch =  $X_{[i,:,:]}$

$10 \times 764$   $10 \times 10$

$10 \times 764$   $10 \times 10$

$Y = \text{torch.matmul}(W, \text{batch.T}) + b$

$100 \times 10$   $100 \times 764$   $100 \times 10$   $100 \times 1$

$100 \times 764$   $100 \times 10$   $100 \times 1$

[in](https://linkedin.com/in/avi-chawla) [linkedin.com/in/avi-chawla](https://linkedin.com/in/avi-chawla)

Aligning the shape of tensors (or vectors/matrices) in a network can be challenging at times.

As the network grows, it is common to lose track of dimensionalities in a complex expression.

Instead of explicitly printing tensor shapes to debug, use **TensorSensor**. It generates an elegant visualization for each statement executed within its block. This makes dimensionality tracking effortless and quick.

In case of errors, it augments default error messages with more helpful details. This further speeds up the debugging process.

Read more: [Documentation](#)



# Don't Print NumPy Arrays! Use Lovely-NumPy Instead.

The image shows two terminal windows side-by-side. The left window has a dark background and displays raw NumPy array output. The right window has a light background and displays the output of the `lo` function from the `lovely_numpy` module. A red arrow points from the text "Summary of Array" on the left to the right window. A yellow arrow points from the text "Only numbers" on the right back to the left window. A blue button at the bottom left says "linkedin.com/in/avi-chawla".

**Left Terminal (Raw NumPy Output):**

```
>>> array = np.random.rand(...)  
>>> array  
tensor([[0.59, 0.03, ..., 0.44, 0.41],  
       [0.60, 0.72, ..., 0.92, 0.61],  
       ...,  
       [0.57, 0.98, ..., 0.01, 0.91],  
       [0.00, 0.53, ..., 0.54, 0.54]])
```

**Right Terminal (Lovely-NumPy Output):**

```
from lovely_numpy import lo  
  
>>> array = np.random.rand(...)  
>>> lo(array)  
array[10, 20] n=200  
x∈[0.0, 1.0] μ=0.51 σ=0.3  
  
>>> array = np.zeros(...)  
>>> lo(array)  
array[10] all_zeros  
  
>>> array = # With NaN and Inf  
>>> lo(array)  
array[10, 20] n=200  
x∈[0.0, 1.0] μ=0.51 σ=0.3 +Inf! NaN!
```

We often print raw numpy arrays during debugging. But this approach is not very useful. This is because printing does not convey much information about the data it holds, especially when the array is large.

Instead, use **lovely-numpy**. Rather than viewing raw arrays, it prints a summary of the array. This includes its shape, distribution, mean, standard deviation, etc.

It also shows if the numpy array has NaNs and Inf values, whether it is filled with zeros, and many more.

P.S. If you work with tensors, then you can use **lovely-tensors**.

Read more: [Documentation](#).



# Performance Comparison of Python 3.11 and Python 3.10

The image shows four terminal windows side-by-side, each containing Python code and its execution results.

- Top Left:** A file named `fib.py` containing a recursive function to calculate the Nth Fibonacci number.
- Top Right:** A file named `calc_pi.py` containing a function to calculate the approximate value of pi using a series.
- Bottom Left:** A file named `python3.10.py` showing benchmarks for `fib(30)`, `fib(40)`, and `pi_approx(10**6)`. The times are 260ms, 32s, and 144ms respectively.
- Bottom Right:** A file named `python3.11.py` showing the same benchmarks. The times are significantly faster: 97ms (62% faster than 3.10), 12s (62% faster than 3.10), and 65ms (55% faster than 3.10).

[linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

Python 3.11 was released recently, and as per the official release, it is expected to be 10-60% faster than Python 3.10.

I ran a few basic benchmarking experiments to verify the performance boost. Indeed, Python 3.11 is much faster.

Although one might be tempted to upgrade asap, there are a few things you should know. Read more [here](#).



# View Documentation in Jupyter Notebook

The screenshot shows a Jupyter Notebook interface. In the top cell (In [1]), the code `import pandas as pd` is run, followed by its execution time and date. Below it, another cell (In [ ]), which is currently active, has the code `pd.DataFrame() # Shift-Tab` entered. A tooltip or documentation panel is open over this cell, displaying the function's signature, parameters (data=None, index='Axes | None' = None, columns='Axes | None' = None, dtype='Dtype | None' = None, copy='bool | None' = None), and docstring: "Two-dimensional size-mutable, potentially heterogeneous tabular data". At the bottom left of the notebook area, there is a LinkedIn link: [linkedin.com/in/avi-chawla](https://linkedin.com/in/avi-chawla).

While working in Jupyter, it is common to forget the parameters of a function and visit the official docs (or Stackoverflow). However, you can view the documentation in the notebook itself.

Pressing **Shift-Tab** opens the documentation panel. This is extremely useful and saves time as one does not have to open the official docs every single time.

This feature also works for your custom functions.

View a video version of this post on LinkedIn: [Post Link](#).



# A No-code Tool To Understand Your Data Quickly

The screenshot shows a dark-themed terminal window containing Python code for generating a profile report:

```
from pandas_profiling import ProfileReport
profile = ProfileReport(iris_data,
                        title="Pandas Profiling Report")
profile.to_widgets()
```

Below the terminal is a screenshot of the Pandas Profiling Report UI. The UI has a tab-based navigation bar at the top: Overview, Variables, Interactions, Correlations, Missing values, Sample, and Duplicate rows. The 'Alerts (7)' tab is selected, showing the following alerts:

Alert Type	Details
Duplicates	Dataset has 1 (0.7%) duplicate rows.
High correlation	sepal length (cm) is highly correlated with sepal width (cm) and 3 other fields.
High correlation	petal length (cm) is highly correlated with sepal length (cm) and 3 other fields.
High correlation	petal width (cm) is highly correlated with sepal length (cm) and 3 other fields.
High correlation	target is highly correlated with sepal length (cm) and 3 other fields.
High correlation	sepal width (cm) is highly correlated with sepal length (cm) and 3 other fields.
Uniform	target is uniformly distributed.

At the bottom left of the UI, it says "Report generated by YData". At the bottom right, there is a LinkedIn link: [linkedin.com/in/avi-chawla](https://linkedin.com/in/avi-chawla).

The preliminary steps of any typical EDA task are often the same. Yet, across projects, we tend to write the same code to carry out these tasks. This gets repetitive and time-consuming.

Instead, use **pandas-profiling**. It automatically generates a standardized report for data understanding in no time. Its intuitive UI makes this effortless and quick.

The report includes the dimension of the data, missing value stats, and column data types. What's more, it also shows the data distribution, the interaction and correlation between variables, etc.

Lastly, the report also includes alerts, which can be extremely useful during analysis/modeling.

Read more: [Documentation](#).



## Why 256 is 256 But 257 is not 257?

```
>>> a = 256
>>> b = 256

>>> a is b
True

>>> a = 257
>>> b = 257

>>> a is b
False

>>> a, b = 257, 257

>>> a is b
True
```

in [linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

Comparing python objects can be tricky at times. Can you figure out what is going on in the above code example? Answer below:

---

When we run Python, it pre-loads a global list of integers in the range [-5, 256]. Every time an integer is referenced in this range, Python does not create a new object. Instead, it uses the cached version.

This is done for optimization purposes. It was considered that these numbers are used a lot by programmers. Therefore, it would make sense to have them ready at startup.



However, referencing any integer beyond 256 (or before -5) will create a new object every time.

In the last example, when a and b are set to 257 in the same line, the Python interpreter creates a new object. Then it references the second variable with the same object.

Share this post on LinkedIn: [Post Link](#).

The below image should give you a better understanding:

```
a = 256  
b = 256  
print(a is b) # Output: True
```

>>> id(a), id(b)  
(1284834, 1284834) **Same**

```
a = 257  
b = 257  
print(a is b) # Output: False
```

>>> id(a), id(b)  
(1848542, 1744933) **Different**

```
>>> print(a==b) # Output: True
```

```
a, b = 257, 257  
print(a is b) # Output: True
```

>>> id(a), id(b)  
(23829574, 23829574) **Same**

in [linkedin.com/in/avi-chawla](https://linkedin.com/in/avi-chawla)



# Make a Class Object Behave Like a Function

The diagram illustrates two code snippets demonstrating how to make a class object callable.

**Top Snippet:** A screenshot of a terminal window showing Python code for a `Quadratic` class. The code defines an `__init__` method to initialize attributes `a`, `b`, and `c`, and a `__call__` method to calculate the quadratic function  $y = ax^2 + bx + c$ . A callout bubble points to the `__call__` method with the text "define \_\_call\_\_ method".

```
class Quadratic:
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c

    def __call__(self, x):
        return (self.a * x**2) +
               (self.b * x) +
               self.c
```

**Bottom Snippet:** A screenshot of a terminal window showing the execution of the `Quadratic` class. It creates an instance `f` with parameters `(1, 2, 3)`, prints the value at `x=1` (output: 6), prints the value at `x=2` (output: 11), and checks if the object is callable (output: True). A callout bubble points to the code with the text "class object behaves like function".

```
f = Quadratic(1, 2, 3)
print(f(1)) # Output: 6
print(f(2)) # Output: 11
print(callable(f)) # Output: True
```

**LinkedIn Profile:** A link to the author's LinkedIn profile: [linkedin.com/in/avi-chawla](https://linkedin.com/in/avi-chawla)

If you want to make a class object callable, i.e., behave like a function, you can do so by defining the `__call__` method.

This method allows you to define the behavior of the object when it is invoked like a function.



This can have many advantages. For instance, it allows us to implement objects that can be used in a flexible and intuitive way. What's more, the familiar function-call syntax, at times, can make your code more readable.

Lastly, it allows you to use a class object in contexts where a callable is expected. Using a class as a decorator, for instance.



## Lesser-known feature of Pickle Files

```
dump.py
import pickle

a, b, c = 1, 2, 3

with open("data.pkl", "wb") as f:
    pickle.dump(a, f)
    pickle.dump(b, f)
    pickle.dump(c, f)
```

Store 3 Variables ←

```
load.py
import pickle

with open("data.pkl", "rb") as f:
    a = pickle.load(f)
    b = pickle.load(f)

print(f"{a} {b}")
## a = 1 b = 2
```

Load Only First 2 → Variables

[linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

Pickles are widely used to dump data objects to disk. But folks often dump just a single object into a pickle file. Moreover, one creates multiple pickles to store multiple objects.

However, did you know that you can store as many objects as you want within a single pickle file? What's more, when reloading, it is not necessary to load all the objects.

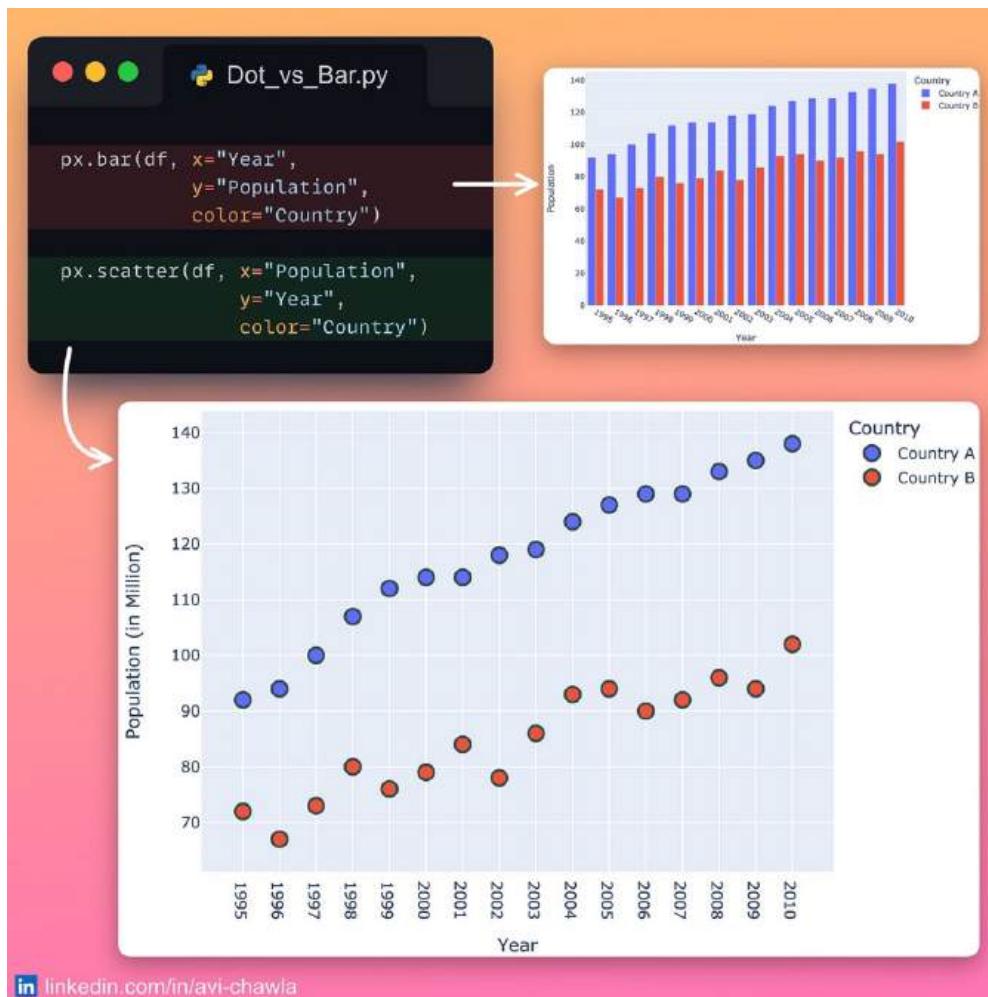
Just make sure to dump the objects within the same context manager (using **with**).



Of course, one solution is to store the objects together as a tuple. But while reloading, the entire tuple will be loaded. This may not be desired in some cases.



# Dot Plot: A Potential Alternative to Bar Plot



Bar plots are extremely useful for visualizing categorical variables against a continuous value. But when you have many categories to depict, they can get too dense to interpret.

In a bar plot with many bars, we're often not paying attention to the individual bar lengths. Instead, we mostly consider the individual endpoints that denote the total value.

