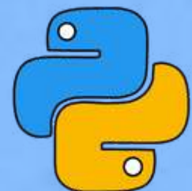
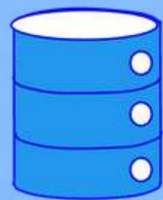
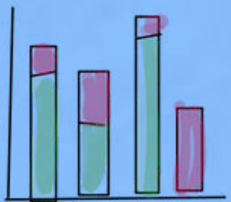
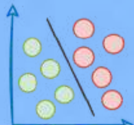
DATA SCIENCE

FULL ARCHIVE

320+ Data Science posts

580+ pages


Daily Dose of Data Science

blog.DailyDoseofDS.com

Table of Contents

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

The Limitation of Static Embeddings Which Made Them Obsolete 109

An Overlooked Technique To Improve KMeans Run-time 116

The Most Underrated Skill in Training Linear Models 119

Poisson Regression: The Robust Extension of Linear Regression 125

The Biggest Mistake ML Folks Make When Using Multiple Embedding Models 126

Probability and Likelihood Are Not Meant To Be Used Interchangeably 129

SummaryTools: A Richer Alternative To Pandas' Describe Method. 135

40 NumPy Methods That Data Scientists Use 95% of the Time 136

An Overly Simplified Guide To Understanding How Neural Networks Handle Linearly Inseparable Data...138

2 Mathematical Proofs of Ordinary Least Squares 145

A Common Misconception About Log Transformation 146

Raincloud Plots: The Hidden Gem of Data Visualisation................................149

7 Must-know Techniques For Encoding Categorical Feature 153

Automated EDA Tools That Let You Avoid Manual EDA Tasks 154

The Limitation Of Silhouette Score Which Is Often Ignored By Many 156

9 Must-Know Methods To Test Data Normality 159

A Visual Guide to Popular Cross Validation Techniques 163

Decision Trees ALWAYS Overfit. Here's A Lesser-Known Technique To Prevent It. 167

Evaluate Clustering Performance Without Ground Truth Labels 169

One-Minute Guide To Becoming a Polars-savvy Data Scientist.......................172

The Most Common Misconception About Continuous Probability Distributions 174

Don't Overuse Scatter, Line and Bar Plots. Try These Four Elegant Alternatives. 175

CNN Explainer: Interactively Visualize a Convolutional Neural Network.........178

Sankey Diagrams: An Underrated Gem of Data Visualization 180

A Common Misconception About Feature Scaling and Standardization..........181

7 Elegant Usages of Underscore in Python...184

Random Forest May Not Need An Explicit Validation Set For Evaluation 185

Declutter Your Jupyter Notebook Using Interactive Controls 188

Avoid Using Pandas' Apply() Method At All Times 190

A Visual and Overly Simplified Guide To Bagging and Boosting 192

10 Most Common (and Must-Know) Loss Functions in ML............................195

How To Enforce Type Hints in Python?..196

A Common Misconception About Deleting Objects in Python........................197

Theil-Sen Regression: The Robust Twin of Linear Regression.........................200

What Makes The Join() Method Blazingly Faster Than Iteration?..................202

A Major Limitation of NumPy Which Most Users Aren't Aware Of................205

The Limitations Of Elbow Curve And What You Should Replace It With.........206

21 Most Important (and Must-know) Mathematical Equations in Data Science 210

Beware of This Unexpected Behaviour of NumPy Methods...........................213

Try This If Your Linear Regression Model is Underperforming........................214

Pandas vs Polars — Run-time and Memory Comparison..............................216

A Hidden Feature of a Popular String Method in Python..............................218

The Limitation of KMeans Which Is Often Overlooked by Many....................219

🚀 Jupyter Notebook + Spreadsheet + AI — All in One Place With Mito..........221

Nine Most Important Distributions in Data Science......................................223

The Limitation of Linear Regression Which is Often Overlooked By Many 229

A Reliable and Efficient Technique To Measure Feature Importance..............231

Does Every ML Algorithm Rely on Gradient Descent?...................................233

Why Sklearn's Linear Regression Has No Hyperparameters?.........................235

Enrich The Default Preview of Pandas DataFrame with Jupyter DataTables...237

Visualize The Performance Of Linear Regression With This Simple Plot.........238

Enrich Your Heatmaps With This Simple Trick 240

Confidence Interval and Prediction Interval Are Not The Same 241

The Ultimate Categorization of Performance Metrics in ML..........................243

The Coolest Matplotlib Hack to Create Subplots Intuitively...........................247

Execute Python Project Directory as a Script..249

The Most Overlooked Problem With One-Hot Encoding................................250

9 Most Important Plots in Data Science 252

Is Categorical Feature Encoding Always Necessary Before Training ML Models? 254

Scikit-LLM: Integrate Sklearn API with Large Language Models....................257

The Counterintuitive Behaviour of Training Accuracy and Training Loss 258

A Highly Overlooked Point In The Implementation of Sigmoid Function 262

The Ultimate Categorization of Clustering Algorithms 265

Improve Python Run-time Without Changing A Single Line of Code 267

A Lesser-Known Feature of the Merge Method in Pandas 269

The Coolest GitHub-Colab Integration You Would Ever See 271

Most Sklearn Users Don't Know This About Its LinearRegression Implementation 272

Break the Linear Presentation of Notebooks With Stickyland.........................274

Visualize The Performance Of Any Linear Regression Model With This Simple Plot 275

Waterfall Charts: A Better Alternative to Line/Bar Plot 277

What Does The Google Styling Guide Say About Imports 278

How To Truly Use The Train, Validation and Test Set 280

Restart Jupyter Kernel Without Losing Variables...283

The Advantages and Disadvantages of PCA To Consider Before Using It 284

Loss Functions: An Algorithm-wise Comprehensive Summary 286

Is Data Normalization Always Necessary Before Training ML Models? 288

Annotate Data With The Click Of A Button Using Pigeon 291

Enrich Your Confusion Matrix With A Sankey Diagram.................................292

A Visual Guide to Stochastic, Mini-batch, and Batch Gradient Descent 294

A Lesser-Known Difference Between For-Loops and List Comprehensions 297

The Limitation of PCA Which Many Folks Often Ignore 299

Magic Methods: An Underrated Gem of Python OOP 302

The Taxonomy Of Regression Algorithms That Many Don't Bother To Remember 305

A Highly Overlooked Approach To Analysing Pandas DataFrames 307

Visualise The Change In Rank Over Time With Bump Charts 308

Use This Simple Technique To Never Struggle With TP, TN, FP and FN Again ..309

The Most Common Misconception About Inplace Operations in Pandas 311

Build Elegant Web Apps Right From Jupyter Notebook with Mercury 313

Become A Bilingual Data Scientist With These Pandas to SQL Translations 315

A Lesser-Known Feature of Sklearn To Train Models on Large Datasets 317

A Simple One-Liner to Create Professional Looking Matplotlib Plots 319

Avoid This Costly Mistake When Indexing A DataFrame................................321

9 Command Line Flags To Run Python Scripts More Flexibly..........................324

Breathing KMeans: A Better and Faster Alternative to KMeans 326

How Many Dimensions Should You Reduce Your Data To When Using PCA?...329

🚀 Mito Just Got Supercharged With AI!...332

Be Cautious Before Drawing Any Conclusions Using Summary Statistics 334

Use Custom Python Objects In A Boolean Context..336

A Visual Guide To Sampling Techniques in Machine Learning.........................338

You Were Probably Given Incomplete Info About A Tuple's Immutability 342

A Simple Trick That Significantly Improves The Quality of Matplotlib Plots 344

A Visual and Overly Simplified Guide to PCA..346

Supercharge Your Jupyter Kernel With ipyflow 349

A Lesser-known Feature of Creating Plots with Plotly 351

The Limitation Of Euclidean Distance Which Many Often Ignore...................353

Visualising The Impact Of Regularisation Parameter 356

AutoProfiler: Automatically Profile Your DataFrame As You Work.................358

A Little Bit Of Extra Effort Can Hugely Transform Your Storytelling Skills 360

A Nasty Hidden Feature of Python That Many Programmers Aren't Aware Of 362

Interactively Visualise A Decision Tree With A Sankey Diagram 365

Use Histograms With Caution. They Are Highly Misleading! 367

Three Simple Ways To (Instantly) Make Your Scatter Plots Clutter Free 369

A (Highly) Important Point to Consider Before You Use KMeans Next Time 372

Why You Should Avoid Appending Rows To A DataFrame 375

Matplotlib Has Numerous Hidden Gems. Here's One of Them.......................377

A Counterintuitive Thing About Python Dictionaries 379

Probably The Fastest Way To Execute Your Python Code 382

Are You Sure You Are Using The Correct Pandas Terminologies? 384

Is Class Imbalance Always A Big Problem To Deal With?...............................387

A Simple Trick That Will Make Heatmaps More Elegant 389

A Visual Comparison Between Locality and Density-based Clustering 391

Why Don't We Call It Logistic Classification Instead? 392

A Typical Thing About Decision Trees Which Many Often Ignore...................394

Always Validate Your Output Variable Before Using Linear Regression 395

A Counterintuitive Fact About Python Functions 396

Why Is It Important To Shuffle Your Dataset Before Training An ML Model 397

The Limitations Of Heatmap That Are Slowing Down Your Data Analysis.......398

The Limitation Of Pearson Correlation Which Many Often Ignore 399

Why Are We Typically Advised To Set Seeds for Random Generators?...........400

An Underrated Technique To Improve Your Data Visualizations 401

A No-Code Tool to Create Charts and Pivot Tables in Jupyter..........................402

If You Are Not Able To Code A Vectorized Approach, Try This. 403

Why Are We Typically Advised To Never Iterate Over A DataFrame?.............405

Manipulating Mutable Objects In Python Can Get Confusing At Times 406

This Small Tweak Can Significantly Boost The Run-time of KMeans 408

Most Python Programmers Don't Know This About Python OOP 410

Who Said Matplotlib Cannot Create Interactive Plots? 412

Don't Create Messy Bar Plots. Instead, Try Bubble Charts!............................413

You Can Add a List As a Dictionary's Key (Technically)!..................................414

Most ML Folks Often Neglect This While Using Linear Regression 415

35 Hidden Python Libraries That Are Absolute Gems 416

Use Box Plots With Caution! They May Be Misleading. 417

An Underrated Technique To Create Better Data Plots 418

The Pandas DataFrame Extension Every Data Scientist Has Been Waiting For 419

Supercharge Shell With Python Using Xonsh 420

Most Command-line Users Don't Know This Cool Trick About Using Terminals 421

A Simple Trick to Make The Most Out of Pivot Tables in Pandas.....................422

Why Python Does Not Offer True OOP Encapsulation...................................423

Never Worry About Parsing Errors Again While Reading CSV with Pandas.....424

An Interesting and Lesser-Known Way To Create Plots Using Pandas 425

Most Python Programmers Don't Know This About Python For-loops 426

How To Enable Function Overloading In Python..427

Generate Helpful Hints As You Write Your Pandas Code 428

Speedup NumPy Methods 25x With Bottleneck 429

Visualizing The Data Transformation of a Neural Network 430

Never Refactor Your Code Manually Again. Instead, Use Sourcery!................431

Draw The Data You Are Looking For In Seconds 432

Style Matplotlib Plots To Make Them More Attractive 433

Speed-up Parquet I/O of Pandas by 5x..434

40 Open-Source Tools to Supercharge Your Pandas Workflow........................435

Stop Using The Describe Method in Pandas. Instead, use Skimpy...................436

The Right Way to Roll Out Library Updates in Python 437

Simple One-Liners to Preview a Decision Tree Using Sklearn..........................438