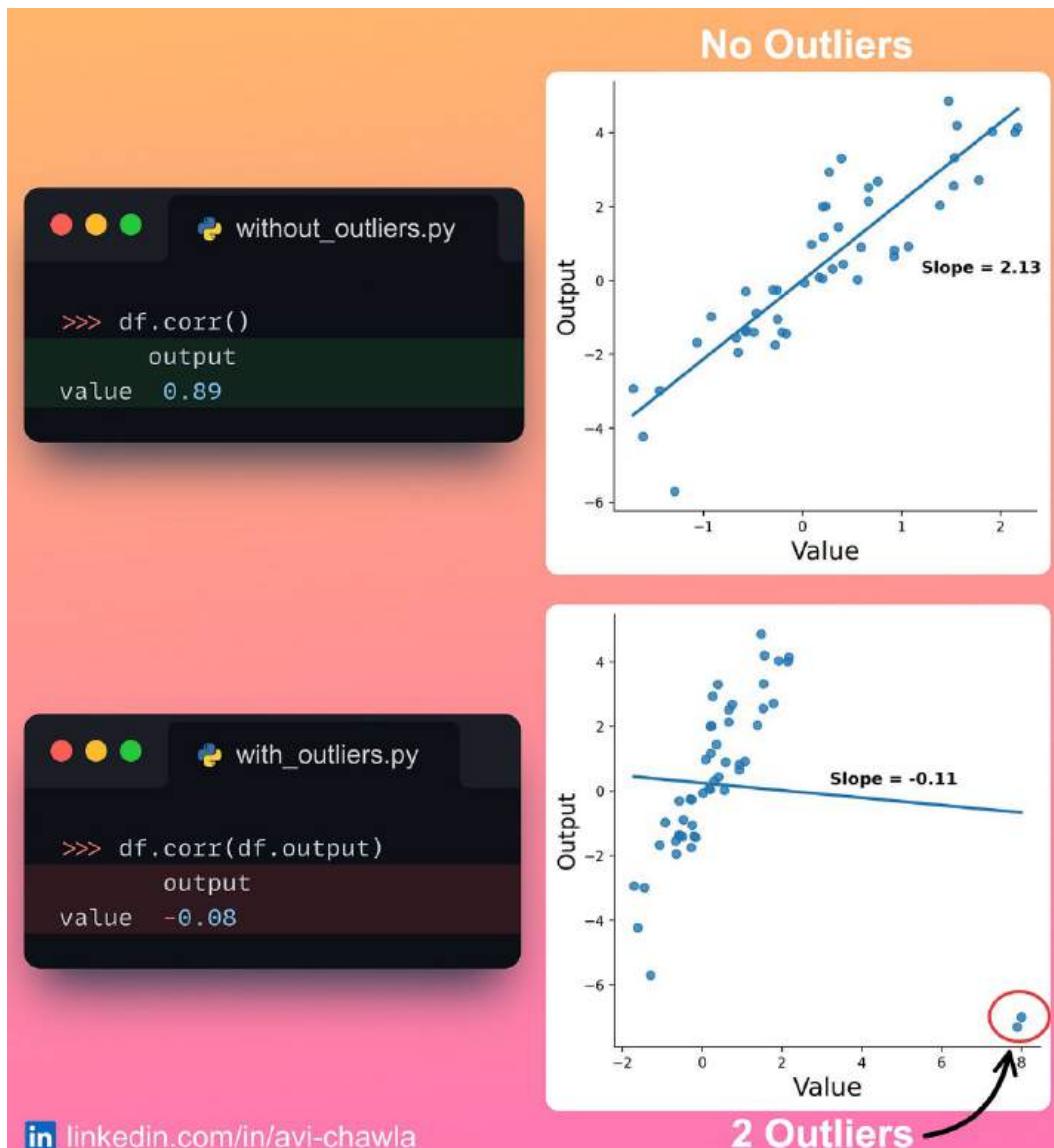
A Dot plot can be a better choice in such cases. They are like scatter plots but with one categorical and one continuous axis.

Compared to a bar plot, they are less cluttered and offer better comprehension. This is especially true in cases where we have many categories and/or multiple categorical columns to depict in a plot.

Read more: [Documentation](#).



# Why Correlation (and Other Statistics) Can Be Misleading.



Correlation is often used to determine the association between two continuous variables. But it has a major flaw that often gets unnoticed.

Folks often draw conclusions using a correlation matrix without even looking at the data. However, the obtained statistics could be heavily driven by outliers or other artifacts.

This is demonstrated in the plots above. The addition of just two outliers changed the correlation and the regression line drastically.

Thus, looking at the data and understanding its underlying characteristics can save from drawing wrong conclusions. Statistics are important, but they can be highly misleading at times.



# Supercharge value\_counts() Method in Pandas With Sidetable

```
sidetable.py
```

```
import sidetable

df.stb.freq(["City"],
            style=True)
```

	City	count	percent	cumulative_count	cumulative_percent
0	West Jamesview	120	12.00%	120	12.00%
1	Aliciafort	113	11.30%	233	23.30%
2	Ricardomouth	106	10.60%	339	33.90%
3	New Cindychester	106	10.60%	445	44.50%
4	Whiteside	104	10.40%	549	54.90%
5	Kristaburgh	97	9.70%	646	64.60%
6	Wardfort	96	9.60%	742	74.20%
7	New Russellton	93	9.30%	835	83.50%
8	Whitakerbury	87	8.70%	922	92.20%
9	North Melissafurt	78	7.80%	1,000	100.00%

[linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

The **value\_counts()** method is commonly used to analyze categorical columns, but it has many limitations.

For instance, if one wants to view the percentage, cumulative count, etc., in one place, things do get a bit tedious. This requires more code and is time-consuming.

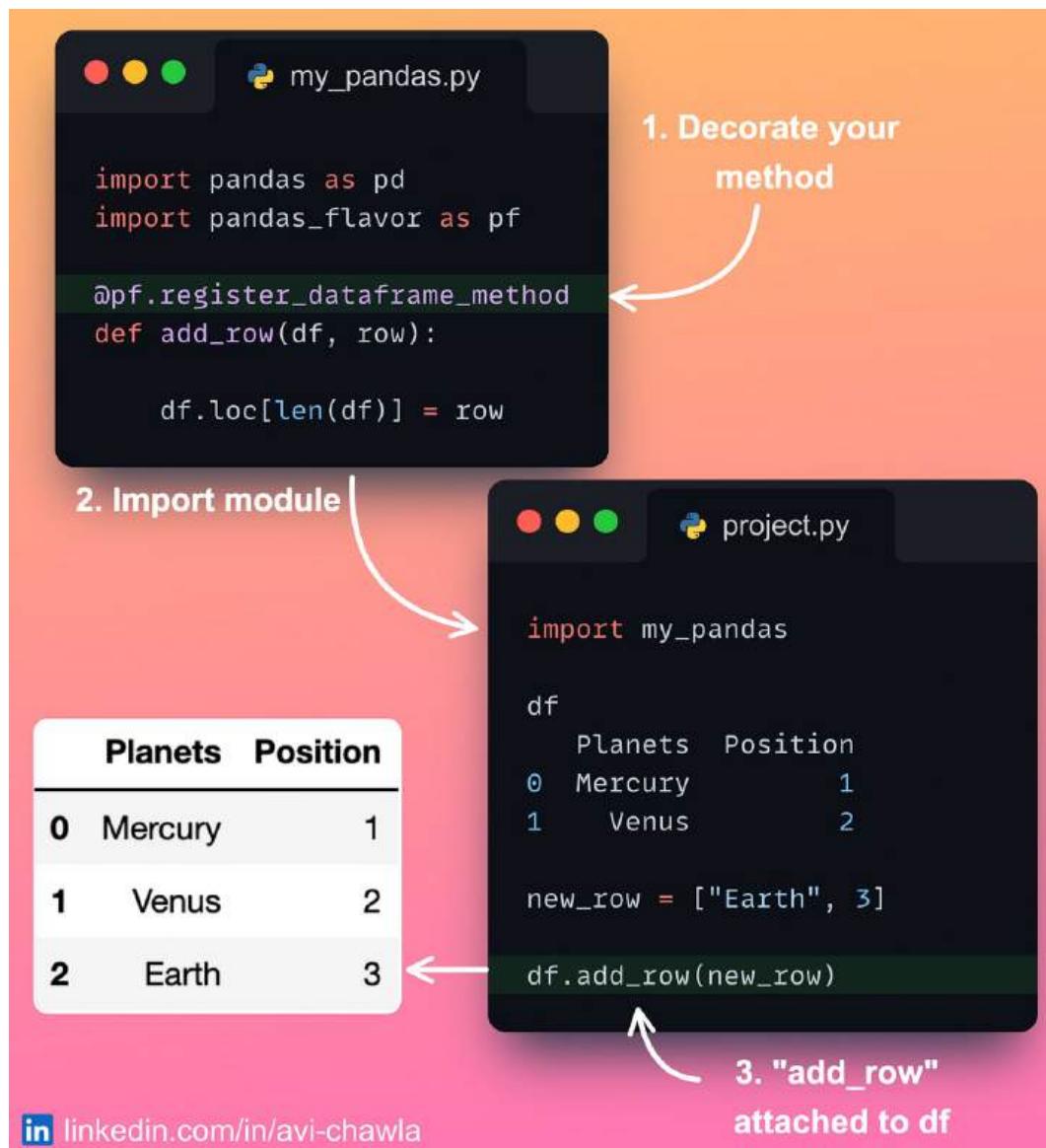
Instead, use **sidetable**. Consider it as a supercharged version of **value\_counts()**. As shown below, the **freq()** method from sidetable provides a more useful summary than **value\_counts()**.

Additionally, sidetable can aggregate multiple columns too. You can also provide threshold points to merge data into a single bucket. What's more, it can print missing data stats, pretty print values, etc.

Read more: [GitHub](#).



# Write Your Own Flavor Of Pandas



If you want to attach a custom functionality to a Pandas DataFrame (or series) object, use "pandas-flavor".

Its decorators allow you to add methods directly to the Pandas' object.

This is especially useful if you are building an open-source project involving Pandas. After installing your library, others can access your library's methods using the `dataframe` object.

P.S. This is how we see `df.progress_apply()` from `tqdm`, `df.parallel_apply()` from `Pandarallel`, and many more.

Read more: [Documentation](#).



# CodeSquire: The AI Coding Assistant You Should Use Over GitHub Copilot

The screenshot shows the CodeSquire.ai interface. At the top right is the logo and text "CodeSquire.ai". Below it is a code editor window containing Python code for a titanic dataset:

```
from catboost.datasets import titanic
import numpy as np
import pandas as pd
from catboost import CatBoostClassifier

train_df, test_df = titanic()

train_df.head()

# one hot encode all the categorical vars in train_df and test_df
train_df = pd.get_dummies(train_df, columns=['Sex', 'Embarked'])
test_df = pd.get_dummies(test_df, columns=['Sex', 'Embarked'])
```

To the right of the code editor, there are two annotations: "Write comment" with a downward arrow pointing to the code, and "Output" with an upward arrow pointing from the bottom of the code editor towards the output area.

At the bottom left is a LinkedIn link: [linkedin.com/in/avi-chawla](https://linkedin.com/in/avi-chawla).

Coding Assistants like GitHub Copilot are revolutionary as they offer many advantages. Yet, Copilot has limited utility for data professionals. This is because it's incompatible with web-based IDEs (Jupyter/Colab).

Moreover, in data science, the subsequent exploratory steps are determined by previous outputs. But Copilot does not consider that (and even markdown cells) to drive its code suggestions.

[CodeSquire](#) is an incredible AI coding assistant that addresses the limitations of Copilot. The good thing is that it has been designed specifically for data scientists, engineers, and analysts.

Besides seamless code generation, it can generate SQL queries from text and explain code. You can leverage AI-driven code generation by simply installing a browser extension.

Read more: [CodeSquire](#).

Watch a video version of this post on LinkedIn: [Post Link](#).



# Vectorization Does Not Always Guarantee Better Performance

The screenshot shows a Jupyter Notebook interface with two code cells and a data frame preview.

**Code Cell 1:** Vectorized.py

```
df.Name.str.split()  
## 1.7 s
```

**Code Cell 2:** Non-vectorized.py

```
def name_split(s):  
    return s.split()  
  
df.Name.apply(name_split)  
## 862 ms
```

**Data Frame Preview:**

	df	Name
0		Beth Alvarez
1		Deborah Watkins
2		Jeffrey Compton
3		Alan Wolfe
4		Kathryn Gordon

[in](https://linkedin.com/in/avi-chawla) [linkedin.com/in/avi-chawla](https://linkedin.com/in/avi-chawla)

Vectorization is well-adopted for improving run-time performance. In a nutshell, it lets you operate data in batches instead of processing a single value at a time.

Although vectorization is extremely effective, you should know that it does not always guarantee performance gains. Moreover, vectorization is also associated with memory overheads.

As demonstrated above, the non-vectorized code provides better performance than the vectorized version.

P.S. `apply()` is also a for-loop.

Further reading: [Here](#).



# In Defense of Match-case Statements in Python

The diagram illustrates the comparison between two Python code snippets: `if-else.py` and `match-case.py`.

**if-else.py:**

```
def make_point(point):
    if isinstance(point, (tuple, list)):
        if len(point) == 2:
            x, y = point
            return Point3D(x, y, 0)

        elif len(point) == 3:
            x, y, z = point
            return Point3D(x, y, z)

        else:
            raise TypeError("Unsupported")
    else:
        raise TypeError("Unsupported")
```

Annotations for `if-else.py`:

- Check type (points to the `isinstance` check)
- Check length (points to the `len` checks)
- Explicit unpacking (points to the tuple unpacking in the `elif` block)

**No type checks** (indicated by a curved arrow pointing from this text to the `isinstance` check in the `if-else` code)

**No length checks**

**No unpacking**

**match-case.py:**

```
def make_point(point):
    match point:
        case (x, y):
            return Point3D(x, y, 0)

        case (x, y, z):
            return Point3D(x, y, z)

        case _: ## Default
            raise TypeError("Unsupported")

    >>> make_point((1, 2))
    Point3D(x=1, y=2, z=0)

    >>> make_point([1, 2, 3])
    Point3D(x=1, y=2, z=0)

    >>> make_point((1, 2, 3, 4))
    TypeError: Unsupported
```

[in/avi-chawla](https://linkedin.com/in/avi-chawla)



I recently came across a post on **match-case** in Python. In a gist, starting Python 3.10, you can use **match-case** statements to mimic the behavior of **if-else**.

Many responses on that post suggested that **if-else** offers higher elegance and readability. Here's an example in defense of **match-case**.

While if-else is traditionally accepted, it also comes with many downsides. For instance, many-a-times, one has to write complex chains of nested if-else statements. This includes multiple calls to **len()**, **isinstance()** methods, etc.

Furthermore, with **if-else**, one has to explicitly destructure the data to extract values. This makes your code inelegant and messy.

Match-case, on the other hand, offers Structural Pattern Matching which makes this simple and concise. In the example above, match-case automatically handles type-matching, length check, and variable unpacking.

Read more here: [Python Docs](#).



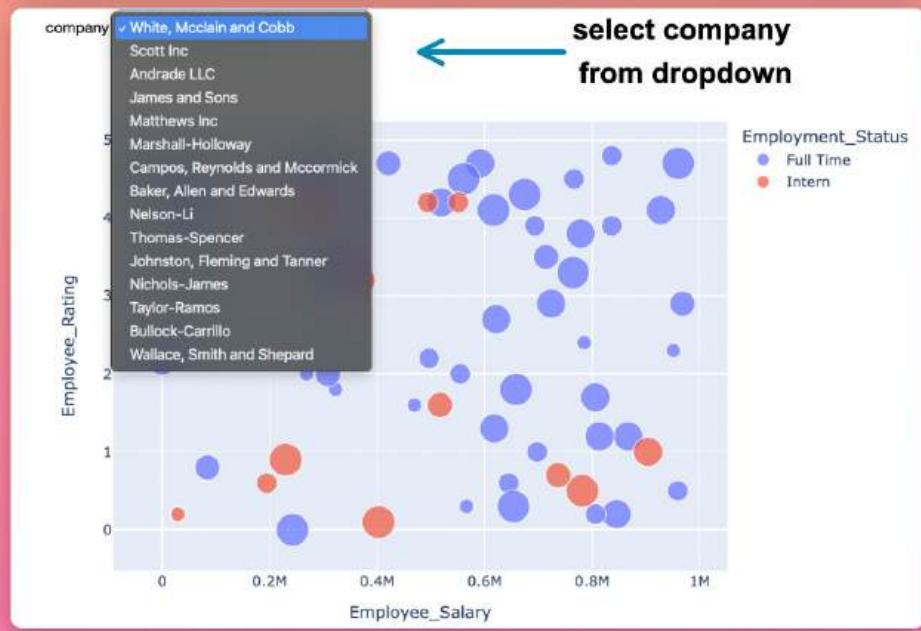
# Enrich Your Notebook With Interactive Controls

```
from ipywidgets import interact

@interact
def plot_company(company = list(df.Company.unique())):

    ## filter the DataFrame
    df_filtered = df[df.Company == company]

    ## Create the plot on df_filtered
    df_filtered.plot(...)
```



[in linkedin.com/in/avi-chawla](https://linkedin.com/in/avi-chawla)

While using Jupyter, we often re-run the same cell repeatedly after changing the input slightly. This is time-consuming and also makes your data exploration tasks tedious and unorganized.



Instead, pivot towards building interactive controls in your notebook. This allows you to alter the inputs without needing to rewrite and re-run your code.

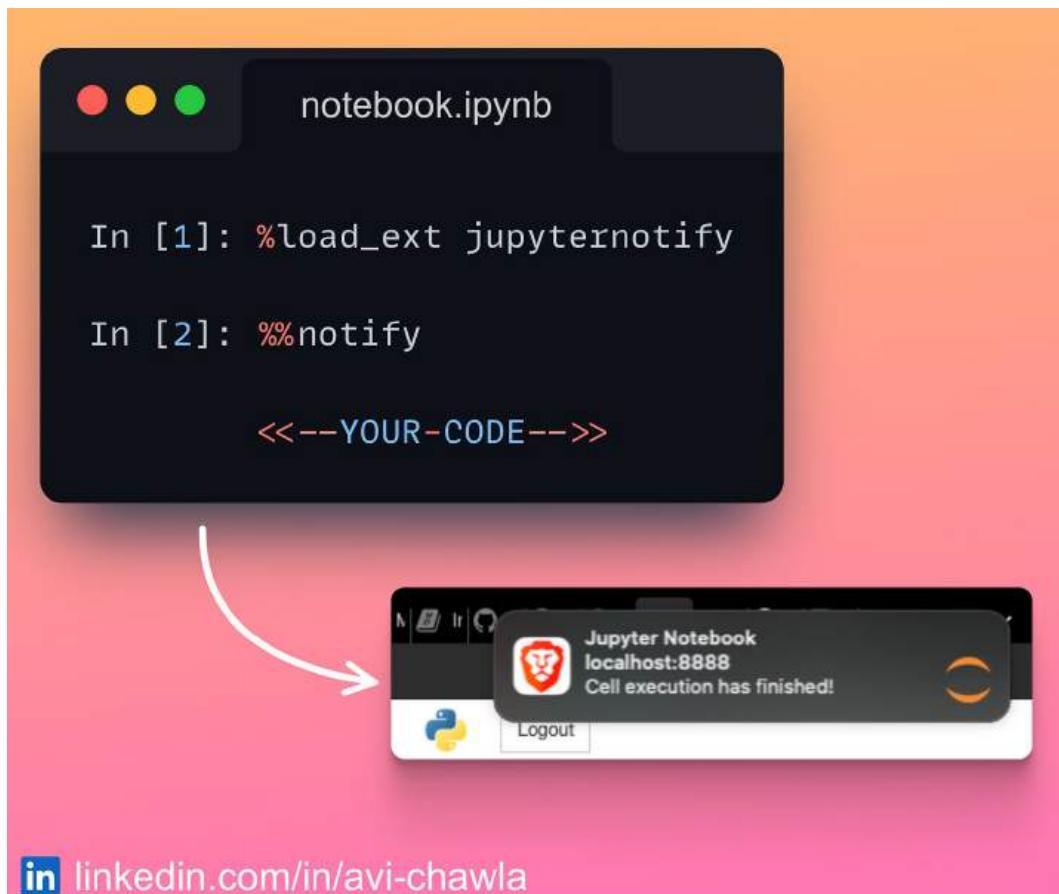
In Jupyter, you can do this using the **IPywidgets** module. Embedding interactive controls is as simple as using a decorator.

As a result, it provides you with interactive controls such as dropdowns and sliders. This saves you from tons of repetitive coding and makes your notebook organized.

Watch a video version of this post on LinkedIn: [Post Link](#).



# Get Notified When Jupyter Cell Has Executed



[in linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

After running some code in a Jupyter cell, we often navigate away to do some other work in the meantime.

Here, one has to repeatedly get back to the Jupyter tab to check whether the cell has been executed or not.

To avoid this, you can use the **%%notify** magic command from the **jupyternotify** extension. As the name suggests, it notifies the user upon completion (both successful and unsuccessful) of a jupyter cell via a browser notification. Clicking on the notification takes you back to the jupyter tab.

Read more: [GitHub](#).



# Data Analysis Using No-Code Pandas In Jupyter

In [1]:

```
1 import mitosheet
2 mitosheet.sheet(analyses_to_replay="id-ymyxvhaces")
executed in 4.97s, finished 14:48:34 2022-11-22
```

The screenshot shows a Jupyter Notebook cell with the code above. Below the cell is a Mito spreadsheet interface. The spreadsheet has a header row with columns: Employee\_City | Employee\_City. The data starts with row 99 and continues down to row 53. The columns are: Name, Company\_Nam, Employee\_City, Employee\_Sal, Employment\_Status, Employee\_Rat. The data includes various employee details like name, company, city, salary, employment status, and rating.

	Name	Company_Nam	Employee_City	Employee_Sal	Employment_Status	Employee_Rat
99	Christopher Jones	Matthews Inc	Aliciafort	5,187.04	Full Time	1.50
96	Mitchell Hill	Baker, Allen and Edwin	Aliciafort	4,078.71	Full Time	1.40
44	Dawn Bailey	White, Mcclain and C	Aliciafort	11,379.39	Full Time	4.50
80	Donald Bowman	Scott Inc	Aliciafort	4,292.43	Full Time	1.30
48	Kelly Liu	Matthews Inc	Aliciafort	4,413.51	Intern	0.90
20	David Mills	Johnston, Fleming an	Aliciafort	6,917.60	Intern	1.90
75	Vanessa Lamb	Taylor-Ramos	Aliciafort	8,391.54	Full Time	2.70
67	Douglas Kennedy	Andrade LLC	Aliciafort	2,815.63	Full Time	0.80
54	Jeffrey Gonzalez	Taylor-Ramos	Aliciafort	10,401.33	Full Time	4.60
37	Emily Weber	Matthews Inc	Kristaburgh	7,676.39	Intern	2.30
45	Zachary Ellison	James and Sons	Kristaburgh	7,194.54	Full Time	2.60
50	Gina Acosta	Nichols-James	Kristaburgh	7,239.98	Full Time	2.10
22	Jason Reyes	Matthews Inc	Kristaburgh	6,760.68	Full Time	2.80
53	James Wright	Nelson-Li	Kristaburgh	3,960.27	Intern	1.00

(100 rows, 6 cols)

[linkedin.com/in/avi-chawla](https://linkedin.com/in/avi-chawla)

The Pandas API provides a wide range of functionalities to analyze tabular datasets.

Yet, across projects, we often use the same methods over and over to analyze our data. This quickly gets repetitive and time-consuming.

To avoid this, use Mito. It's an incredible tool that allows you to analyze your data within a spreadsheet interface in Jupyter, without writing any code.

The coolest thing about Mito is that each edit in the spreadsheet automatically generates an equivalent Python code. This makes it extremely convenient to reproduce the analysis later.

Read more: [Documentation](#).



# Using Dictionaries In Place of If-conditions

```
if_else.py
number = int(input())

if number == 1:
    func1()

elif number == 2:
    func2()

else:
    func3()
```

```
dict.py
number = int(input())

func_map = {1:func1,
            2:func2}

func_map.get(number, func3)()
```

replace with dictionary

in [linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

Default

Dictionaries are mainly used as a data structure in Python for maintaining key-value pairs.

However, there's another special use case that dictionaries can handle. This is — Eliminating IF conditions from your code.

Consider the code snippet above. Here, corresponding to an input value, we invoke a specific function. The traditional way requires you to hard-code every case.

But with a dictionary, you can directly retrieve the corresponding function by providing it with the key. This makes your code concise and elegant.



[blog.DailyDoseofDS.com](http://blog.DailyDoseofDS.com)



# Clear Cell Output In Jupyter Notebook During Run-time

The screenshot shows a Jupyter Notebook cell with the following Python code:

```
import time
from IPython.display import clear_output

for i in range(100):

    ## Wait for the next
    ## output before clearing
    clear_output(wait=True)

    print(f'Output Number {i+1}')
    time.sleep(1)
```

The cell has been executed, and the output is visible below it. The output shows the command and its execution details:

In [6]:

```
1 for i in range(100):
2
3     ## Wait for the next
4     ## output before clearing
5     clear_output(wait=True)
6
7     print(f'Output Number {i+1}')
8     time.sleep(1)
```

executed in 1m 40.6s, finished 15:55:44 2022-11-19

Output Number 100 ← Only Last Output

Below the cell, there is a LinkedIn profile link:

in [linkedin.com/in/avi-chawla](https://linkedin.com/in/avi-chawla)

While using Jupyter, we often print many details to track the code's progress.

However, it gets frustrating when the output panel has accumulated a bunch of details, but we are only interested in the most recent output. Moreover, scrolling to the bottom of the output each time can be annoying too.

To clear the output of the cell, you can use the **clear\_output** method from the **IPython** package. When invoked, it will remove the current output of the cell, after which you can print the latest details.



# A Hidden Feature of Describe Method In Pandas

The image shows a Jupyter Notebook interface with two code cells and their corresponding outputs.

**Only Numerical Columns:**

```
>>> df
      col1  col2  col3  col4
0      1      2      A      D
1      3      4      B      E
2      5      6      A      E

>>> df.describe()
```

	col1	col2
count	3.0	3.0
mean	3.0	4.0
std	2.0	2.0
min	1.0	2.0
25%	2.0	3.0
50%	3.0	4.0
75%	4.0	5.0
max	5.0	6.0

**All Columns:**

```
>>> df.describe(include = "all")
```

	col1	col2	col3	col4
count	3.0	3.0	3	3
unique	NaN	NaN	2	2
top	NaN	NaN	A	E
freq	NaN	NaN	2	2
mean	3.0	4.0	NaN	NaN
std	2.0	2.0	NaN	NaN
min	1.0	2.0	NaN	NaN
25%	2.0	3.0	NaN	NaN
50%	3.0	4.0	NaN	NaN
75%	4.0	5.0	NaN	NaN
max	5.0	6.0	NaN	NaN

[linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

The **describe()** method in Pandas is commonly used to print descriptive statistics about the data.

But have you ever noticed that its output is always limited to numerical columns? Of course, details like mean, median, std. dev., etc. hold no meaning for non-numeric columns, so the results make total sense.

However, **describe()** can also provide a quick summary of non-numeric columns. You can do this by specifying **include="all."** As a result, it will return the number of unique elements, the top element with its frequency.

Read more: [Documentation](#).



# Use Slotted Class To Improve Your Python Code

The image shows two code editors side-by-side. The top editor, titled 'Without\_slots.py', contains the following code:

```
class Person:
    def __init__(self, name, age):
        self.Name = name
        self.Age = age
    person = Person('Mike', 22)
    person.name = 'Peter'
## No Error
```

A callout bubble points to the line 'person.name = 'Peter'' with the text: "'Name' mistakenly written as 'name' raises no error'.

The bottom editor, titled 'With\_slots.py', contains the following code:

```
class Person:
    __slots__ = ['Name', 'Age']
    def __init__(self, name, age):
        self.Name = name
        self.Age = age
    person = Person('Mike', 22)
    person.name = 'Peter'
## AttributeError: 'Person' object has no attribute 'name'
```

A callout bubble points to the line 'defining \_\_slots\_\_ raises AttributeError' with an arrow pointing to the line 'def \_\_init\_\_(self, name, age):'.

[linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

If you want to fix the attributes a class can hold, consider defining it as a slotted class.

While defining classes, `__slots__` allows you to explicitly specify the class attributes. This means you cannot randomly add new attributes to a slotted class object. This offers many advantages.

For instance, slotted classes are memory efficient and they provide faster access to class attributes. What's more, it also helps you avoid common typos. This, at times, can be a costly mistake that can go unnoticed.

Read more: [StackOverflow](#).



# Stop Analysing Raw Tables. Use Styling Instead!

The image shows a Jupyter Notebook interface with two code cells and their corresponding outputs.