Stop Using The Describe Method in Pandas. Instead, use Summarytools. 439

Never Search Jupyter Notebooks Manually Again To Find Your Code 440

F-strings Are Much More Versatile Than You Think 441

Is This The Best Animated Guide To KMeans Ever?.......................................442

An Effective Yet Underrated Technique To Improve Model Performance.........443

Create Data Plots Right From The Terminal 444

Make Your Matplotlib Plots More Professional...445

37 Hidden Python Libraries That Are Absolute Gems 446

Preview Your README File Locally In GitHub Style..447

Pandas and NumPy Return Different Values for Standard Deviation. Why? ...448

Visualize Commit History of Git Repo With Beautiful Animations..................449

Perfplot: Measure, Visualize and Compare Run-time With Ease.....................450

This GUI Tool Can Possibly Save You Hours Of Manual Work..........................451

How Would You Identify Fuzzy Duplicates In A Data With Million Records?....452

Stop Previewing Raw DataFrames. Instead, Use DataTables..........................454

🚀 *A Single Line That Will Make Your Python Code Faster*..............................455

Prettify Word Clouds In Python...456

How to Encode Categorical Features With Many Categories? 457

Calendar Map As A Richer Alternative to Line Plot..458

10 Automated EDA Tools That Will Save You Hours Of (Tedious) Work...........459

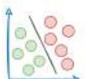
Why KMeans May Not Be The Apt Clustering Algorithm Always....................460

Converting Python To LaTeX Has Possibly Never Been So Simple....................461

Density Plot As A Richer Alternative to Scatter Plot 462

30 Python Libraries to (Hugely) Boost Your Data Science Productivity...........463

Sklearn One-liner to Generate Synthetic Data..464



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

Debugging Made Easy With PySnooper..498

Lesser-Known Feature of the Merge Method in Pandas................................499

The Best Way to Use Apply() in Pandas..500

Deep Learning Network Debugging Made Easy...501

Don't Print NumPy Arrays! Use Lovely-NumPy Instead.................................502

Performance Comparison of Python 3.11 and Python 3.10...........................503

View Documentation in Jupyter Notebook..504

A No-code Tool To Understand Your Data Quickly..505

Why 256 is 256 But 257 is not 257?...506

Make a Class Object Behave Like a Function..508

Lesser-known feature of Pickle Files..510

Dot Plot: A Potential Alternative to Bar Plot..512

Why Correlation (and Other Statistics) Can Be Misleading............................513

Supercharge value_counts() Method in Pandas With Sidetable....................514

Write Your Own Flavor Of Pandas..515

CodeSquire: The AI Coding Assistant You Should Use Over GitHub Copilot.....516

Vectorization Does Not Always Guarantee Better Performance....................517

In Defense of Match-case Statements in Python..518

Enrich Your Notebook With Interactive Controls..520

Get Notified When Jupyter Cell Has Executed..522

Data Analysis Using No-Code Pandas In Jupyter...523

Using Dictionaries In Place of If-conditions..524

Clear Cell Output In Jupyter Notebook During Run-time...............................526

A Hidden Feature of Describe Method In Pandas..527

Use Slotted Class To Improve Your Python Code...528

Stop Analysing Raw Tables. Use Styling Instead!..529

Explore CSV Data Right From The Terminal..530

Generate Your Own Fake Data In Seconds...531

Import Your Python Package as a Module...532

Specify Loops and Runs In %%timeit..533

Waterfall Charts: A Better Alternative to Line/Bar Plot................................534

Hexbin Plots As A Richer Alternative to Scatter Plots...................................535

Importing Modules Made Easy with Pyforest..536

Analyse Flow Data With Sankey Diagrams 538

Feature Tracking Made Simple In Sklearn Transformers.................................540

Lesser-known Feature of f-strings in Python 542

Don't Use time.time() To Measure Execution Time 543

Now You Can Use DALL·E With OpenAI API 544

Polynomial Linear Regression Plot Made Easy With Seaborn 545

Retrieve Previously Computed Output In Jupyter Notebook 546

Parallelize Pandas Apply() With Swifter 547

Create DataFrame Hassle-free By Using Clipboard..548

Run Python Project Directory As A Script 549

Inspect Program Flow with IceCream 550

Don't Create Conditional Columns in Pandas with Apply...............................551

Pretty Plotting With Pandas 552

Build Baseline Models Effortlessly With Sklearn...553

Fine-grained Error Tracking With Python 3.11 554

Find Your Code Hiding In Some Jupyter Notebook With Ease..........................555

Restart the Kernel Without Losing Variables...556

How to Read Multiple CSV Files Efficiently 557

Elegantly Plot the Decision Boundary of a Classifier......................................559

An Elegant Way to Import Metrics From Sklearn 560

Configure Sklearn To Output Pandas DataFrame 561

Display Progress Bar With Apply() in Pandas 562

Modify a Function During Run-time...563

Regression Plot Made Easy with Plotly 564

Polynomial Linear Regression with NumPy...565

Alter the Datatype of Multiple Columns at Once...566

Datatype For Handling Missing Valued Columns in Pandas...........................567

Parallelize Pandas with Pandarallel...568

Why you should not dump DataFrames to a CSV 569

Save Memory with Python Generators 571

Don't use print() to debug your code. 572

Find Unused Python Code With Ease 574

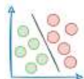
Define the Correct DataType for Categorical Columns..................................575

Transfer Variables Between Jupyter Notebooks *576*

Why You Should Not Read CSVs with Pandas *577*

Modify Python Code During Run-Time..*578*

Handle Missing Data With Missingno...*579*



The Must-Know Categorisation of Discriminative Models

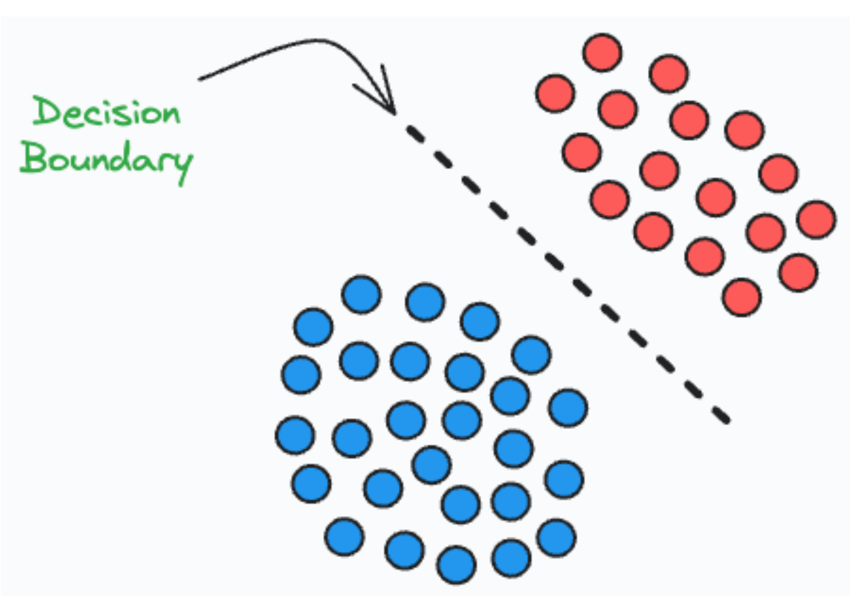
In one of the earlier posts, we discussed **Generative and Discriminative Models**.

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

Let's understand.

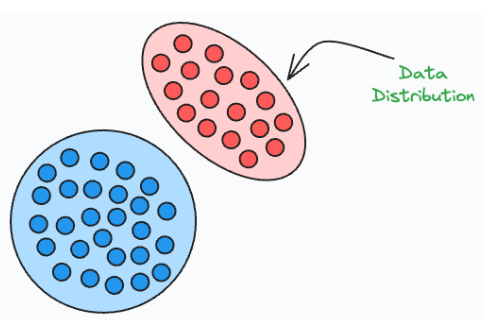
To recap:

Discriminative models:



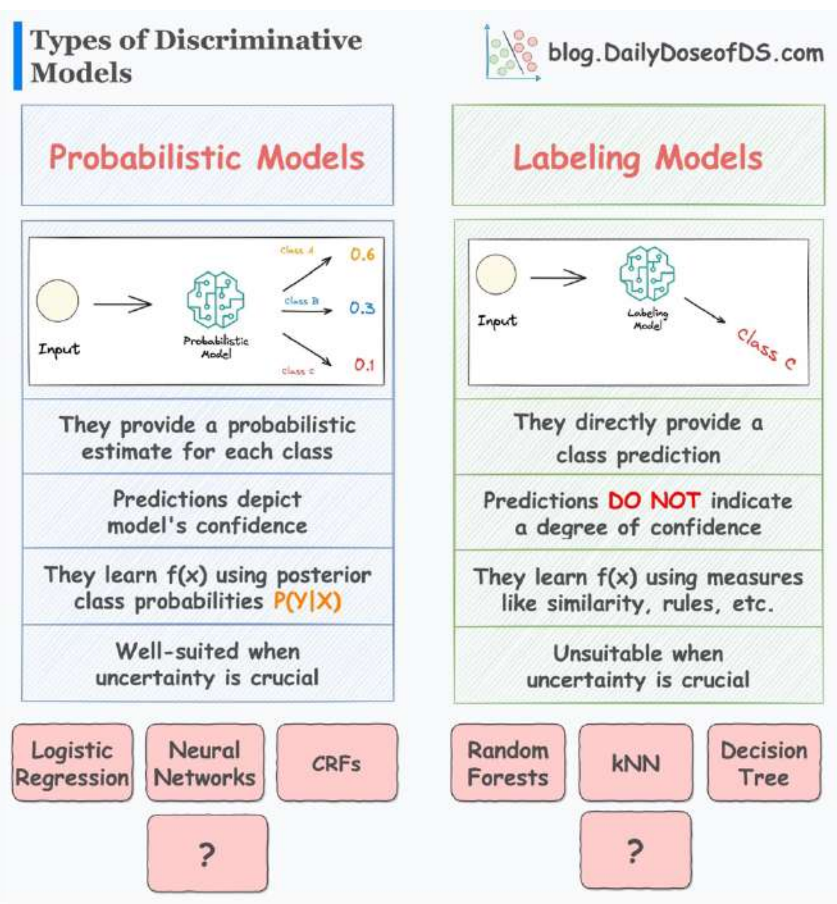
- learn decision boundaries that separate different classes.
- maximize the conditional probability: P(Y|X) — Given an input X, maximize the probability of label Y.
- are meant explicitly for classification tasks.

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.

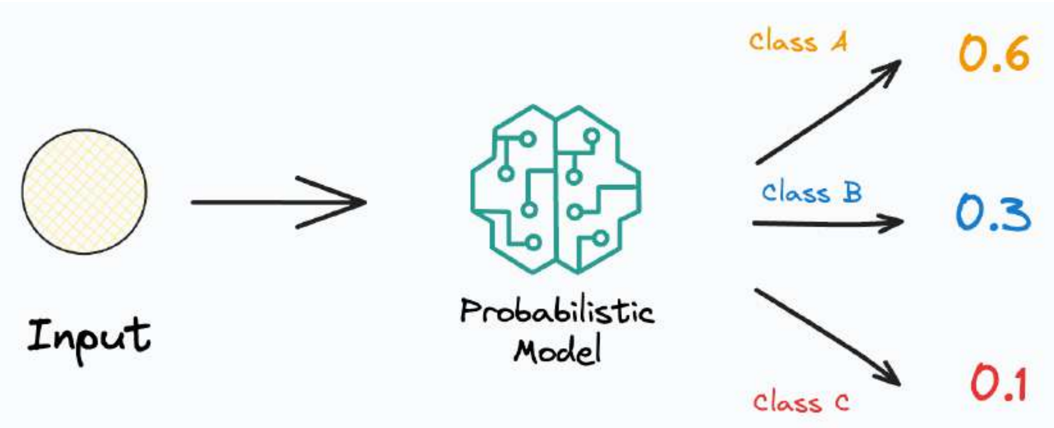
In a gist, discriminative models directly learn the function *f* that maps an input vector (x) to a label (y).



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).

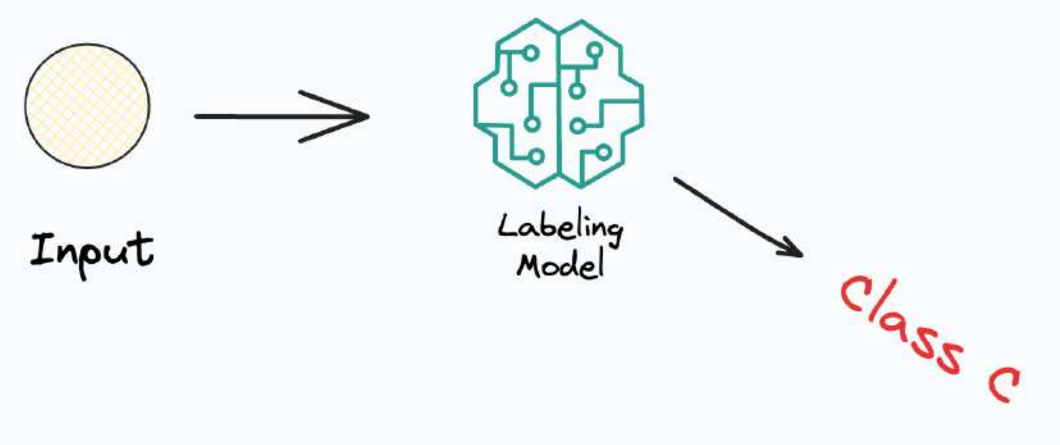
As a result, their predictions depict the model's confidence in predicting a specific class label.

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.

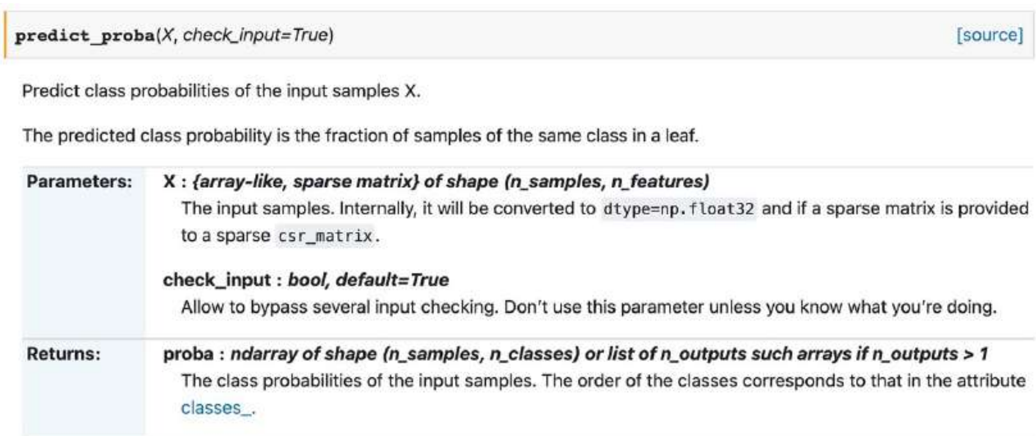
This makes them unsuitable when uncertainty in a model's prediction is crucial.

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]

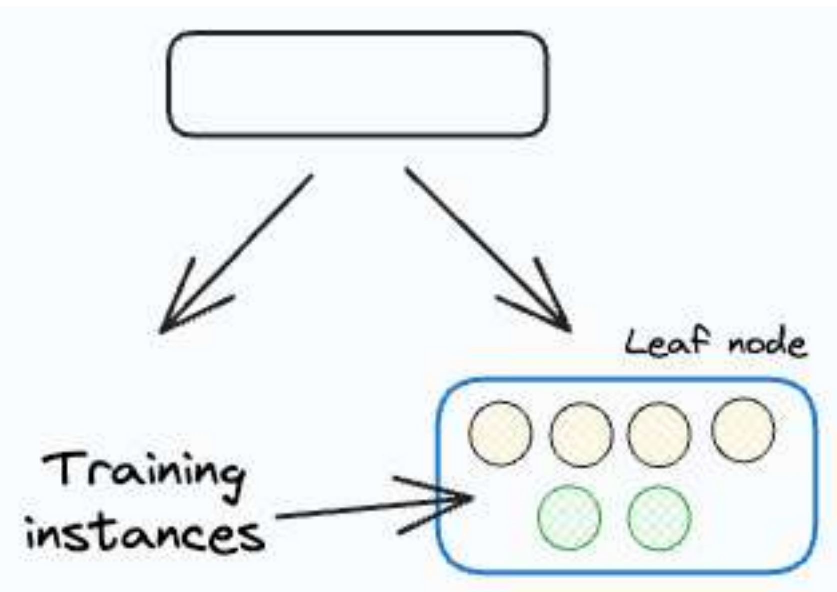
Predict class probabilities of the input samples X.

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

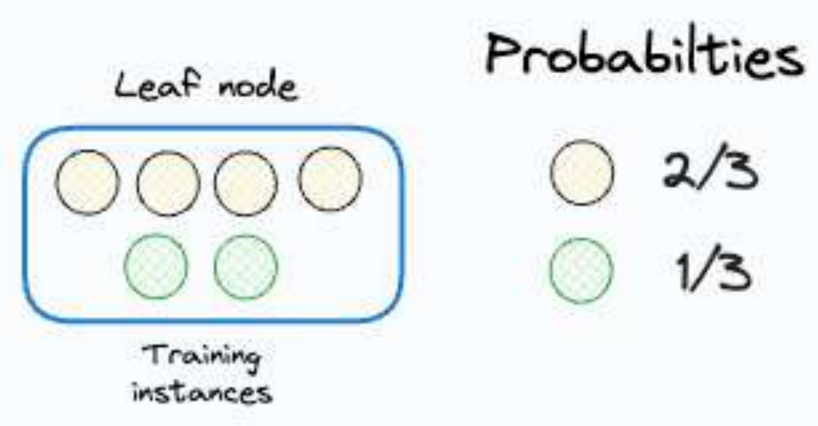
Parameters:	**X : {array-like, sparse matrix} of shape (n_samples, n_features)** The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`. **check_input : bool, default=True** Allow to bypass several input checking. Don't use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

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.

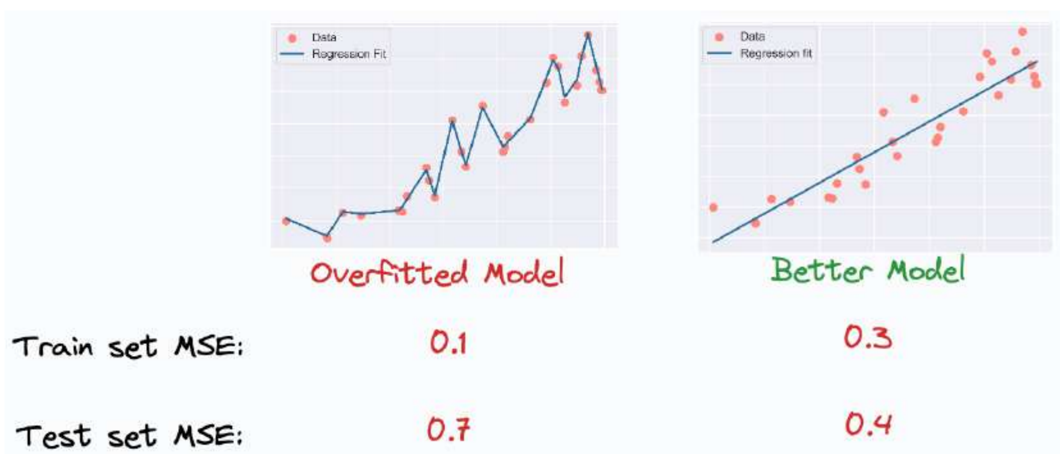
👉 Over to you: Can you add one more model for probabilistic and labeling models?

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.

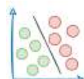
This may happen because the model is trying too hard to capture all **unrelated and random noise** in our training dataset, as shown below:

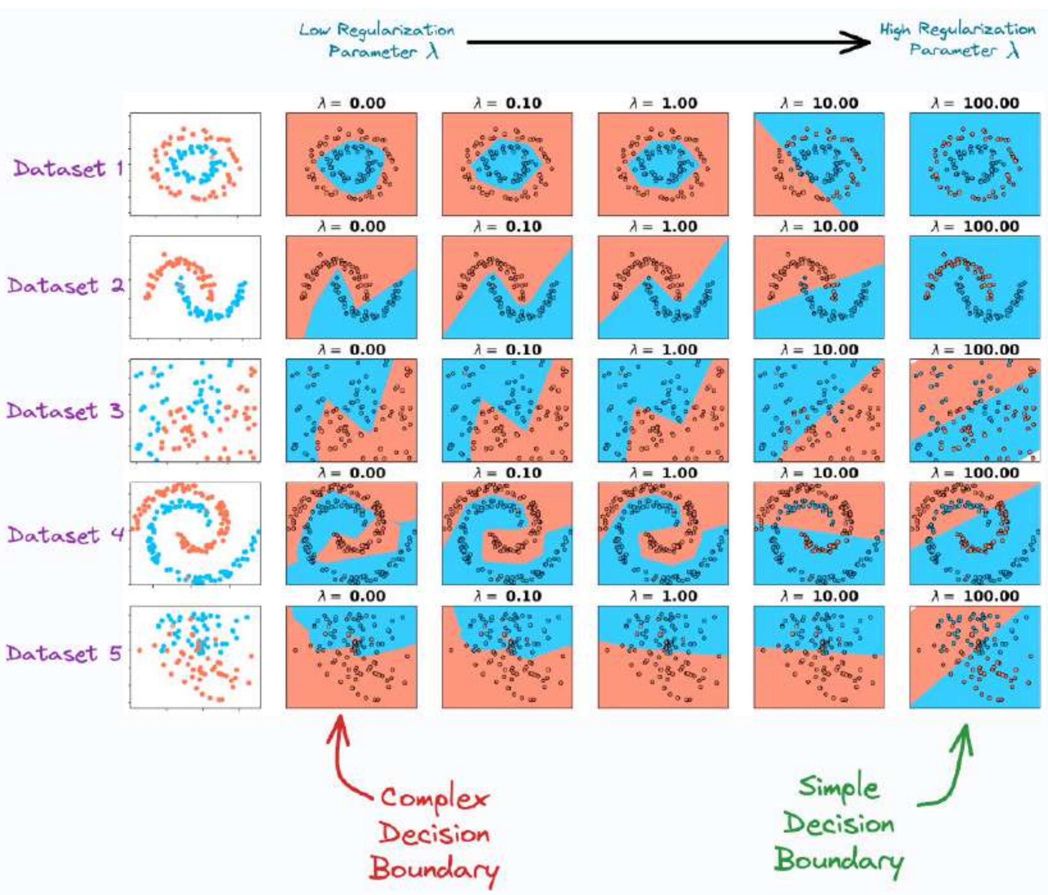


And one of the most common techniques to avoid overfitting is **regularization**.

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

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

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**."