**Code Cell 1:** df\_bar.py

```
data.style.bar(  
    color='lightgreen',  
    subset='Count')
```

**Code Cell 2:** df\_gradient.py

```
data.style.background_gradient(  
    cmap='Blues',  
    subset='Count')
```

**Output 1 (df\_bar.py):** A bar chart where the bars for 'AED' and 'GHS' are light green, while others are white.

	Currency	Count
0	AED	1132
1	USD	315
2	INR	700
3	EUR	522
4	SGD	261
5	GBP	719
6	ARS	97
7	AUD	604
8	NZD	411
9	AZN	516
10	BRL	712
11	GHS	111

**Output 2 (df\_gradient.py):** A heatmap where the values are color-coded according to a Blues colormap, with higher values (like AED at 1132) being dark blue and lower values (like GHS at 111) being light blue.

	Currency	Count
0	AED	1132
1	USD	315
2	INR	700
3	EUR	522
4	SGD	261
5	GBP	719
6	ARS	97
7	AUD	604
8	NZD	411
9	AZN	516
10	BRL	712
11	GHS	111

[in](https://linkedin.com/in/avi-chawla) linkedin.com/in/avi-chawla

Jupyter is a web-based IDE. Thus, whenever you print/display a DataFrame in Jupyter, it is rendered using HTML and CSS.

This means you can style your output in many different ways.

To do so, use the Styling API of Pandas. Here, you can make many different modifications to a DataFrame's styler object (**df.style**). As a result, the DataFrame will be displayed with the specified styling.

Styling makes these tables visually appealing. Moreover, it allows for better comprehensibility of data than viewing raw tables.

Read more here: [Documentation](#).



# Explore CSV Data Right From The Terminal

The screenshot shows a terminal window with three tabs, each displaying a command and its output. The background of the terminal has a gradient from orange at the top to pink at the bottom.

- Excel to CSV:**  
\$ in2csv data.xlsx > data.csv
- Column Names:**  
\$ csvcut -n data.csv  
1: Name  
2: Marks  
3: Grade
- data.csv:**

Name	Marks	Grade
Joe	95	A
Hanna	89	B
Chris	92	A
Julie	94	A
- Column Stats:**  
\$ csvstat data.csv  
2. "Marks"  
Type of data: Number  
Contains null values: False  
Unique values: 4  
Smallest value: 89  
Largest value: 95  
Sum: 370  
Mean: 92.5  
Median: 93  
StDev: 2.646
- Query:**  
\$ csvsql --query "select \* from data where Marks>90" data.csv  
| Name | Marks | Grade |  
| ----- | ----- | ----- |  
| Joe | 95 | A |  
| Chris | 92 | A |  
| Julie | 94 | A |

in [linkedin.com/in/avi-chawla](https://linkedin.com/in/avi-chawla)

If you want to quickly explore some CSV data, you may not always need to run a Jupyter session.

Rather, with "**csvkit**", you can do it from the terminal itself. As the name suggests, it provides a bunch of command-line tools to facilitate data analysis tasks.

These include converting Excel to CSV, viewing column names, data statistics, and querying using SQL. Moreover, you can also perform popular Pandas functions such as sorting, merging, and slicing.

Read more: [Documentation](#).



# Generate Your Own Fake Data In Seconds

```
from faker import Faker

fake = Faker()

>>> fake.name()
'Darrell Alexander'

>>> fake.email()
'ryanrichard@example.com'

>>> fake.address()
'205 Brown Point, West Melissaport, MN 93828'

>>> fake.company()
'Lam, Thomas and Cooper'

>>> fake.date_of_birth()
datetime.date(1973, 1, 21)

>>> fake.color_name()
'LightBlue'
```

[linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

Usually, for executing/testing a pipeline, we need to provide it with some dummy data.

Although using Python's "**random**" library, one can generate random strings, floats, and integers. Yet, being random, it does not output any meaningful data such as people's names, city names, emails, etc.

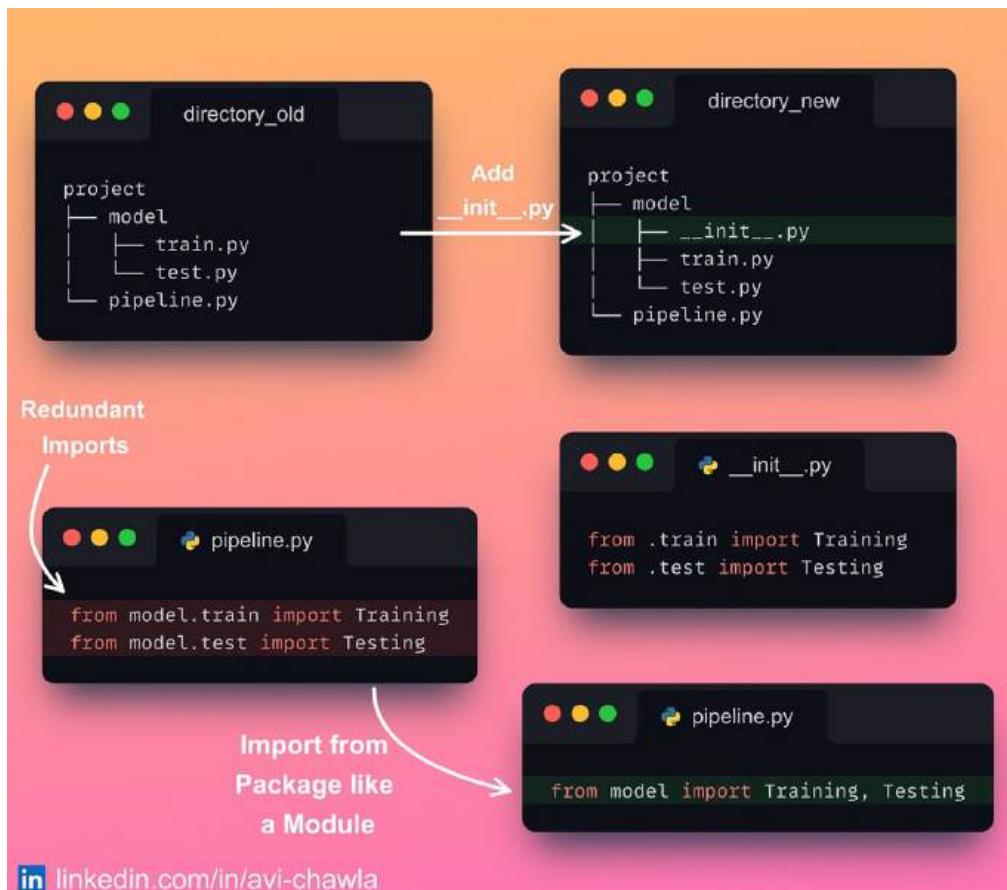
Here, looking for open-source datasets can get time-consuming. Moreover, it's possible that the dataset you find does not fit pretty well into your requirements.

The **Faker** module in Python is a perfect solution to this. Faker allows you to generate highly customized fake (yet meaningful) data quickly. What's more, you can also generate data specific to a demographic.

Read more here: [Documentation](#).



# Import Your Python Package as a Module



A python module is a single python file (`.py`). An organized collection of such python files is called a python package.

While developing large projects, it is a good practice to define an `__init__.py` file inside a package.

Consider `train.py` has a **Training** class and `test.py` has a **Testing** class.

Without `__init__.py`, one has to explicitly import them from specific python files. As a result, it is redundant to write the two import statements.

With `__init__.py`, you can group python files into a single importable module. In other words, it provides a mechanism to treat the whole package as a python module.

This saves you from writing redundant import statements and makes your code cleaner in the calling script.

Read more in this blog: [Blog Link](#).



## Specify Loops and Runs In `%timeit`

```
notebook.ipynb
In [1]: %timeit -n 1000 -r 4
        time.sleep(2)

## 2 s ± 800 µs per loop
## (mean ± std. dev. of 4 runs, 1000 loops each)
```

in [linkedin.com/in/avi-chawla](https://linkedin.com/in/avi-chawla)

We commonly use the `%timeit` (or `%%timeit`) magic command to measure the execution time of our code.

Here, `timeit` limits the number of runs depending on how long the script takes to execute. This is why you get to see a different number of loops (and runs) across different pieces of code.

However, if you want to explicitly define the number of loops and runs, use the `-n` and `-r` options. Use `-n` to specify the loops and `-r` for the number the runs.



# Waterfall Charts: A Better Alternative to Line/Bar Plot



If you want to visualize a value over some period, a line (or bar) plot may not always be an apt choice.

A line-plot (or bar-plot) depicts the actual values in the chart. Thus, sometimes, it can get difficult to visually estimate the scale of incremental changes.

Instead, you can use a waterfall chart, which elegantly depicts these rolling differences.

To create one, you can use **waterfall\_chart** in Python. Here, the start and final values are represented by the first and last bars. Also, the marginal changes are automatically color-coded, making them easier to interpret.

Read more here: [GitHub](https://github.com/avichawla/waterfall_chart).



# Hexbin Plots As A Richer Alternative to Scatter Plots



Scatter plots are extremely useful for visualizing two sets of numerical variables. But when you have, say, thousands of data points, scatter plots can get too dense to interpret.

Hexbins can be a good choice in such cases. As the name suggests, they bin the area of a chart into hexagonal regions. Each region is assigned a color intensity based on the method of aggregation used (the number of points, for instance).

Hexbins are especially useful for understanding the spread of data. It is often considered an elegant alternative to a scatter plot. Moreover, binning makes it easier to identify data clusters and depict patterns.



# Importing Modules Made Easy with Pyforest

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import sys

from sklearn.linear_model
import LinearRegression
```

```
from pyforest import *

pd.read_csv("file.csv")    ✓
np.array([1,2,3])          ✓
sys.path                  ✓
LinearRegression()         ✓
```

[in linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

The typical programming-related stuff in data science begins by importing relevant modules.

However, across notebooks/projects, the modules one imports are mostly the same. Thus, the task of importing all the individual libraries is kinda repetitive.

With **pyforest**, you can use the common Python libraries without explicitly importing them. A good thing is that it imports all the libraries with their standard conventions. For instance, **pandas** is imported with the **pd** alias.



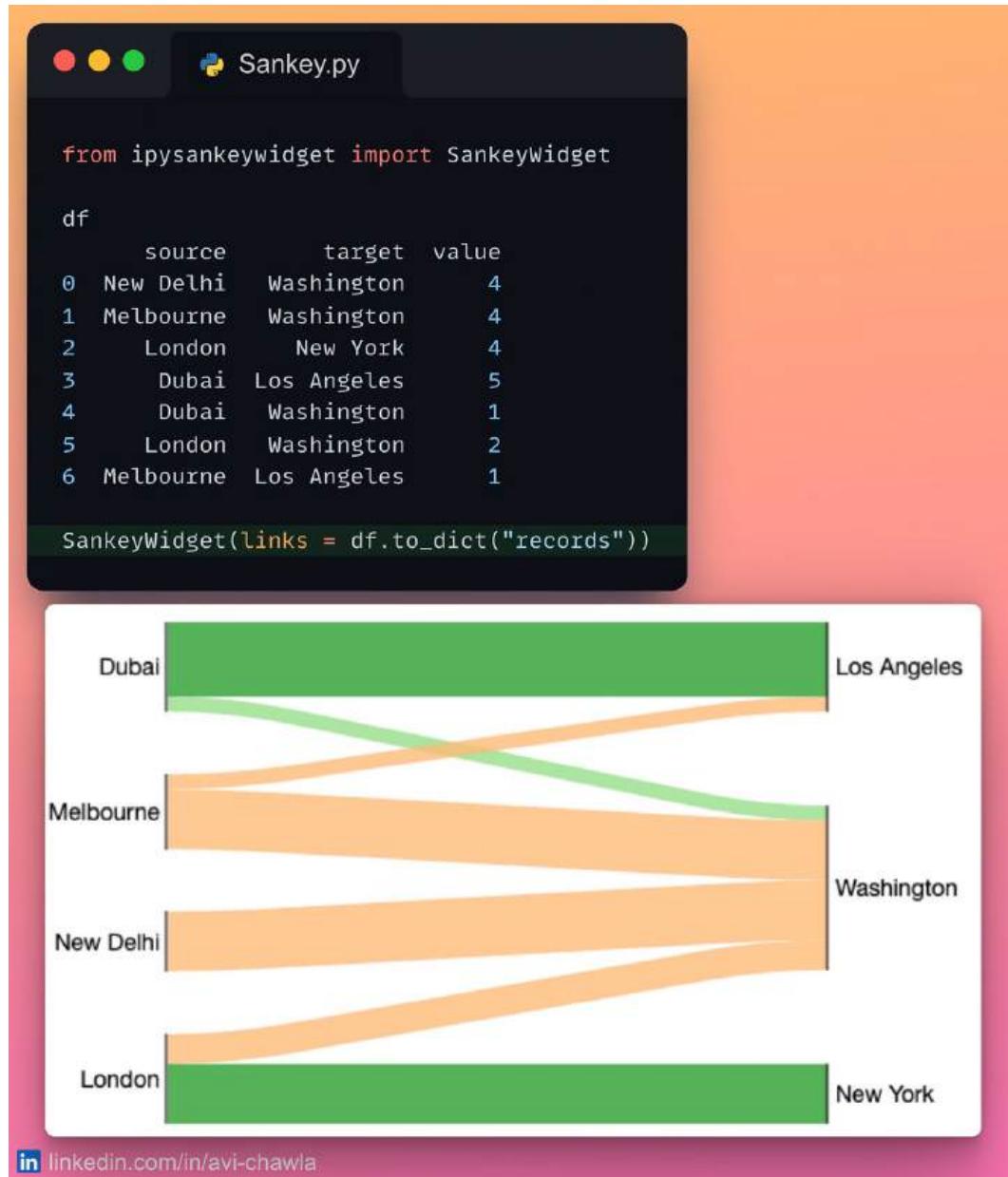
With that, you should also note that it is a good practice to keep Pyforest limited to prototyping stages. This is because once you say, develop and open-source your pipeline, other users may face some difficulties understanding it.

But if you are up for some casual experimentation, why not use it instead of manually writing all the imports?

Read more: [GitHub](#).



# Analyse Flow Data With Sankey Diagrams



Many tabular data analysis tasks can be interpreted as a flow between the source and a target.

Here, manually analyzing tabular reports/data to draw insights is typically not the right approach.

Instead, Flow diagrams serve as a great alternative in such cases.



Being visually appealing, they immensely assist you in drawing crucial insights from your data, which you may find challenging to infer by looking at the data manually.

For instance, from the diagram above, one can quickly infer that:

1. Washington hosts flights from all origins.
2. New York only receives passengers from London.
3. Majority of flights in Los Angeles come from Dubai.
4. All flights from New Delhi go to Washington.

Now imagine doing that by just looking at the tabular data. Not only will it be time-consuming, but there are chances that you may miss out on a few insights.

To generate a flow diagram, you can use floWeaver. It helps you to visualize flow data using Sankey diagrams.

Read more here: [Documentation](#).



# Feature Tracking Made Simple In Sklearn Transformers