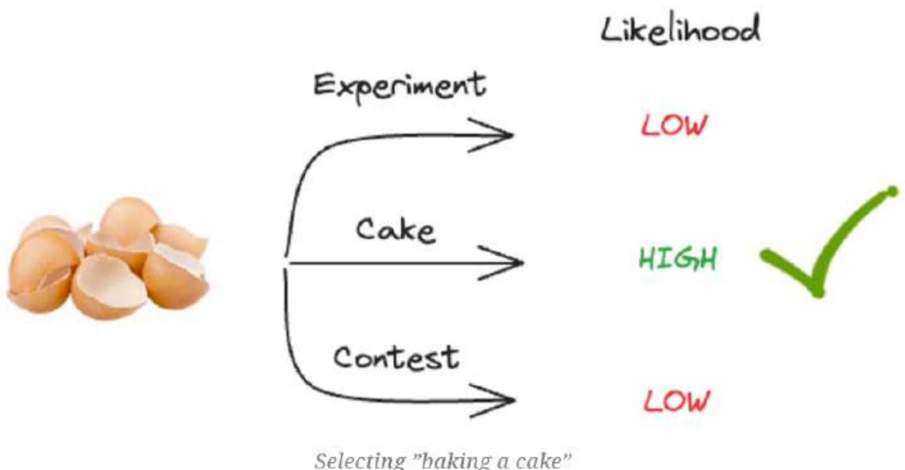
The Probabilistic Origin of Regularization

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.



Selecting "baking a cake"

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).

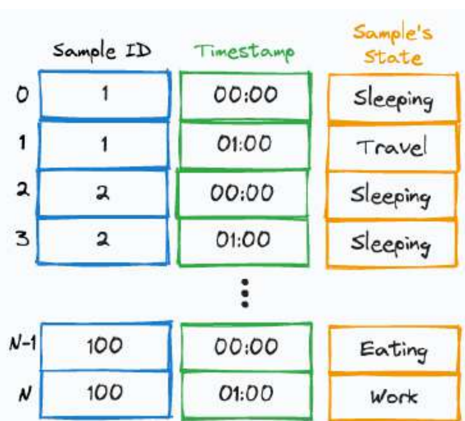
I always wondered how one can create them in Python.

Turns out, there's a pretty simple way to do it just three lines of Python using **D3Blocks**.

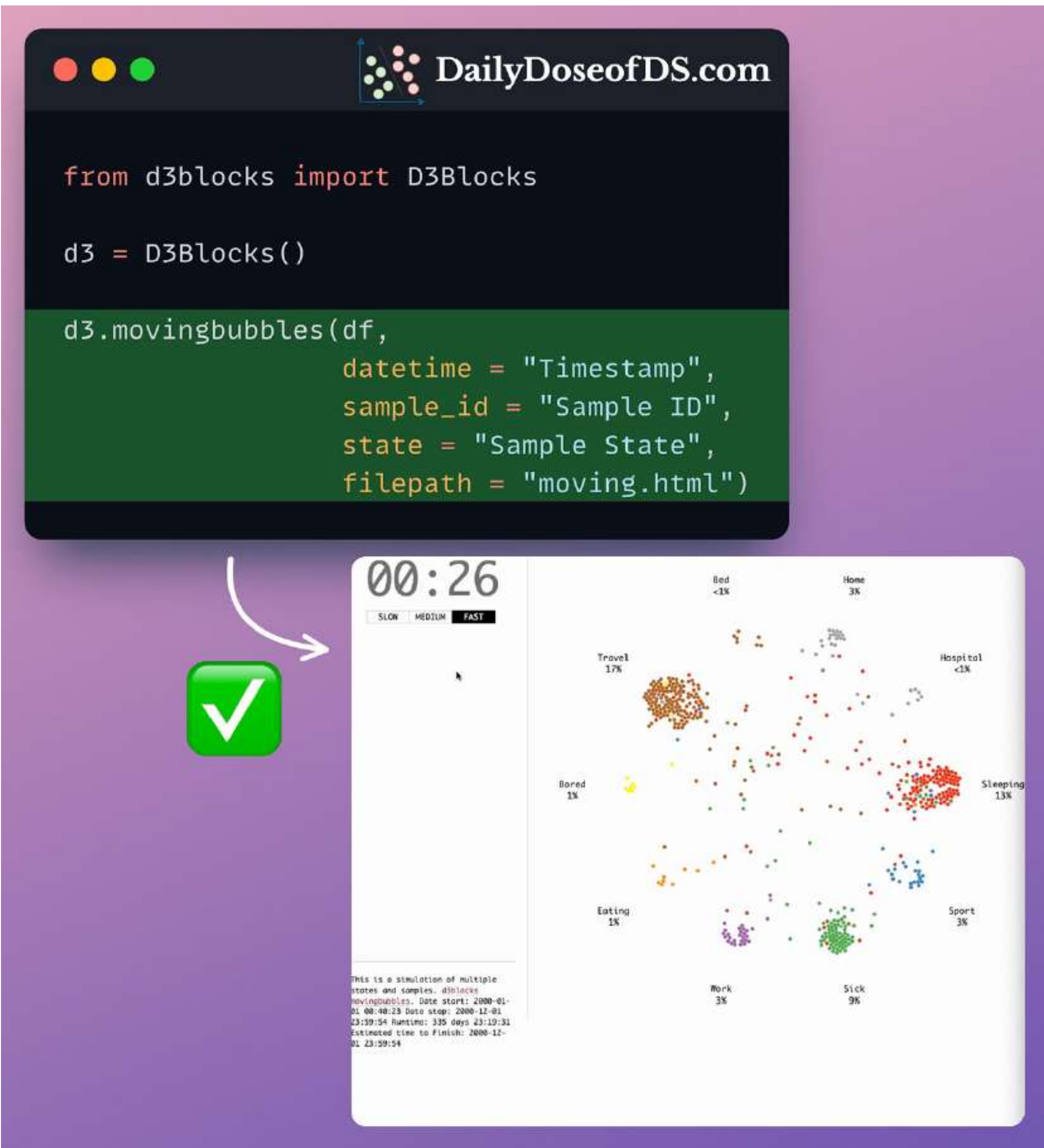
The library utilizes the graphics of the popular **d3js Javascript library** to create visually appealing charts with only a few lines of Python code.

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:



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



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.

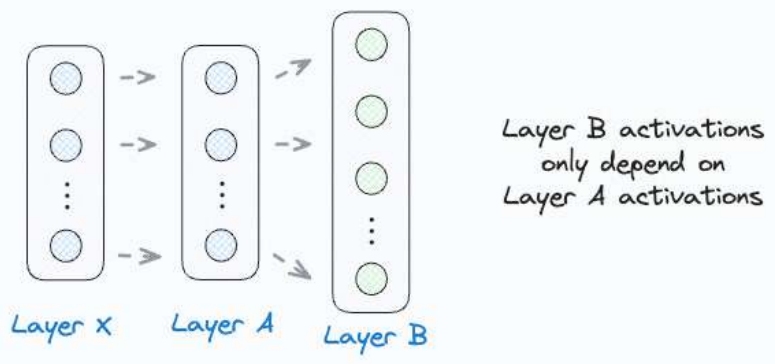
Gradient checkpointing is an incredible technique to reduce the memory overheads of neural nets.

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

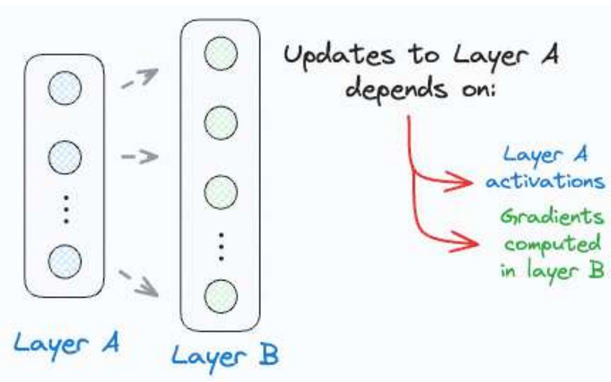
Let's understand how it works.

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:

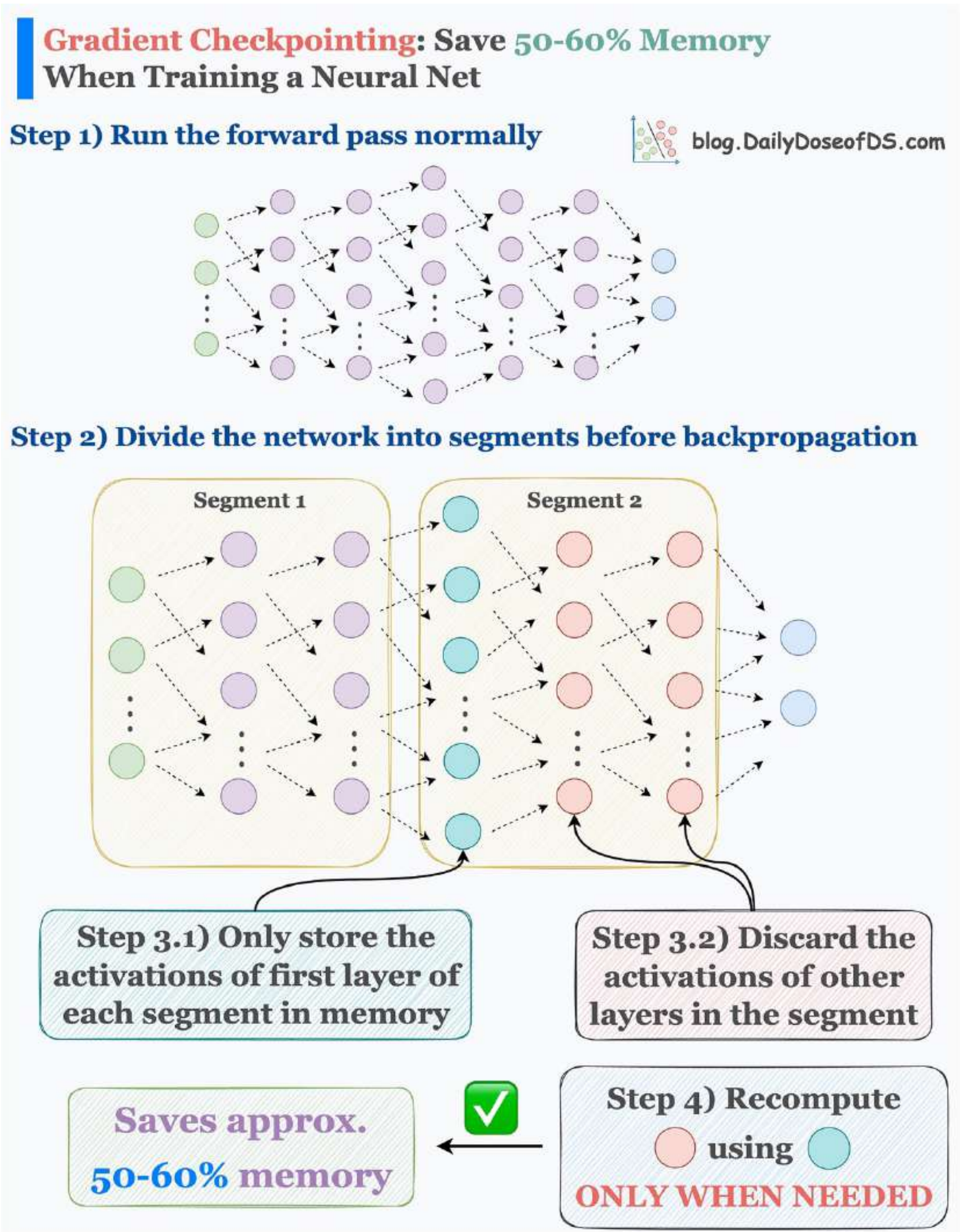


  - The activations of that layer.
  - The gradients computed in the next (right) layer.