```
numpy_output.py
```

```
from sklearn.preprocessing
import PolynomialFeatures

df
   col_A  col_B
0      1      2
1      3      4
2      5      6
```

```
array([[ 1.,  1.,  2.,  1.,  2.,  4.],
       [ 1.,  3.,  4.,  9., 12., 16.],
       [ 1.,  5.,  6., 25., 30., 36.]])
```

```
PolynomialFeatures().fit_transform(df)
```

```
pandas_output.py
```

```
from sklearn import set_config

set_config(transform_output = "pandas")
```

```
df
   col_A  col_B
0      1      2
1      3      4
2      5      6
```

	1	col_A	col_B	col_A^2	col_A*col_B	col_B^2
0	1.0	1.0	2.0	1.0	2.0	4.0
1	1.0	3.0	4.0	9.0	12.0	16.0
2	1.0	5.0	6.0	25.0	30.0	36.0

```
PolynomialFeatures().fit_transform(df)
```

[in](https://www.linkedin.com/in/avi-chawla) linkedin.com/in/avi-chawla

Recently, [scikit-learn](#) announced the release of one of the most awaited improvements. In a gist, sklearn can now be configured to output Pandas DataFrames.

Until now, Sklearn's transformers were configured to accept a Pandas DataFrame as input. But they always returned a NumPy array as an output. As a result, the output had to be manually projected back to a



Pandas DataFrame. This, at times, made it difficult to track and assign names to the features.

For instance, consider the snippet above.

In **numpy\_output.py**, it is tricky to infer the name (or computation) of a column by looking at the NumPy array.

However, in the upcoming release, the transformer can return a Pandas DataFrame (**pandas\_output.py**). This makes tracking feature names incredibly simple.

Read more: [Release page](#).



# Lesser-known Feature of f-strings in Python

```
Count = 2
Fruit = "Apple"

print(f"Count = {Count}")
print(f"Fruit = {Fruit}")
## Count = 2
## Fruit = Apple
```

Don't write variable name explicitly

```
print(f"{Count = }")
print(f"{Fruit = }")
## Count = 2
## Fruit = Apple
```

Add "=" in curly braces {}

[in linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

While debugging, one often explicitly prints the name of the variable with its value to enhance code inspection.

Although there's nothing wrong with this approach, it makes your print statements messy and lengthy.

f-strings in Python offer an elegant solution for this.

To print the name of the variable, you can add an equals sign (=) in the curly braces after the variable. This will print the name of the variable along with its value but it is concise and clean.



# Don't Use `time.time()` To Measure Execution Time

```
time.py
import time

start = time.time()
time.sleep(10)
end = time.time()

print(end - start)
## 10.00482
```

```
perf_counter.py
import time

start = time.perf_counter()
time.sleep(10)
end = time.perf_counter()

print(end - start)
## 10.00435
```

[in linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

The `time()` method from the `time` library is frequently used to measure the execution time.

However, `time()` is not meant for timing your code. Rather, its actual purpose is to tell the current time. This, at many times, compromises the accuracy of measuring the exact run time.

The correct approach is to use `perf_counter()`, which deals with relative time. Thus, it is considered the most accurate way to time your code.



# Now You Can Use DALL·E With OpenAI API

```
DALL·E API

import openai
openai.api_key = "Your-API-Key"
response = openai.Image.create(
    prompt="The city of Paris on Mars.",
    n = 2
)
image_url = response['data'][0]['url']
```

[linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

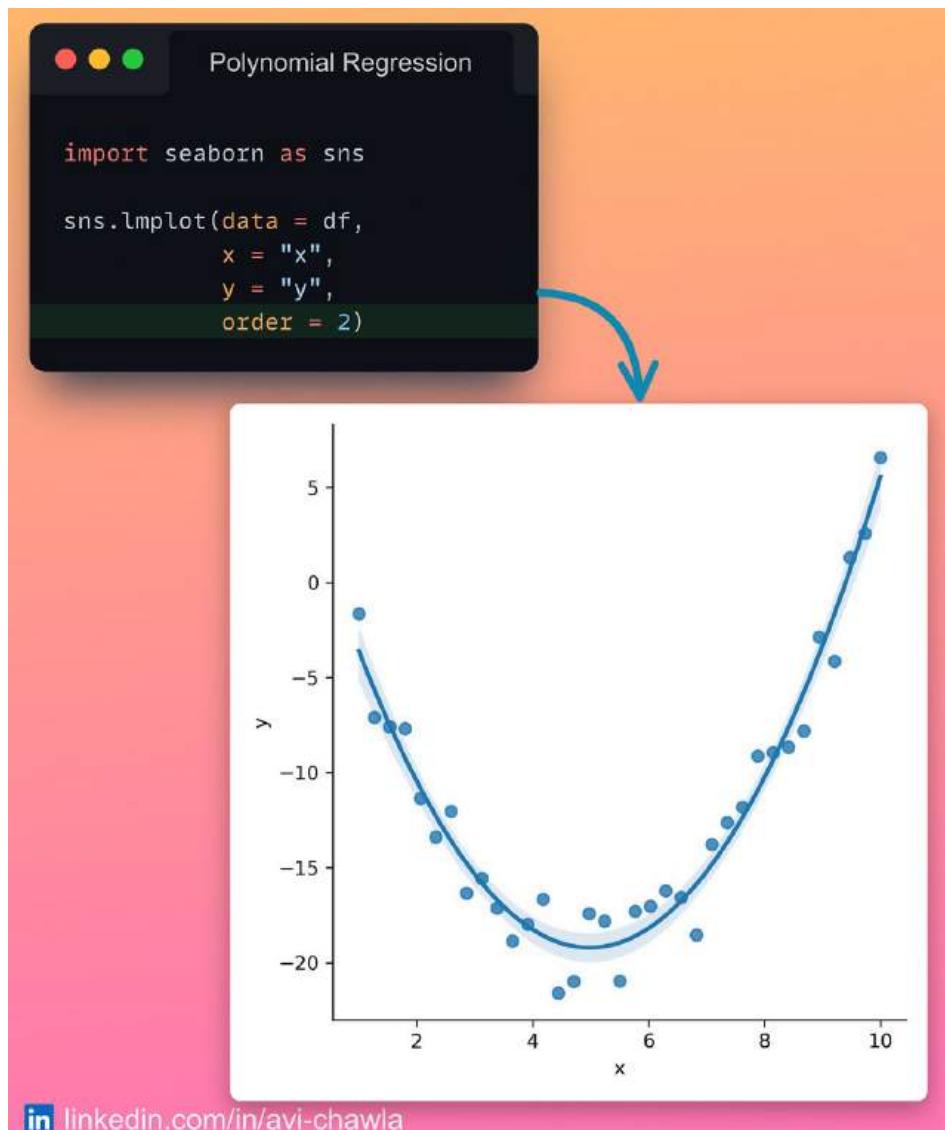
DALL·E is now accessible using the OpenAI API.

**OpenAI** recently made a big announcement. In a gist, developers can now integrate OpenAI's popular text-to-image model DALL·E into their apps using OpenAI API.

To achieve this, first, specify your API key (obtained after signup). Next, pass a text prompt to generate the corresponding image.



# Polynomial Linear Regression Plot Made Easy With Seaborn



While creating scatter plots, one is often interested in displaying a linear regression (simple or polynomial) fit on the data points.

Here, training a model and manually embedding it in the plot can be a tedious job to do.

Instead, with Seaborn's **lmplot()**, you can add a regression fit to a plot, without explicitly training a model.

Specify the degree of the polynomial as the "**order**" parameter. Seaborn will add the corresponding regression fit on the scatter plot.

Read more here: [Seaborn Docs](#).



# Retrieve Previously Computed Output In Jupyter Notebook

```
In [3]: df.groupby("col1").col2.mean().reset_index()
Out[3]:
   col1  col2
0     A    4.0
1     B    3.0
2     C    5.0

In [4]: Out[3]
Out[4]:
   col1  col2
0     A    4.0
1     B    3.0
2     C    5.0
```

in [linkedin.com/in/avi-chawla](https://linkedin.com/in/avi-chawla)

This is indeed one of the coolest things I have learned about Jupyter Notebooks recently.

Have you ever been in a situation where you forgot to assign the results obtained after some computation to a variable? Left with no choice, one has to unwillingly recompute the result and assign it to a variable for further use.

Thankfully, you don't have to do that anymore!

IPython provides a dictionary "**Out**", which you can use to retrieve a cell's output. All you need to do is specify the cell number as the dictionary's key, which will return the corresponding output. Isn't that cool?

View a video version of this post on LinkedIn: [Post Link](#).



# Parallelize Pandas Apply() With Swifter

```
Pandas Apply
df = ... ## Shape: (10M, 4)

def sum_row(row):
    return sum(row)

df.apply(sum_row, axis = 1)
```

Run-time: 35 seconds

```
Swifter Apply
import swifter

df.swifter.apply(sum_row,
                 axis = 1)
```

Run-time: 15 seconds

[in linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

The Pandas library has no inherent support to parallelize its operations. Thus, it always adheres to a single-core computation, even when other cores are idle.

Things get even worse when we use **apply()**. In Pandas, **apply()** is nothing but a glorified for-loop. As a result, it cannot even take advantage of vectorization.

A quick solution to parallelize **apply()** is to use **swifter** instead.

Swifter allows you to apply any function to a Pandas DataFrame in a parallelized manner. As a result, it provides considerable performance gains while preserving the old syntax. All you have to do is use **df.swifter.apply** instead of **df.apply**.

Read more here: [Swifter Docs](#).



# Create DataFrame Hassle-free By Using Clipboard

The screenshot illustrates a two-step process for reading a DataFrame from clipboard:

**Step 1: Copy Table** (Left): A Jupyter Notebook cell titled "Products" contains Python code to filter a DataFrame and print its head. The output shows a Pandas DataFrame with columns A, B, C, and D, and rows indexed from 0 to 7. The data is as follows:

	A	B	C	D
0	foo	one	0	0
2	foo	two	2	4
4	foo	two	4	8
6	foo	one	6	12
7	foo	three	7	14

**Step 2: Read in Pandas** (Right): A terminal window titled "Read Clipboard" shows the command `df = pd.read_clipboard()`. Below it, the resulting DataFrame is displayed with the same data as Step 1.

[linkedin.com/in/avi-chawla](https://linkedin.com/in/avi-chawla)

Many Pandas users think that a DataFrame can ONLY be loaded from disk. However, this is not true.

Imagine one wants to create a DataFrame from tabular data printed on a website. Here, they are most likely to be tempted to copy the contents to a CSV and read it using Pandas' `read_csv()` method. But this is not an ideal approach here.

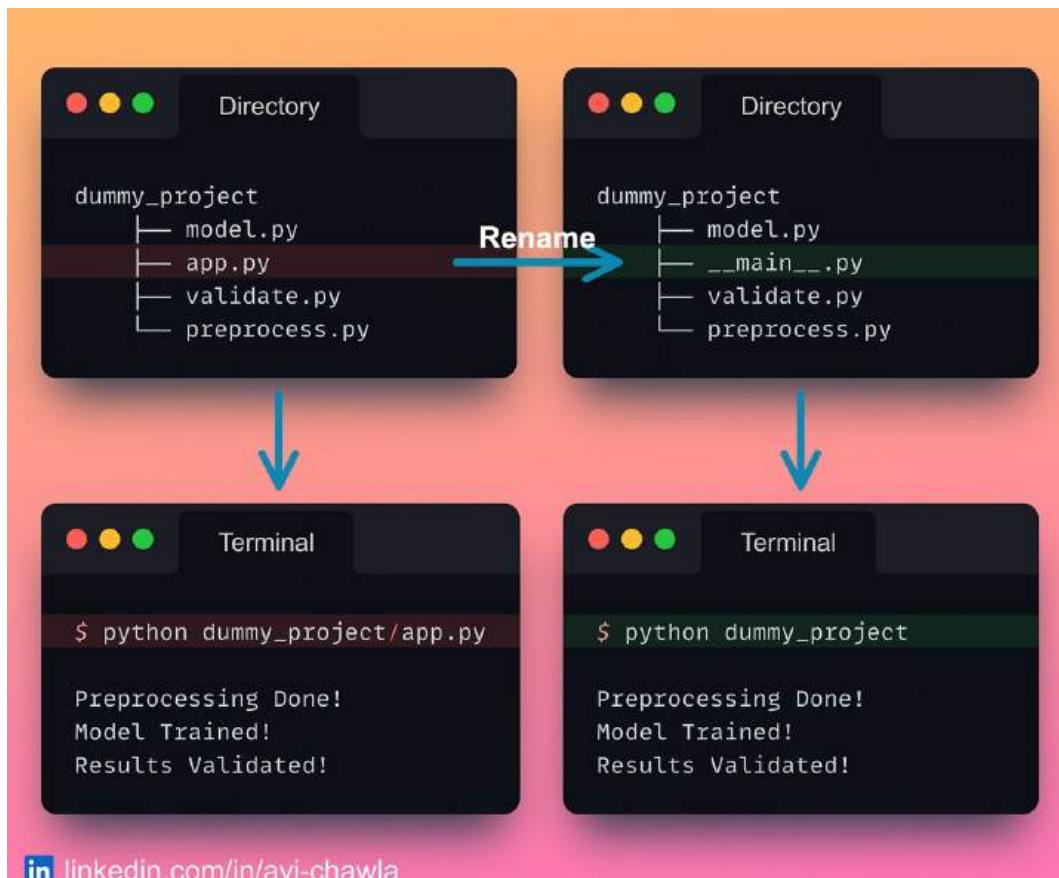
Instead, with the `read_clipboard()` method, you can eliminate the CSV step altogether.

This method allows you to create a DataFrame from tabular data stored in a clipboard buffer. Thus, you just need to copy the data and invoke the method to create a DataFrame. This is an elegant approach that saves plenty of time.

Read more here: [Pandas Docs](#).



# Run Python Project Directory As A Script



A Python script is executed when we run a **.py** file. In large projects with many files, there's often a source (or base) Python file we begin our program from.

To make things simpler, you can instead rename this base file to **\_\_main\_\_.py**. As a result, you can execute the whole pipeline by running the parent directory itself.

This is concise and also makes it slightly easier for other users to use your project.



# Inspect Program Flow with IceCream

```
file.py
```

```
1 def func():
2     print(0)
3     ...
4
5     if condition:
6         print(1)
7         ...
8     else:
9         print(2)
10    ...
```

```
Terminal
```

```
$ python file.py
0
2
```

```
file.py
```

```
1 from icecream import ic
2
3 def func():
4     ic()
5     ...
6     if condition:
7         ic()
8         ...
9     else:
10        ic()
11    ...
```

```
Terminal
```

```
$ python file.py
ic| file.py:4 in func()
ic| file.py:10 in func()
```

[in linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

While debugging, one often writes many `print()` statements to inspect the program's flow. This is especially true when we have many IF conditions.

Using empty `ic()` statements from the IceCream library can be a better alternative here. It outputs many additional details that help in inspecting the flow of the program.

This includes the line number, the name of the function, the file name, etc.

Read more in my Medium Blog: [Link](#).



# Don't Create Conditional Columns in Pandas with Apply

```
def assign_class(num):
    if num>0.5:
        return "Class A"
    else:
        return "Class B"

df.col1.apply(assign_class)
## 987 ms ± 47.1 ms per loop
```

```
import numpy as np

np.where(df["col1"]>0.5,
         "Class A",
         "Class B")
## 194 ms ± 23.7 ms per loop
```

If condition is True

If condition is False

[linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

While creating conditional columns in Pandas, we tend to use the **apply()** method almost all the time.

However, **apply()** in Pandas is nothing but a glorified for-loop. As a result, it misses the whole point of vectorization.

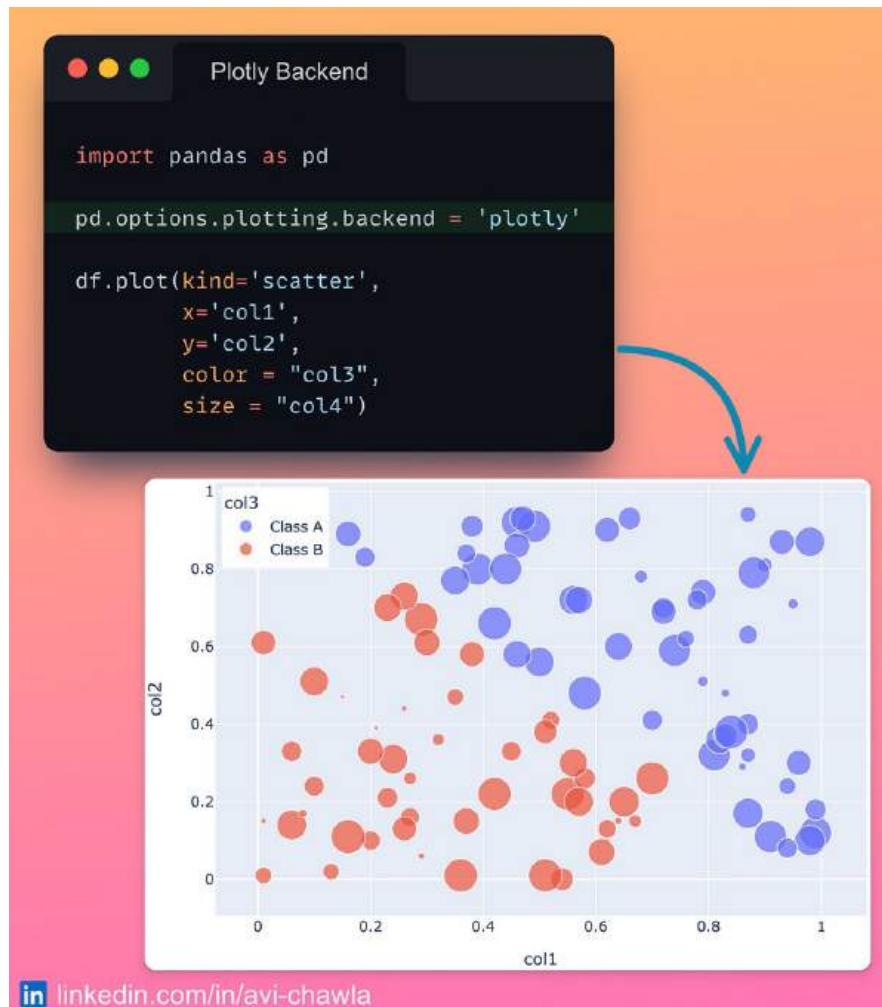
Instead, you should use the **np.where()** method to create conditional columns. It does the same job but is extremely fast.

The condition is passed as the first argument. This is followed by the result if the condition evaluates to True (second argument) and False (third argument).

Read more here: [NumPy docs](#).



# Pretty Plotting With Pandas



Matplotlib is the default plotting API of Pandas. This means you can create a Matplotlib plot in Pandas, without even importing it.

Despite that, these plots have always been basic and not so visually appealing. Plotly, with its pretty and interactive plots, is often considered a suitable alternative. But familiarising yourself with a whole new library and its syntax can be time-consuming.

Thankfully, Pandas does allow you to change the default plotting backend. Thus, you can leverage third-party visualization libraries for plotting with Pandas. This makes it effortless to create prettier plots while almost preserving the old syntax.



# Build Baseline Models Effortlessly With Sklearn

```
dummy.py
```

```
from sklearn.dummy import DummyClassifier

dummy_clf = DummyClassifier(
    strategy="most_frequent"
).fit(X, y)

>>> dummy_clf.predict(X)
array([0, 0, 0, 0, 0])

>>> dummy_clf.score(X, y)
0.6
```

[in](https://www.linkedin.com/in/avi-chawla) [linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

Before developing a complex ML model, it is always sensible to create a baseline first.

The baseline serves as a benchmark for the engineered model. Moreover, it ensures that the model is better than making random (or fixed) predictions. But building baselines with various strategies (random, fixed, most frequent, etc.) can be tedious.

Instead, Sklearn's **DummyClassifier()** (and **DummyRegressor()**) makes it totally effortless and straightforward. You can select the specific behavior of the baseline with the **strategy** parameter.

Read more here: [Documentation](#).



# Fine-grained Error Tracking With Python 3.11

```
$ python expt.py

Traceback (most recent call last):

  File "expt.py", line 11, in <module>
    print(function(a=2, b=0))
                       ^^^^^^^^^^

  File "expt.py", line 6, in function
    return (b / a) + (a / b)
                      ~~^~~

ZeroDivisionError: division by zero
```

[in](https://www.linkedin.com/in/avi-chawla) [linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

Python 3.11 was released today, and many exciting features have been introduced.

For instance, various speed improvements have been implemented. As per the official release, Python 3.11 is, on average, 25% faster than Python 3.10. Depending on your work, it can be up to 10-60% faster.

One of the coolest features is the fine-grained error locations in tracebacks.

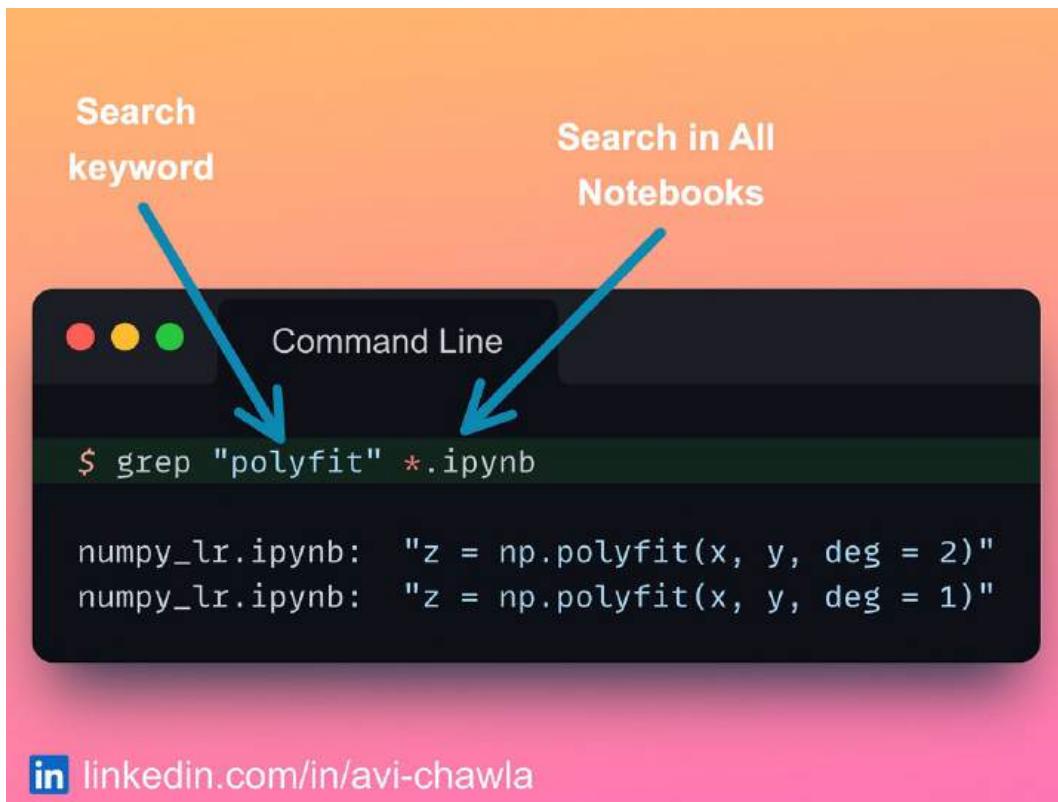
In Python 3.10 and before, the interpreter showed the specific line that caused the error. This, at many times, caused ambiguity during debugging.

In Python 3.11, the interpreter will point to the exact location that caused the error. This will immensely help programmers during debugging.

Read more here: [Official Release](#).



# Find Your Code Hiding In Some Jupyter Notebook With Ease



Programmers who use Jupyter often refer to their old notebooks to find a piece of code.

However, it gets tedious when they have multiple files to look for and can't recall the specific notebook of interest. The file name

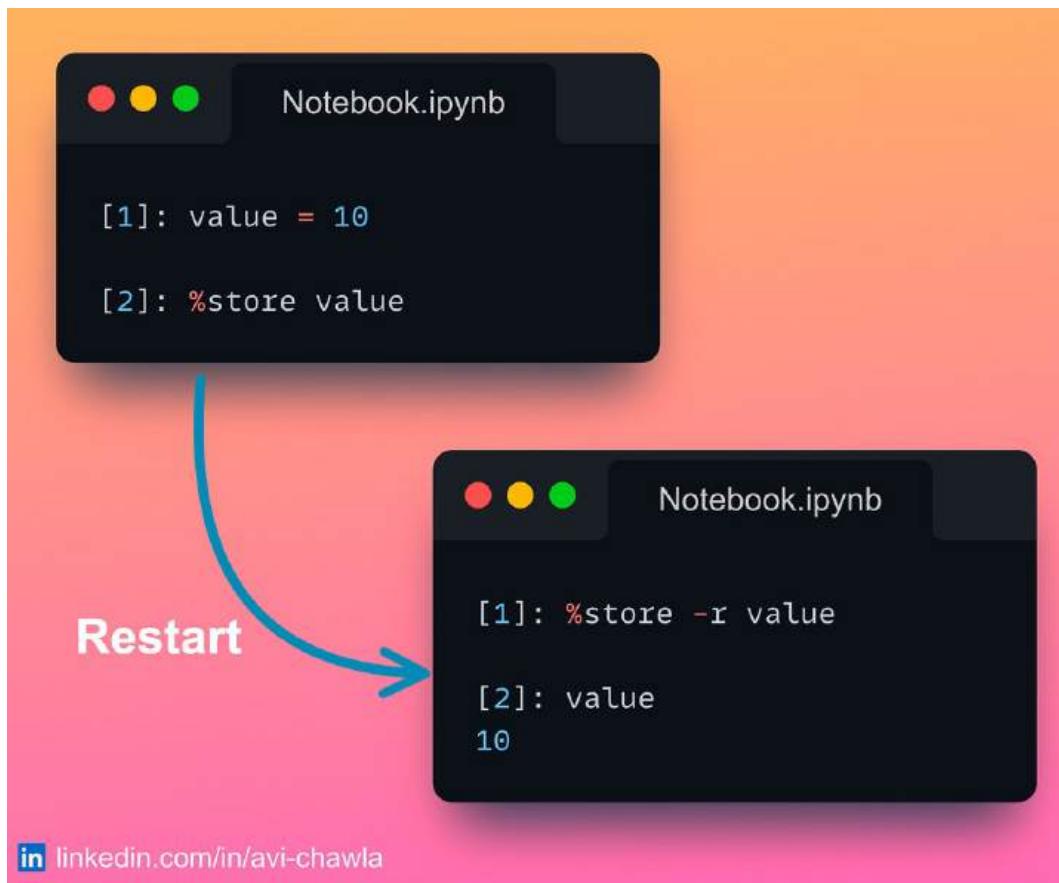
**Untitled1.ipynb**, ..., and **Untitled82.ipynb**, don't make it any easier.

The "**grep**" command is a much better solution to this. Very know that you can use "**grep**" in the command line to search in notebooks, as you do in other files (.txt, for instance). This saves plenty of manual work and time.

P.S. How do you find some previously written code in your notebooks (if not manually)?



# Restart the Kernel Without Losing Variables



While working in a Jupyter Notebook, you may want to restart the kernel due to several reasons. But before restarting, one often tends to dump data objects to disk to avoid recomputing them in the subsequent run.

The "store" magic command serves as an ideal solution to this. Here, you can obtain a previously computed value even after restarting your kernel. What's more, you never need to go through the hassle of dumping the object to disk.



# How to Read Multiple CSV Files Efficiently

```
Pandas_read.py
```

```
import pandas as pd

files = ["jan.csv", "feb.csv",
         "mar.csv", "apr.csv",
         "may.csv", "jun.csv"]
## 300 MBs each

df_list = []
for i in files:
    df_list.append(pd.read_csv(i))
data = pd.concat(df_list)
```

Run-time:  
64 seconds