Gradient checkpointing exploits these ideas to optimize backpropagation:

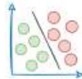
- Divide the network into segments before backpropagation
- In each segment:
  - Only store the activations of the first layer.
  - Discard the rest of the activations.
- When updating the weights of layers in a segment, recompute its activations using the first layer in that segment.

This is depicted in the image below:



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.

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?

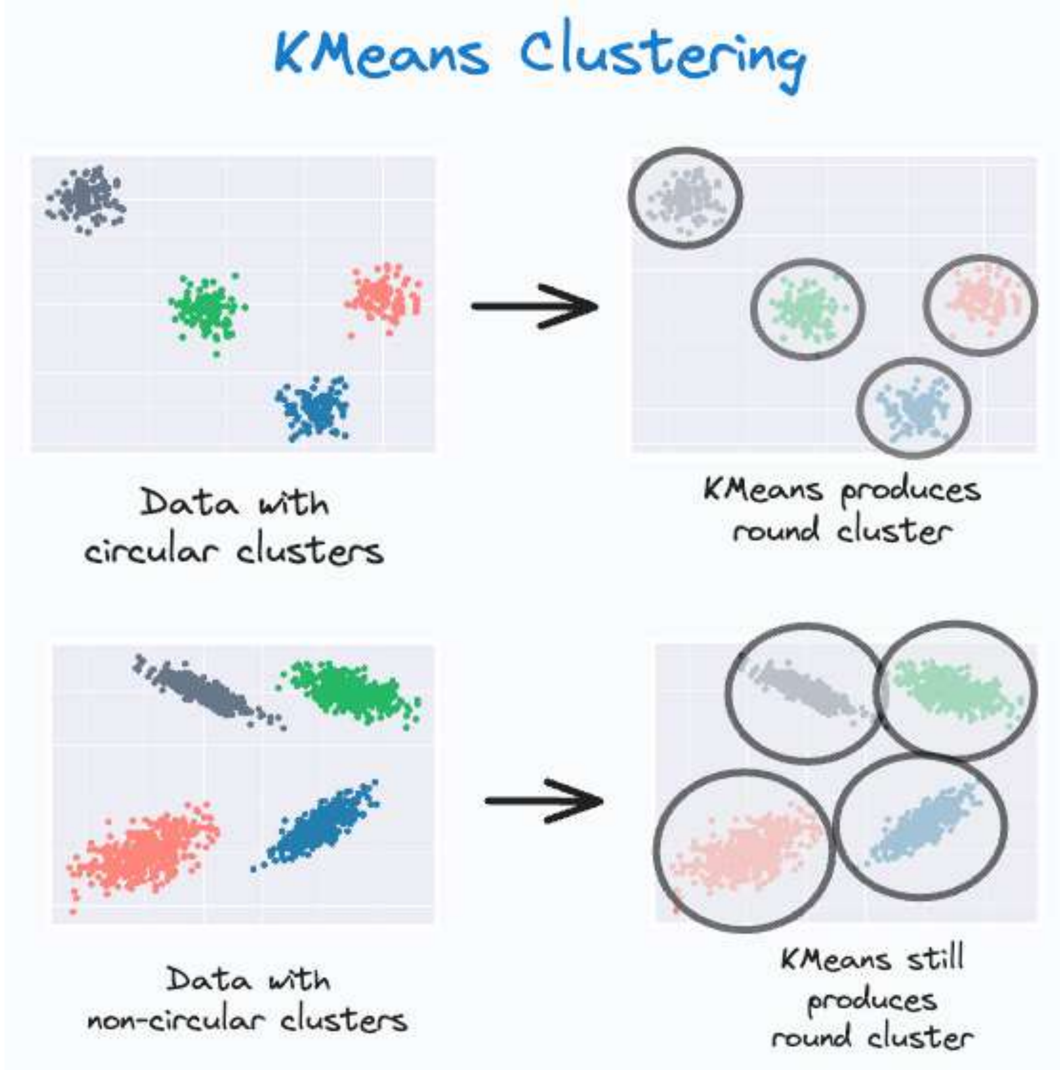
Gaussian Mixture Models: The Flexible Twin of KMeans

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

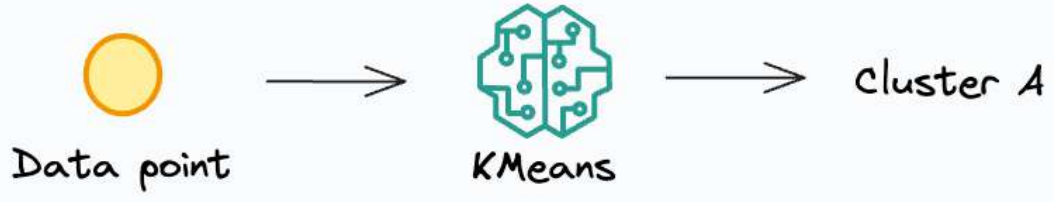
But it has many limitations.

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.

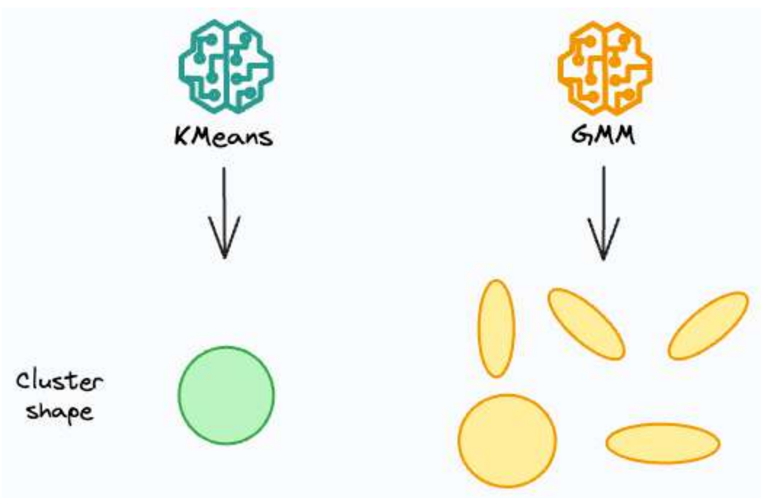
As the name suggests, they can cluster a dataset that has a mixture of many Gaussian distributions.

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

The primary difference is that:

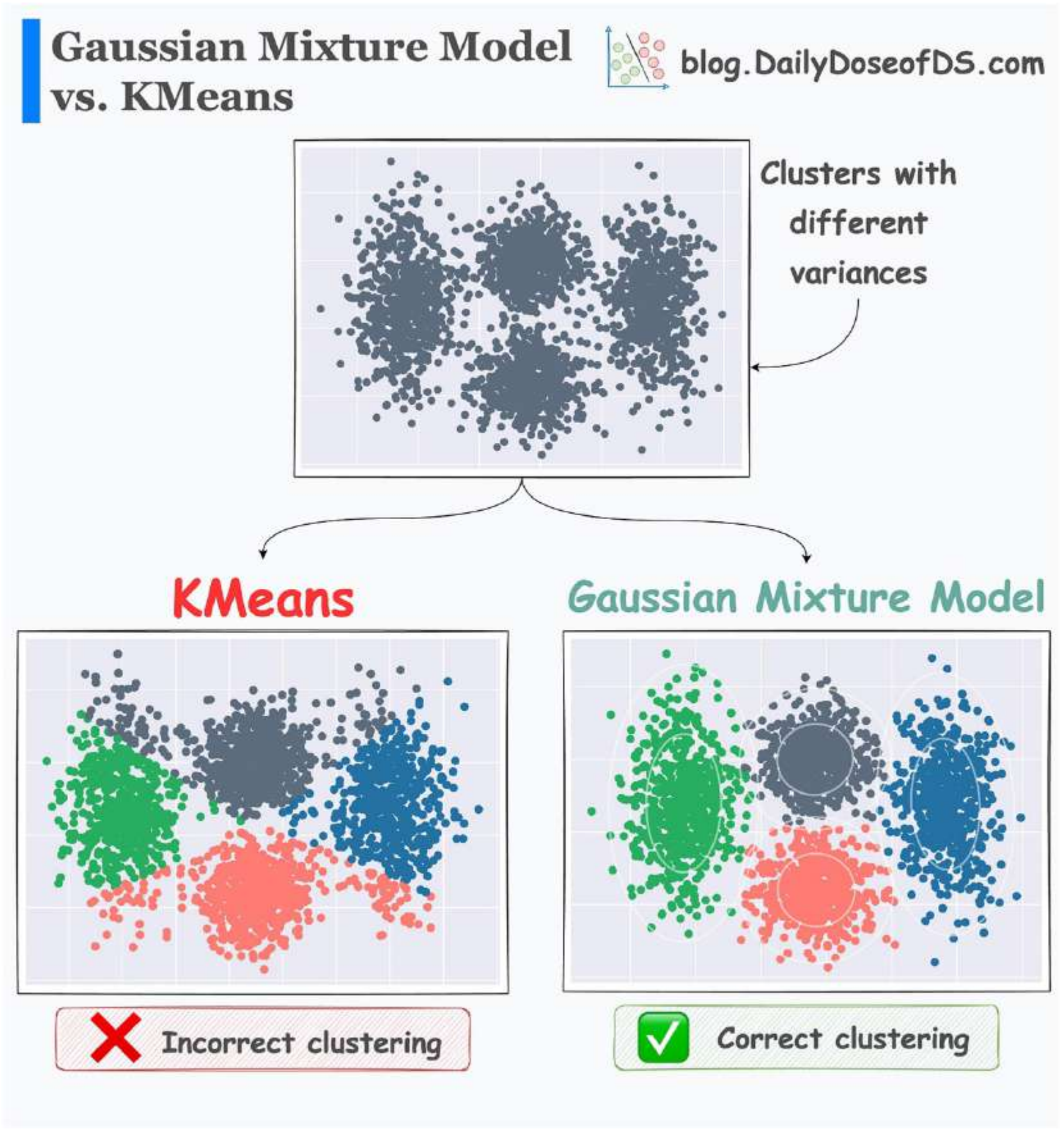
- KMeans learns **centroids**.
- Gaussian mixture models learn a **distribution**.

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.

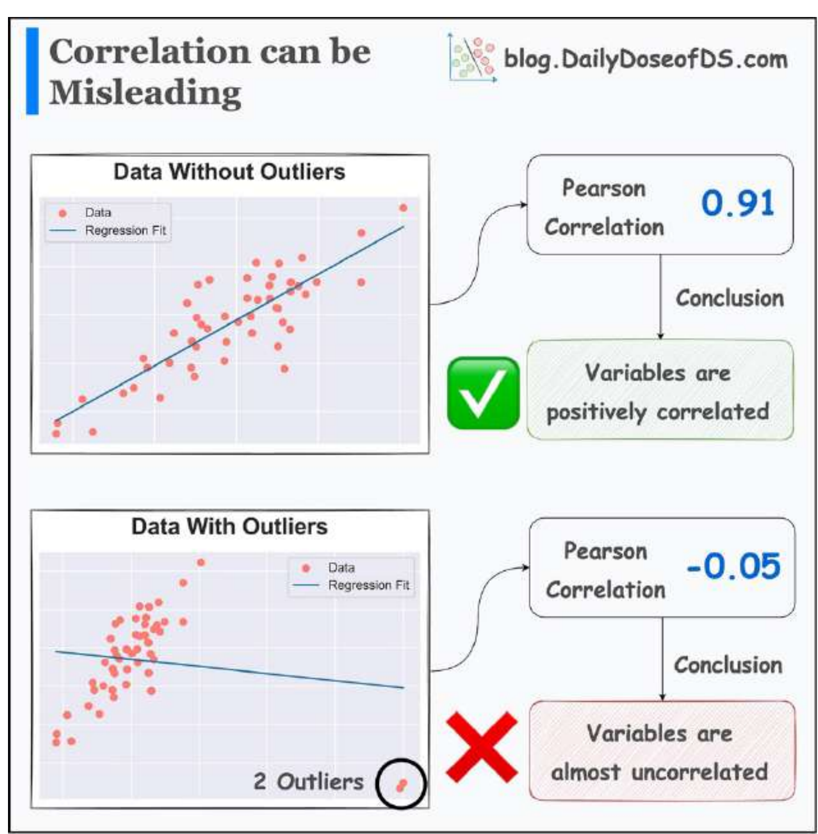
Thus, today's article covers:

- 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.