```
Datatable_read.py
```

```
import datatable as dt

files = ["jan.csv", "feb.csv",
         "mar.csv", "apr.csv",
         "may.csv", "jun.csv"]
## 300 MBs each

df = dt.iread(files) ## read files
df = dt.rbind(df) ## concatenate row-wise
df = df.to_pandas() ## convert to Pandas
```

Run-time:  
36 seconds

[linkedin.com/in/avi-chawla](https://linkedin.com/in/avi-chawla)

In many situations, the data is often split into multiple CSV files and transferred to the DS/ML team for use.

As Pandas does not support parallelization, one has to iterate over the list of files and read them one by one for further processing.

"Datatable" can provide a quick fix for this. Instead of reading them iteratively with Pandas, you can use Datatable to read a bunch of files. Being parallelized, it provides a significant performance boost as compared to Pandas.

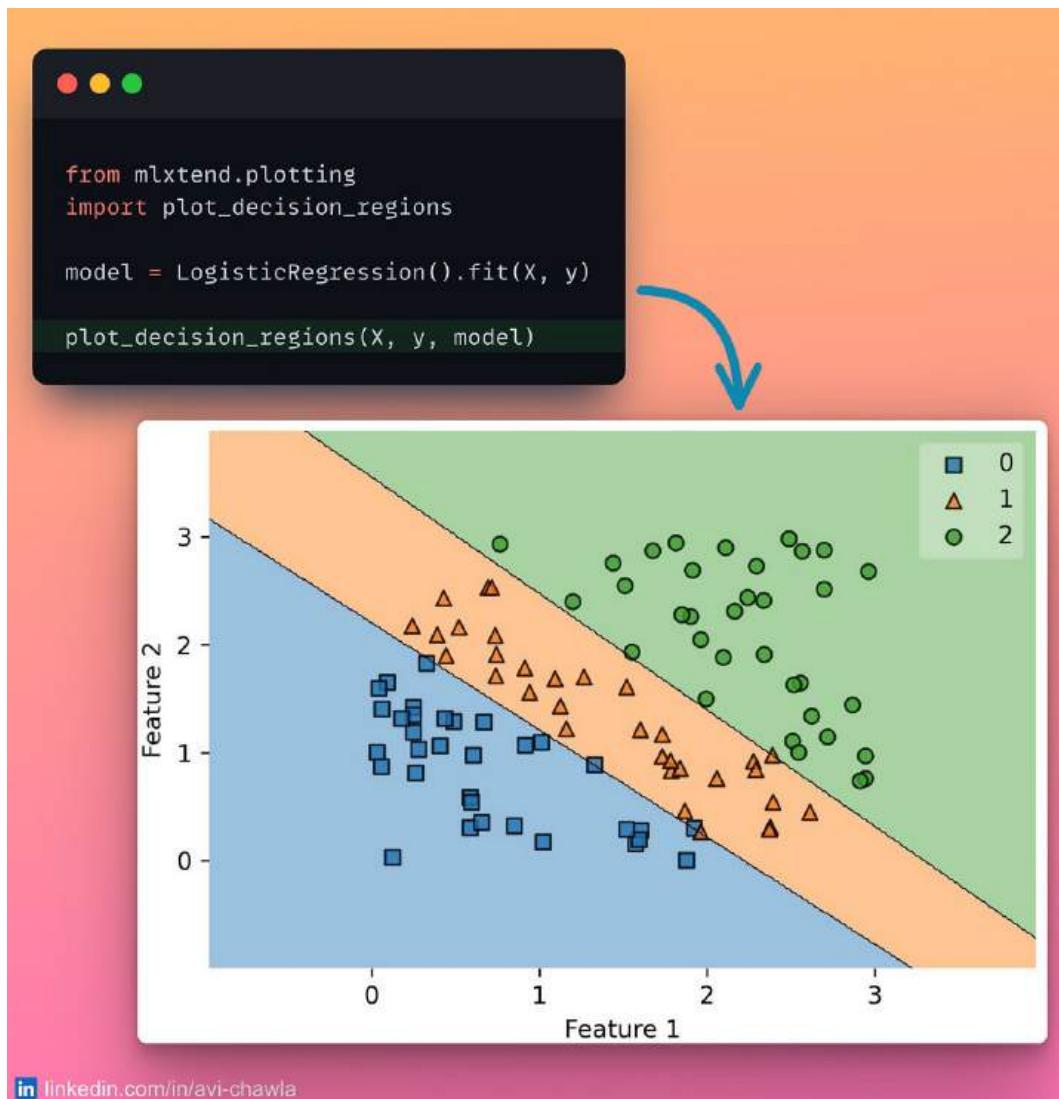


The performance gain is not just limited to I/O but is observed in many other tabular operations as well.

Read more here: [DataTable Docs.](#)



## Elegantly Plot the Decision Boundary of a Classifier



[linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

Plotting the decision boundary of a classifier can reveal many crucial insights about its performance.

Here, region-shaded plots are often considered a suitable choice for visualization purposes. But, explicitly creating one can be extremely time-consuming and complicated.

Mlxtend condenses that to a simple one-liner in Python. Here, you can plot the decision boundary of a classifier with ease, by just providing it the model and the data.



# An Elegant Way to Import Metrics From Sklearn

The image shows two Jupyter notebook cells side-by-side. The left cell demonstrates the traditional way of importing metrics from `sklearn.metrics`, while the right cell shows a more elegant approach using `get_scoring()`.

**Left Cell (Traditional Import):**

```
from sklearn.metrics
import accuracy_score, f1_score,
precision_score, recall_score,
roc_auc_score, ...  
  
>>> accuracy_score(y_true, y_pred)  
0.5  
  
>>> precision_score(y_true, y_pred)  
0.8
```

**Right Cell (Elegant Import):**

```
from sklearn.metrics import get_scorer  
  
accuracy = get_scorer("accuracy")  
>>> accuracy._score_func(y_true, y_pred)  
0.5  
  
precision = get_scorer("precision")  
>>> precision._score_func(y_true, y_pred)  
0.8
```

**Annotations:**

- A blue arrow points from the text "Import all metrics individually" to the first code cell.
- A blue double-headed curved arrow connects the text "Get a scorer from string" to the second code cell.

**LinkedIn Profile:** [linkedin.com/in/avi-chawla](https://linkedin.com/in/avi-chawla)

While using **scikit-learn**, one often imports multiple metrics to evaluate a model. Although there is nothing wrong with this practice, it makes the code inelegant and cluttered - with the initial few lines of the file overloaded with imports.

Instead of importing the metrics individually, you can use the `get_scorer()` method. Here, you can pass the metric's name as a string, and it returns a scorer object for you.

Read more here: [Scikit-learn page](#).



# Configure Sklearn To Output Pandas DataFrame

The image shows two Jupyter notebook cells side-by-side. The top cell, titled 'Scikit-learn 1.1', contains Python code that scales a Pandas DataFrame. The bottom cell, titled 'Scikit-learn 1.2.dev', shows the same code but includes a call to `scaler.set_output(transform="pandas")`. A white arrow points from the 'Output is NumPy Array' text in the 1.1 cell to the 'Output is Pandas DataFrame' text in the 1.2.dev cell.

**Scikit-learn 1.1**

```
from sklearn.preprocessing
import StandardScaler

X_train = ... ## Pandas DataFrame

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_train)

type(X_scaled) ## numpy.ndarray
```

**Scikit-learn 1.2.dev**

```
scaler = StandardScaler()
scaler.set_output(transform="pandas")
X_scaled = scaler.fit_transform(X_train)

type(X_scaled) ## pandas.core.frame.DataFrame
```

Output is NumPy Array

Output is Pandas DataFrame

[in](https://www.linkedin.com/in/avi-chawla) linkedin.com/in/avi-chawla

Recently, Scikit-learn announced the release of one of the most awaited improvements. In a gist, sklearn can now be configured to output Pandas DataFrames instead of NumPy arrays.

Until now, Sklearn's transformers were configured to accept a Pandas DataFrame as input. But they always returned a NumPy array as an output. As a result, the output had to be manually projected back to a Pandas DataFrame.

Now, the **set\_output** API will let transformers output a Pandas DataFrame instead.

This will make running pipelines on DataFrames smoother. Moreover, it will provide better ways to track feature names.



# Display Progress Bar With Apply() in Pandas

```
Without Progress
```

```
import pandas as pd
```

```
df.apply(func)
```

```
With Progress
```

```
import pandas as pd
```

```
from tqdm.notebook import tqdm
```

```
tqdm.pandas()
```

```
a = df.progress_apply(func)
```

100% [██████████] 1000000/1000000 [00:04<00:00, 281929.55it/s]

in [linkedin.com/in/avi-chawla](https://linkedin.com/in/avi-chawla)

While applying a method to a DataFrame using **apply()**, we don't get to see the progress and an estimated remaining time.

To resolve this, you can instead use **progress\_apply()** from **tqdm** to display a progress bar while applying a method.

Read more here: [GitHub](#).



# Modify a Function During Run-time

from time import sleep  
from reloading import reloading

@reloading  
def func(num):  
 if number % 2:  
 print(f"{number} is Odd")  
 else:  
 pass

for i in range(100):  
 func(i)

1 is Odd  
3 is Odd  
5 is Odd  
7 is Odd  
9 is Odd  
11 is Odd

from time import sleep  
from reloading import reloading

@reloading  
def func(num):  
 if number % 2:  
 print(f"{number} is Odd")  
 else:  
 print(f"{number} is Even")

for i in range(100):  
 func(i)

1 is Odd  
3 is Odd  
5 is Odd  
7 is Odd  
9 is Odd  
11 is Odd  
13 is Odd  
14 is Even  
15 is Odd  
16 is Even  
17 is Odd  
18 is Even  
19 is Odd

[linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

Have you ever been in a situation where you wished to add more details to an already running code?

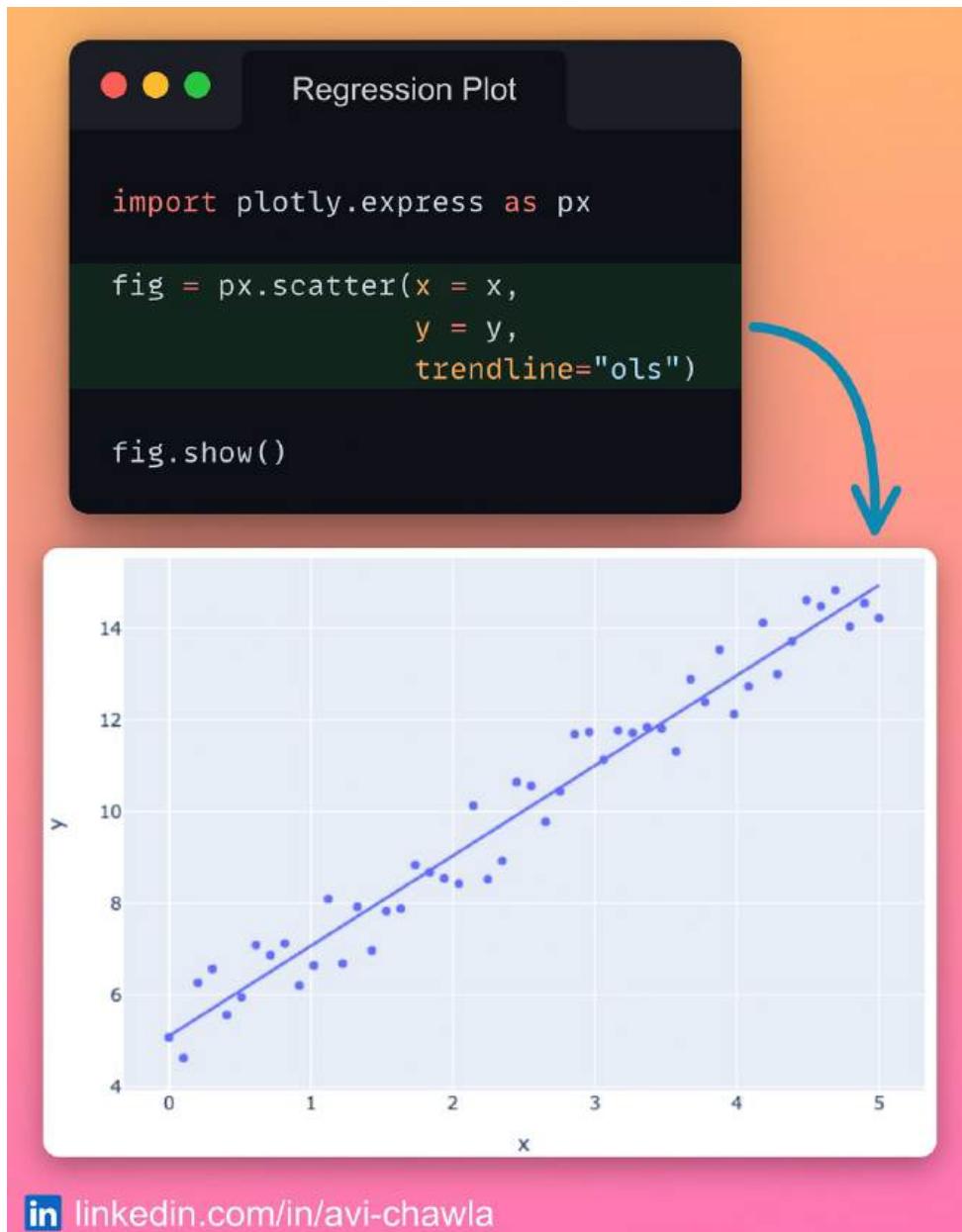
This is typically observed in ML where one often forgets to print all the essential training details/metrics. Executing the entire code again, especially when it has been up for some time is not an ideal approach here.

If you want to modify a function during execution, decorate it with the reloading decorator (**@reloading**). As a result, Python will reload the function from the source before each execution.

Link to reloading: [GitHub](#).



## Regression Plot Made Easy with Plotly



While creating scatter plots, one is often interested in displaying a simple linear regression fit on the data points.

Here, training a model and manually embedding it in the plot can be a tedious job to do.

Instead, with Plotly, you can add a regression line to a plot, without explicitly training a model.

Read more [here](#).



# Polynomial Linear Regression with NumPy

The image shows two Jupyter Notebook cells side-by-side. The top cell is titled 'Sklearn' and the bottom cell is titled 'NumPy'. Both cells contain Python code for polynomial regression.

**Sklearn Cell:**

```
## 1 Degree Polynomial  
model = LinearRegression().fit(x, y)  
  