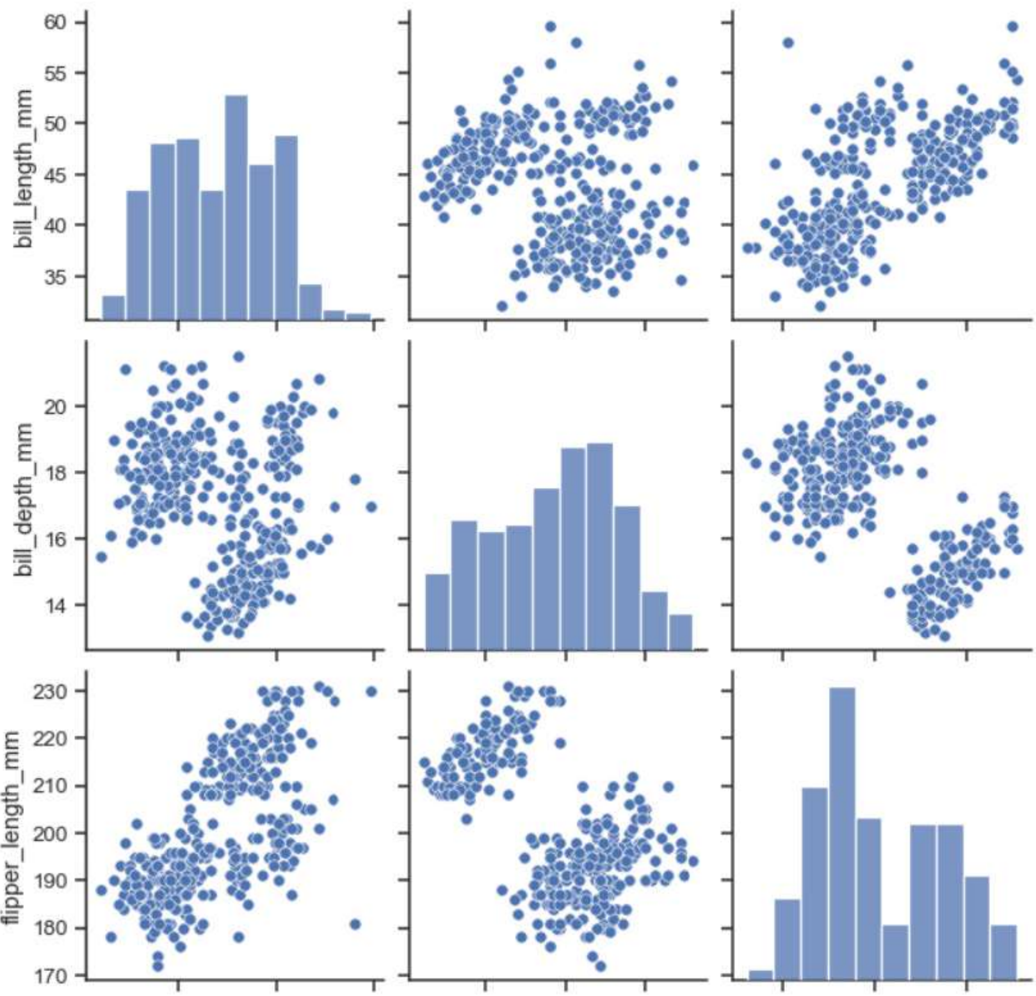
The addition of just two outliers drastically changed:

- 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.

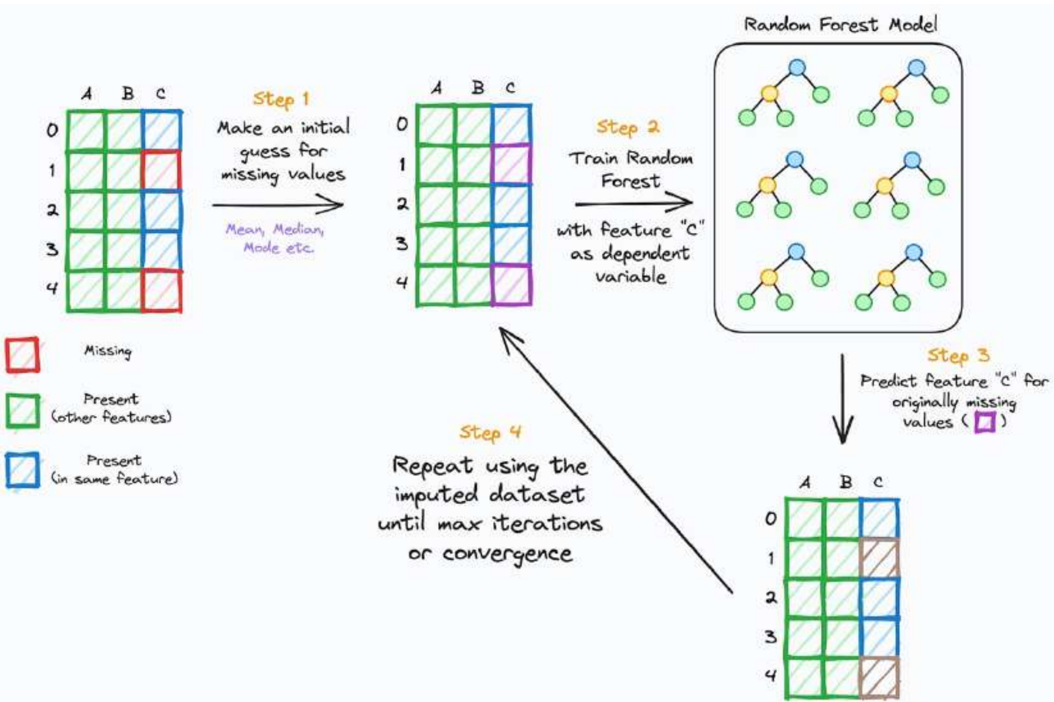
In one of the earlier posts, we discussed kNN imputer. Today's post builds on that by addressing its limitations, which are:

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.

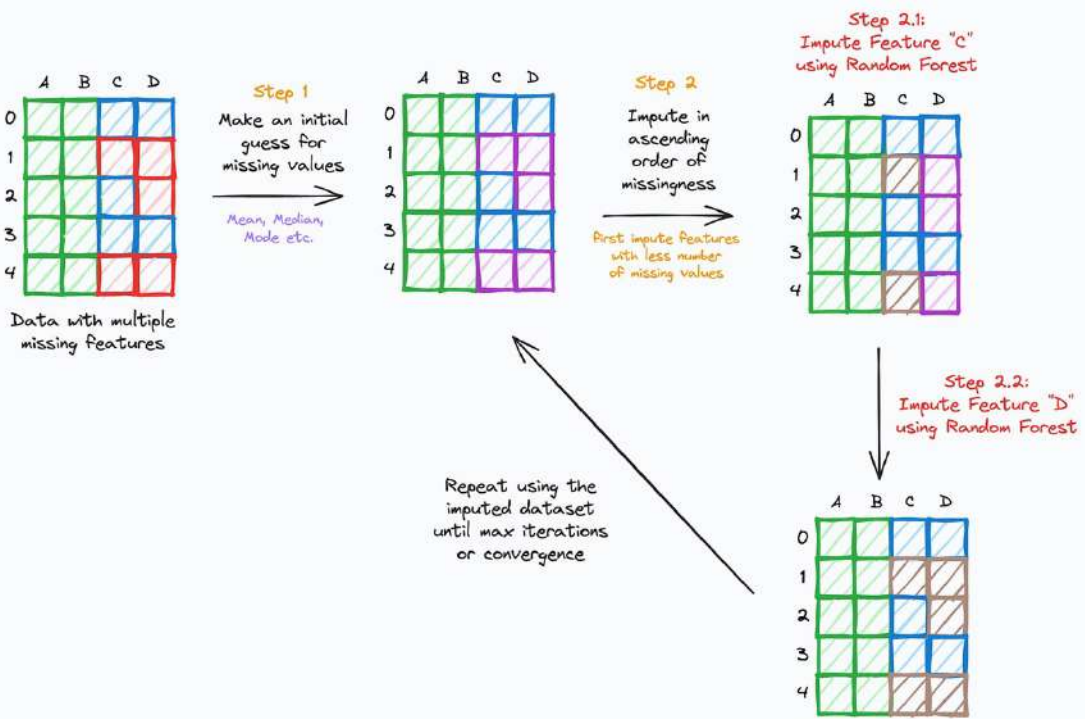
The following figure depicts how it works:



Visual illustration of MissForest imputer

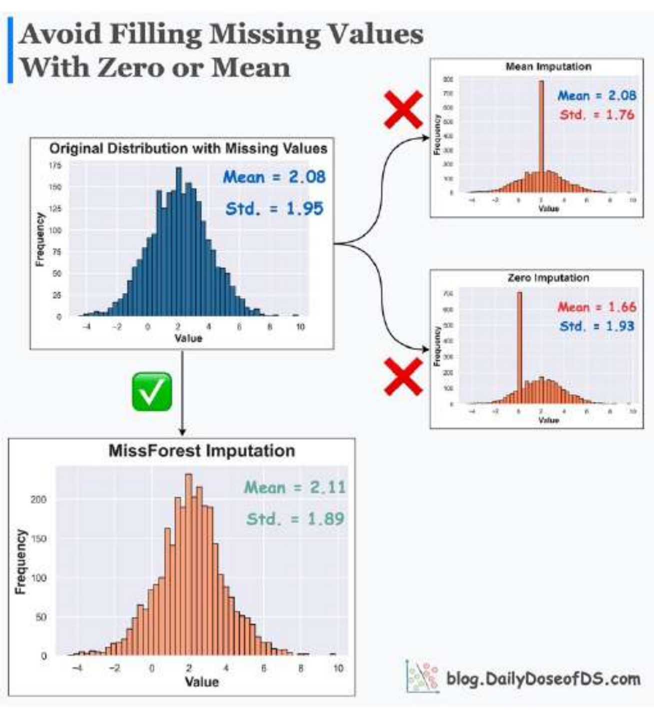
- **Step 1:** To begin, impute the missing feature with a random guess — Mean, Median, etc.
- **Step 2:** Model the missing feature using Random Forest.
- **Step 3:** Impute ONLY originally missing values using Random Forest's prediction.
- **Step 4:** Back to Step 2. Use the imputed dataset from Step 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 missingness — features with fewer missing values are imputed first.

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

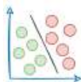


- Mean/Zero alters the summary statistics and distribution.
- MissForest imputer preserves them.