## 2 Degree Polynomial  
x = np.hstack((x, x**2))  
model = LinearRegression().fit(x, y)  
  
=> x = 2  
=> inp = np.array([[x, x**2]])  
=> model.predict(inp)  
-10.4
```

**NumPy Cell:**

```
coeff = np.polyfit(x, y, deg = 2)  
model = np.poly1d(coeff)  
  
=> inp = 2  
=> model(inp)  
-10.4
```

Annotations with arrows point from specific lines of code to their descriptions:

- An arrow points from the line `x = np.hstack((x, x**2))` to the text "Create Polynomial Features".
- An arrow points from the line `deg = 2` to the text "Specify Degree".

[linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

Polynomial linear regression using Sklearn is tedious as one has to explicitly code its features. This can get challenging when one has to iteratively build higher-degree polynomial models.

NumPy's **polyfit()** method is an excellent alternative to this. Here, you can specify the degree of the polynomial as a parameter. As a result, it automatically creates the corresponding polynomial features.

The downside is that you cannot add custom features such as trigonometric/logarithmic. In other words, you are restricted to only polynomial features. But if that is not your requirement, NumPy's **polyfit()** method can be a better approach.

Read more:

<https://numpy.org/doc/stable/reference/generated/numpy.polyfit.html>.



## Alter the Datatype of Multiple Columns at Once

```
>>> df
   col1  col2  col3 col4
0     1      7      4    A
1     3      9      6    B
2     6      2      5    A
```

```
df["col1"] = df.col1.astype(np.int32)
df["col2"] = df.col2.astype(np.int16)
df["col3"] = df.col3.astype(np.float16)
```

Multiple Calls

Single Call

```
df = df.astype({
    "col1":np.int32,
    "col2":np.int16,
    "col3":np.float16})
```

[linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

A common approach to alter the datatype of multiple columns is to invoke the **astype()** method individually for each column.

Although the approach works as expected, it requires multiple function calls and more code. This can be particularly challenging when you want to modify the datatype of many columns.

As a better approach, you can condense all the conversions into a single function call. This is achieved by passing a dictionary of column-to-datatype mapping, as shown below.



# Datatype For Handling Missing Valued Columns in Pandas

```
>>> len(df.col1)
## Total entries: 1,000,000

>>> len(df[df.col1.isna()])
## NaN entries: 700,000 (70%)
```

```
Sparse Datatype

df.col1.memory_usage()
## Memory usage before conversion: 7.6 MB

df["col1"] = df.col1.astype("Sparse[float32]")

df.col1.memory_usage()
## Memory usage after conversion: 2.0 MB
```

[in](https://www.linkedin.com/in/avi-chawla) [linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

If your data has NaN-valued columns, Pandas provides a datatype specifically for representing them - called the Sparse datatype.

This is especially handy when you are working with large data-driven projects with many missing values.

The snippet compares the memory usage of float and sparse datatype in Pandas.



## Parallelize Pandas with Pandarallel

```
from pandarallel import pandarallel
pandarallel.initialize()

def add_row(row):
    return sum(row)

df = ... ## 10M Rows, 2 Columns
```

```
df.apply(add_row, axis = 1)
## 53 secs

df.parallel_apply(add_row, axis = 1)
## 11 secs
```

[in linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

Pandas' operations do not support parallelization. As a result, it adheres to a single-core computation, even when other cores are available. This makes it inefficient and challenging, especially on large datasets.

"Pandarallel" allows you to parallelize its operations to multiple CPU cores - by changing just one line of code. Supported methods include apply(), applymap(), groupby(), map() and rolling().

Read more: [GitHub](#).



# Why you should not dump DataFrames to a CSV

Save DF

```
1 df = ... ## 1M Rows, 30 Columns
2
3 df.to_csv("file.csv")
4
5 df.to_pickle("file.pickle")
6
7 df.to_parquet("file.parquet")
```

A bar chart comparing the save time (in seconds) for three file formats: CSV, Pickle, and Parquet. The Y-axis represents 'Save Time (in secs)' from 0 to 10. The X-axis represents 'File Format'. The bars show values of 9.6 for CSV, 6.1 for Pickle, and 2.8 for Parquet.

File Format	Save Time (in secs)
CSV	9.6
Pickle	6.1
Parquet	2.8

[linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

The CSV file format is widely used to save Pandas DataFrames. But are you aware of its limitations? To name a few,

1. The CSV does not store the datatype information. Thus, if you modify the datatype of column(s), save it to a CSV, and load again, Pandas will not return the same datatypes.
2. Saving the DataFrame to a CSV file format isn't as optimized as



other supported formats by Pandas. These include Parquet, Pickle, etc.

Of course, if you need to view your data outside Python (Excel, for instance), you are bound to use a CSV. But if not, prefer other file formats.

Further reading: [Why I Stopped Dumping DataFrames to a CSV and Why You Should Too.](#)



# Save Memory with Python Generators

The image shows two side-by-side terminal windows. The left window is titled 'List.py' and the right is titled 'Generator.py'. Both windows run the same code, which generates a large list of integers from 0 to 10^7 and calculates its sum. In 'List.py', line 7 shows a memory usage of 89095160 bytes. In 'Generator.py', line 7 shows a memory usage of 112 bytes. This demonstrates that generators are more memory-efficient than lists for large static iterables.

```
1 from sys import getsizeof
2
3 my_list = [i for i in range(10**7)]
4 ## use [] to create a list
5
6 >>> getsizeof(my_list)
7 ## 89095160 bytes
8
9 >>> sum(my_list)
10 ## 49999995000000
11
12 >>> sum(my_list)
13 ## 49999995000000
```

```
1 from sys import getsizeof
2
3 my_gen = (i for i in range(10**7))
4 ## use () to create a generator
5
6 >>> getsizeof(my_gen)
7 ## 112 bytes
8
9 >>> sum(my_gen)
10 ## 49999995000000
11
12 >>> sum(my_gen)
13 ## 0
```

in [linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

If you use large static iterables in Python, a list may not be an optimal choice, especially in memory-constrained applications.

A list stores the entire collection in memory. However, a generator computes and loads a single element at a time ONLY when it is required. This saves both memory and object creation time.

Of course, there are some limitations of generators too. For instance, you cannot use common list operations such as `append()`, `slicing`, etc.

Moreover, every time you want to reuse an element, it must be regenerated (see `Generator.py`: line 12).



## Don't use print() to debug your code.

```
Print

def func(arr, n):
    print("arr =", arr, "n =", n)

func([1,2,3], 2)
## arr = [1, 2, 3] n = 2
```

```
Icecream

from icecream import ic

def func(arr, n):
    ic(arr, n)

func([1,2,3], 2)
## ic| arr: [1, 2, 3], n: 2
```

[in linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

Debugging with print statements is a messy and inelegant approach. It is confusing to map the output to its corresponding debug statement. Moreover, it requires extra manual formatting to comprehend the output.

The "**icecream**" library in Python is an excellent alternative to this. It makes debugging effortless and readable, with minimal code.



Features include printing expressions, variable names, function names, line numbers, filenames, and many more.

P.S. The snippet only gives a brief demonstration. However, the actual functionalities are much more powerful and elegant as compared to debugging with print().

More about icecream

here: <https://github.com/gruns/icecream>.



## Find Unused Python Code With Ease

```
code.py
```

```
1 def sum_func(arr):
2     return sum(arr)
3
4 def max_func(arr):
5     return max(arr)
6
7 if __name__ == "__main__":
8
9     input_arr = [1, 3, 5, 2, 9]
10    flag = 1
11
12    input_sum = sum_func(input_arr)
13    print(input_sum)
```

```
Terminal
```

```
$ vulture code.py
code.py:4: unused function 'max_func'
code.py:10: unused variable 'flag'
```

[linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

As the size of your codebase increases, so can the number of instances of unused code. This inhibits its readability and conciseness.

With the "vulture" module in Python, you can locate dead (unused) code in your pipeline, as shown in the snippet.



# Define the Correct DataType for Categorical Columns

The image shows two Jupyter Notebook cells side-by-side. The left cell, titled 'Categorical Col.', contains the following code:

```
import pandas as pd  
  
len(df.Gender)  
## 1500  
  
df.Gender.unique()  
## ["Male", "Female"]
```

The right cell, titled 'Reduce Memory Usage', contains the following code:

```
import pandas as pd  
  
df.Gender.memory_usage(), df.Gender.dtype  
## 90.5 KB, object  
  
df["Gender"] = df.Gender.astype("category")  
  
df.Gender.memory_usage(), df.Gender.dtype  
## 1.8 KB, CategoricalDtype
```

Below the notebook screenshot is a LinkedIn profile link: [linkedin.com/in/avi-chawla](https://linkedin.com/in/avi-chawla).

If your data has categorical columns, you should not represent them as int/string data type.

Rather, Pandas provides an optimized data type specifically for categorical columns. This is especially handy when you are working with large data-driven projects.

The snippet compares the memory usage of string and categorical data types in Pandas.



# Transfer Variables Between Jupyter Notebooks

```
value = 10
%store value
```

```
%store -r value
print(value)
## 10
```

[in linkedin.com/in/avi-chawla](https://linkedin.com/in/avi-chawla)

While working with multiple jupyter notebooks, you may need to share objects between them.

With the "store" magic command, you can transfer variables across notebooks without storing them on disk.

P.S. You can also restart the kernel and retrieve an old variable with "store".



# Why You Should Not Read CSVs with Pandas

The screenshot shows two terminal windows side-by-side. The left window is titled 'Pandas' and the right window is titled 'Datatable'. Both windows show Python code snippets for reading a CSV file named 'file.csv'.

**Pandas:**

```
1 file = "file.csv"
2 ## 1M rows and 30 columns
3
4 import pandas as pd
5
6 df = pd.read_csv(file)
7 ## 8.82 secs
```

**Datatable:**

```
1 file = "file.csv"
2
3 import datatable as dt
4
5 df = dt.fread(file)
6 df = df.to_pandas()
7 ## 4.04 secs (line 5 + 6)
```

Pandas adheres to a single-core computation, which makes its operations extremely inefficient, especially on large datasets.

The "datatable" library in Python is an excellent alternative with a Pandas-like API. Its multi-threaded data processing support makes it faster than Pandas.

The snippet demonstrates the run-time comparison of creating a "Pandas DataFrame" from a CSV using Pandas and Datatable.



# Modify Python Code During Run-Time

The image shows two screenshots of a Mac terminal window. The top screenshot displays a script that prints odd numbers from 1 to 11. The bottom screenshot shows the same script, but the last two lines have been modified to also print even numbers (14, 16, 18) as 'Even'. A red arrow points from the top terminal to the bottom one, labeled 'Modified During Run-time'.

```
from time import sleep
from reloading import reloading

for number in reloading(range(100)):

    if number % 2:
        print(f"{number} is Odd")
    else:
        pass
```

```
1 is Odd
3 is Odd
5 is Odd
7 is Odd
9 is Odd
11 is Odd
```

```
from time import sleep
from reloading import reloading

for number in reloading(range(100)):

    if number % 2:
        print(f"{number} is Odd")
    else:
        print(f"{number} is Even")
```

```
1 is Odd
3 is Odd
5 is Odd
7 is Odd
9 is Odd
11 is Odd
13 is Odd
14 is Even
15 is Odd
16 is Even
17 is Odd
18 is Even
19 is Odd
```

[in linkedin.com/in/avi-chawla](https://www.linkedin.com/in/avi-chawla)

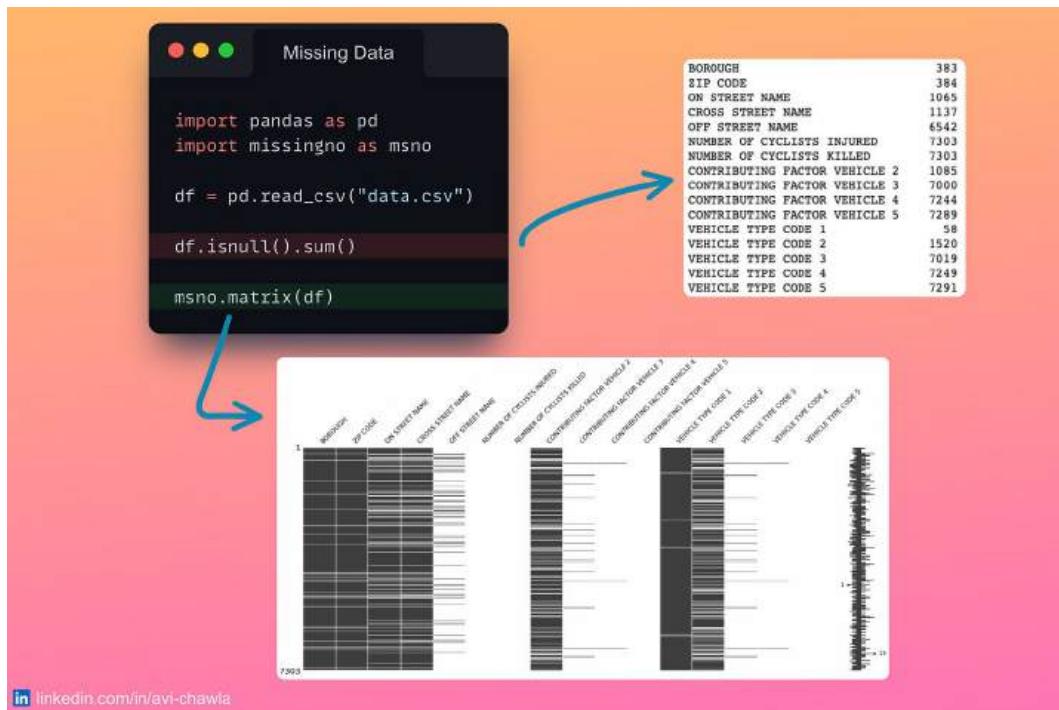
Have you ever been in a situation where you wished to add more details to an already running code (printing more details in a for-loop, for instance)?

Executing the entire code again, especially when it has been up for some time, is not the ideal approach here.

With the "reloading" library in Python, you can add more details to a running code without losing any existing progress.



# Handle Missing Data With Missingno



If you want to analyze missing values in your dataset, Pandas may not be an apt choice.

Pandas' methods hide many important details about missing values. These include their location, periodicity, the correlation across columns, etc.

The "missingno" library in Python is an excellent resource for exploring missing data. It generates informative visualizations for improved data analysis.

The snippet demonstrates missing data analysis using Pandas and Missingno.