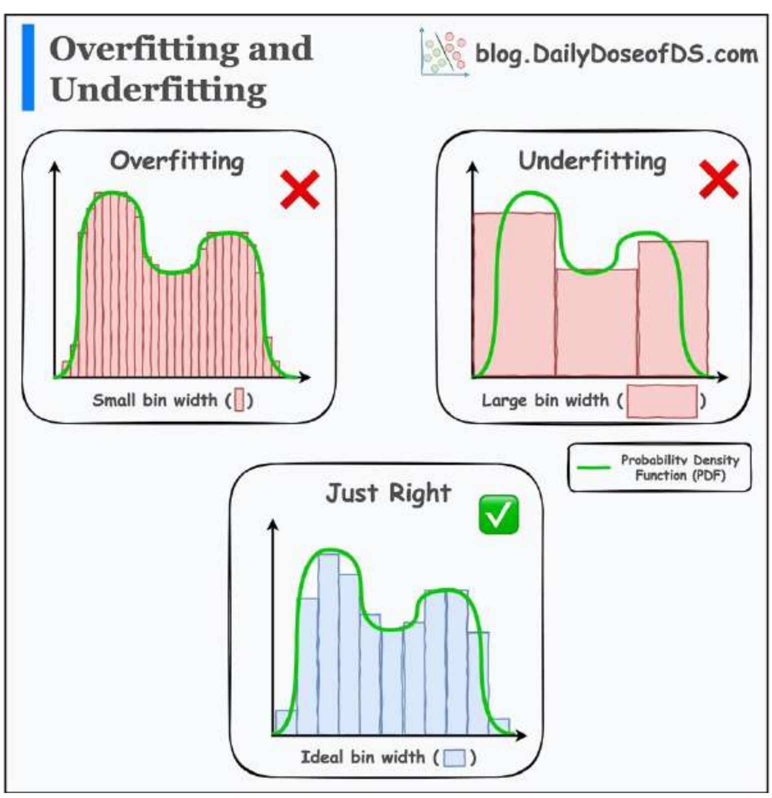
What's more, MissForest can impute even if the data has categorical non-missing features.

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:

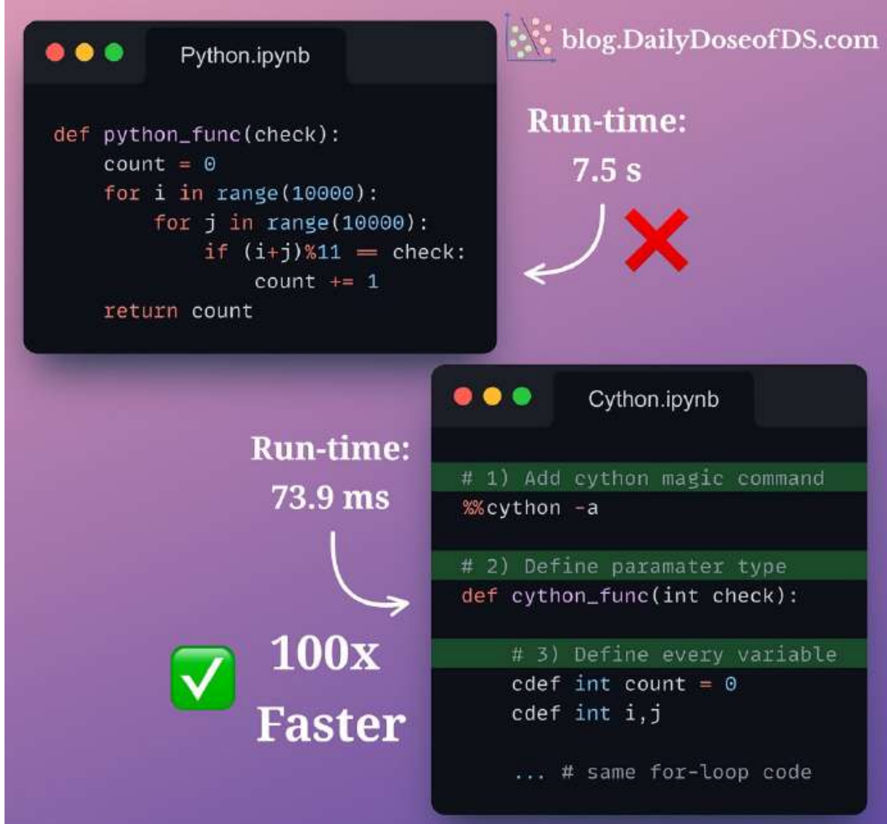
```
In [24]: %%cython -a

         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.

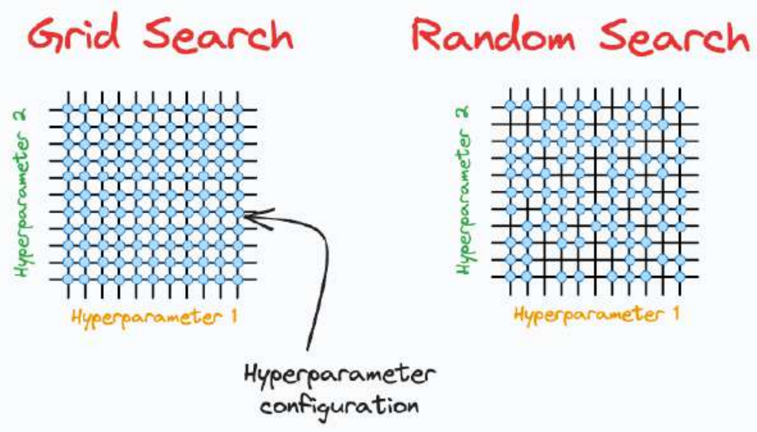
Isn't that cool?

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:

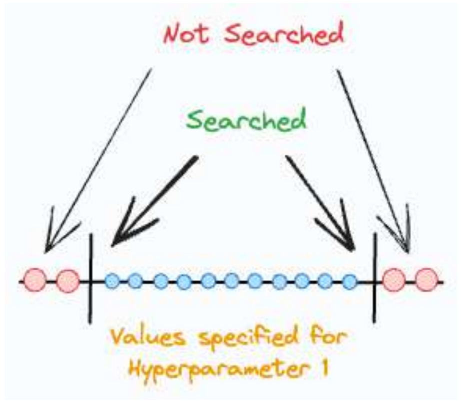
- Grid search
- Random search



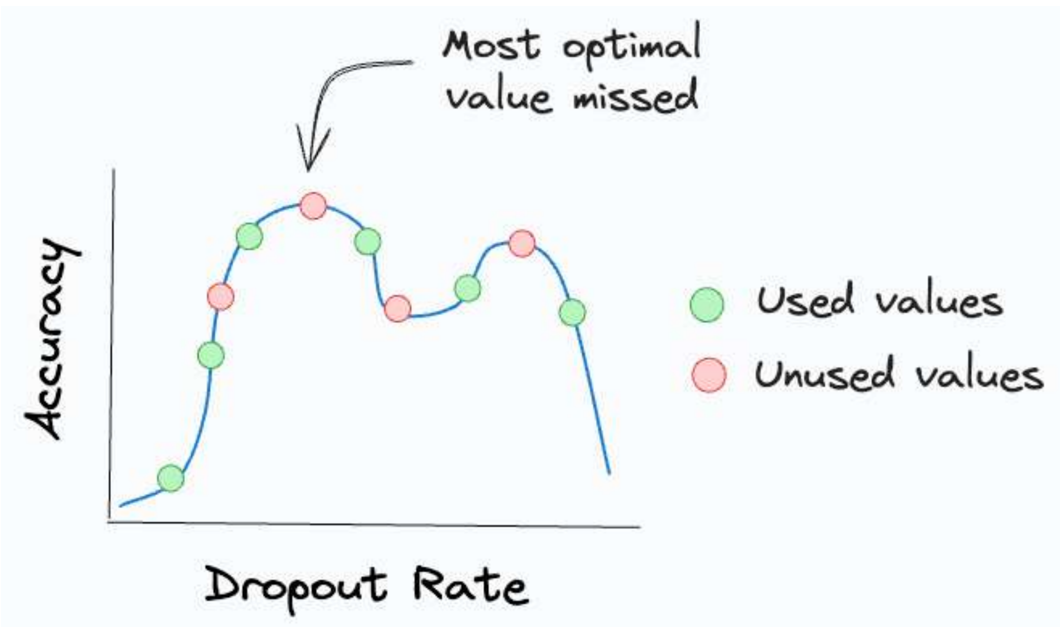
But they have many limitations.

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.



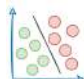
Grid search and random search can only try discrete values for continuous hyperparameters

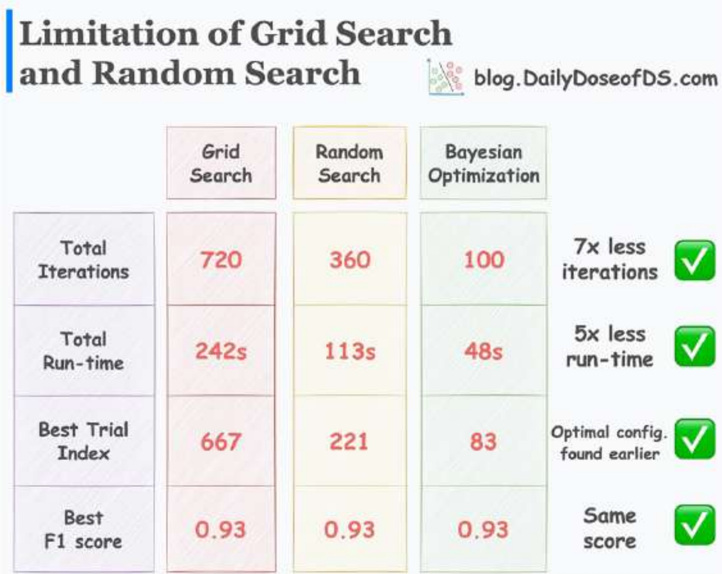
To this end, **Bayesian Optimization** is a highly underappreciated yet immensely powerful approach for tuning hyperparameters.

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**.

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.

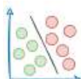
Thus, learning about Bayesian optimization will be immensely valuable if you envision doing the same.

Thus, today's article covers:

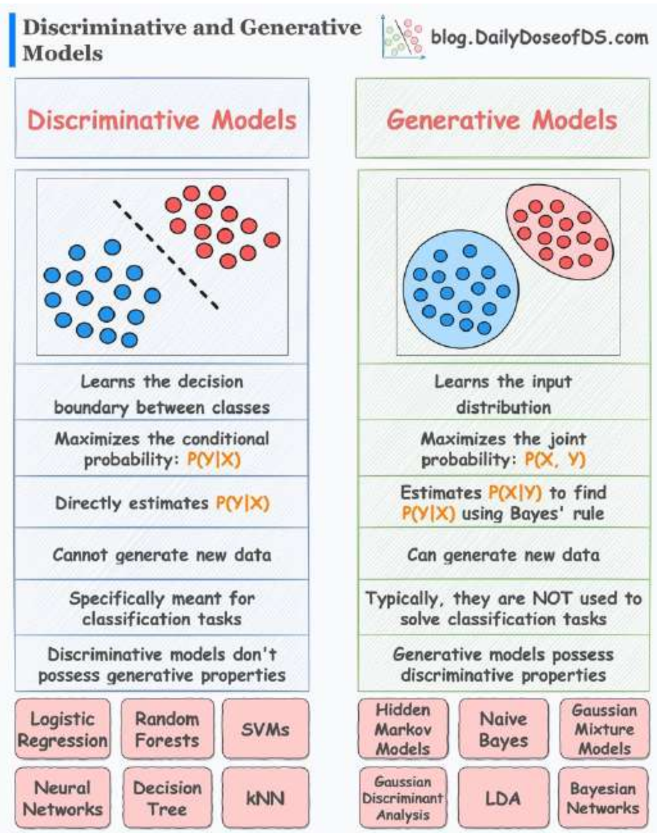
- Issues with traditional hyperparameter tuning approaches.
- What is the motivation for Bayesian optimization?
- How does Bayesian optimization work?
- The intuition behind Bayesian optimization.
- Results from the research paper that proposed Bayesian optimization for hyperparameter tuning.
- **A hands-on 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



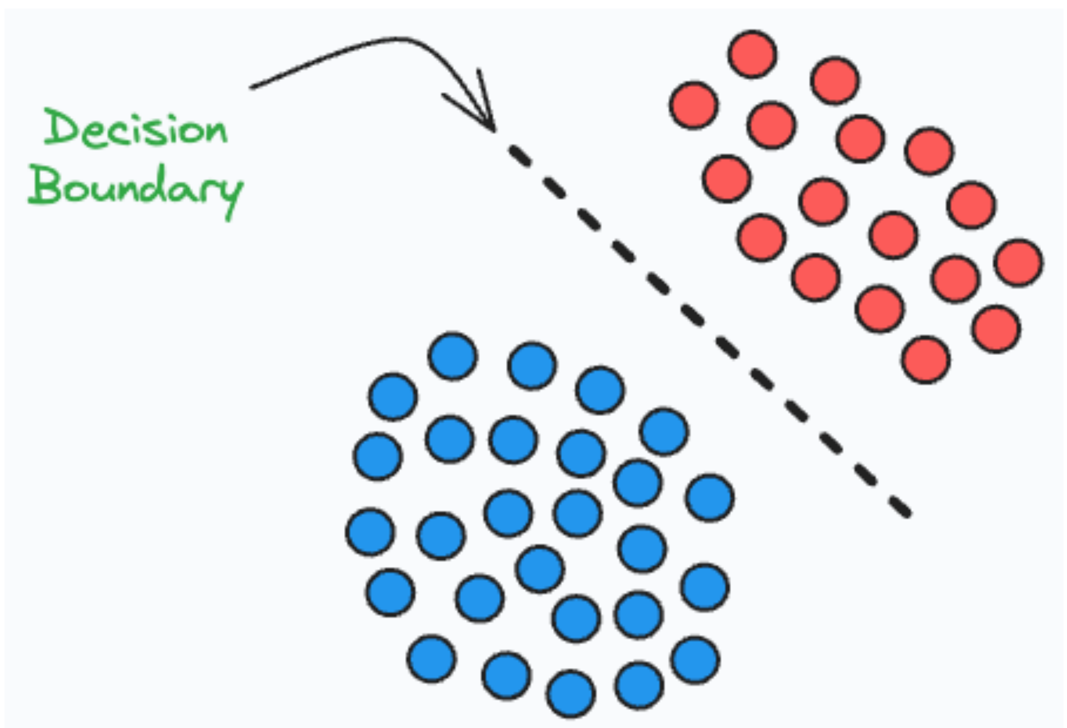
Many machine learning models can be classified into two categories:

- Generative
- Discriminative

This is depicted in the image above.

Today, let's understand what they are.

Discriminative models



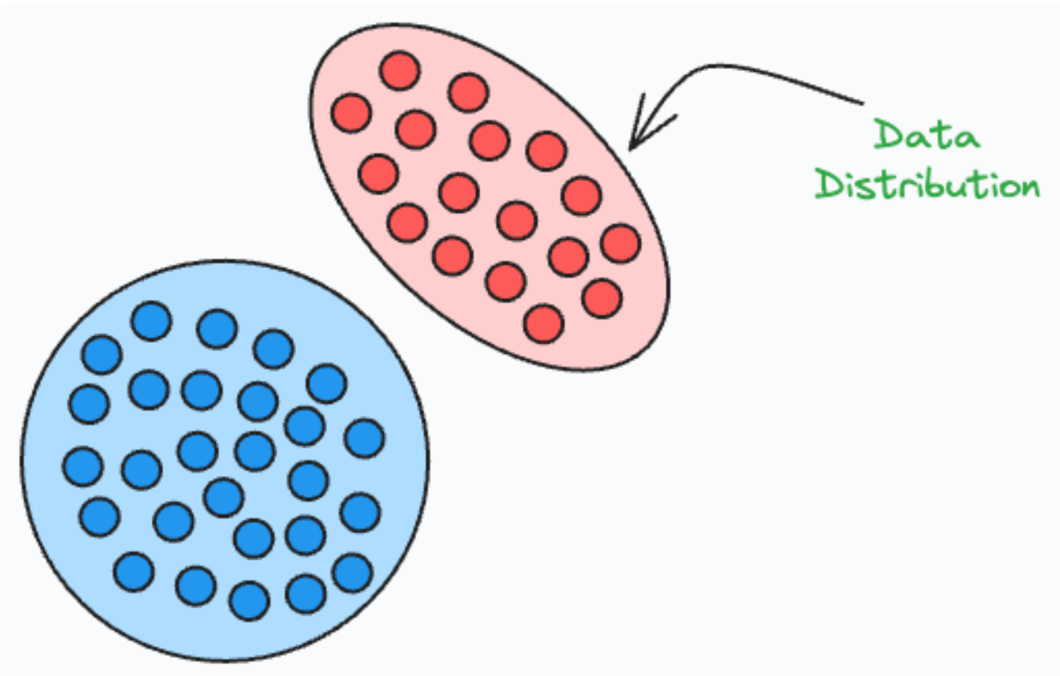
Discriminative models:

- 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:

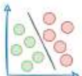
- maximize the joint probability: P(X, Y)
- learn the class-conditional distribution P(X|Y)
- are **typically** not meant for classification tasks.

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.



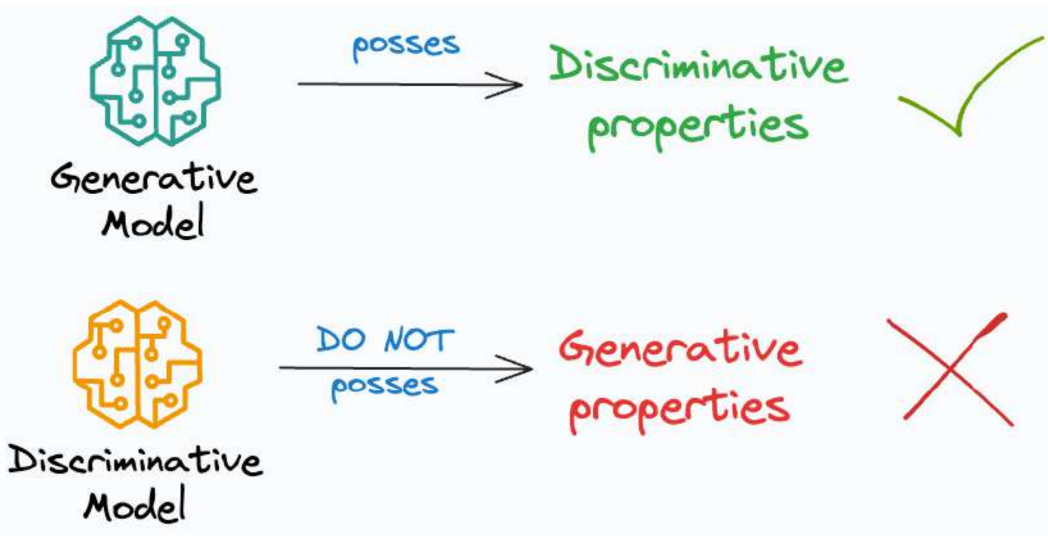
learned from data

$$P(Y \mid X) \sim P(X \mid Y) \cdot P(Y)$$

use this to sample points

However, this is not possible with discriminative models.

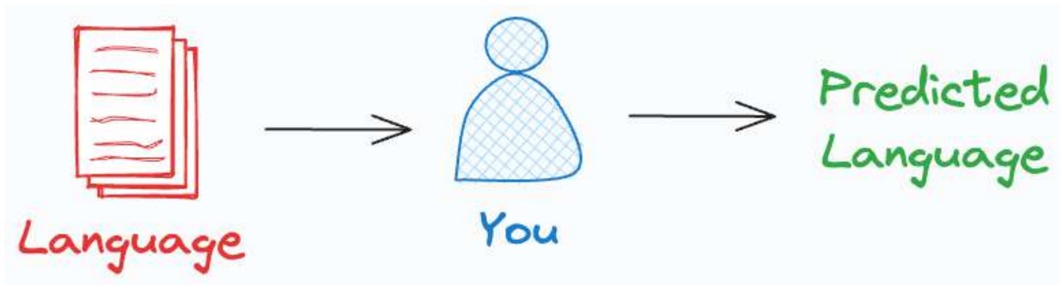
Furthermore, generative models possess discriminative properties, i.e., they can be used for classification tasks (if needed).



However, discriminative models do not possess generative properties.

Let's consider an example.

Imagine yourself as a language classification system.



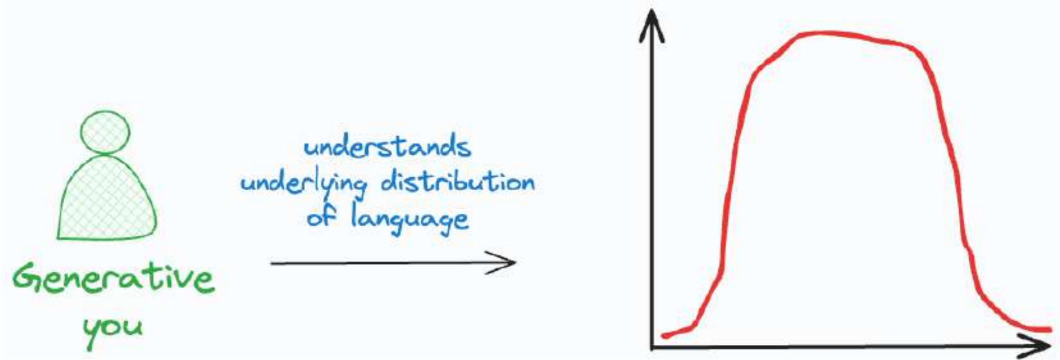
There are two ways you can classify languages.

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 $P(\text{Words, Language})$.

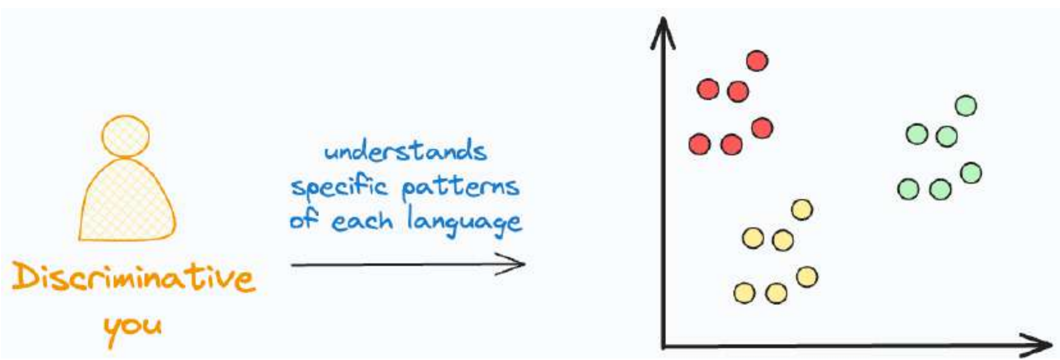


Moreover, as you understand the underlying distribution, now you can generate new sentences, can't you?

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(Language|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.

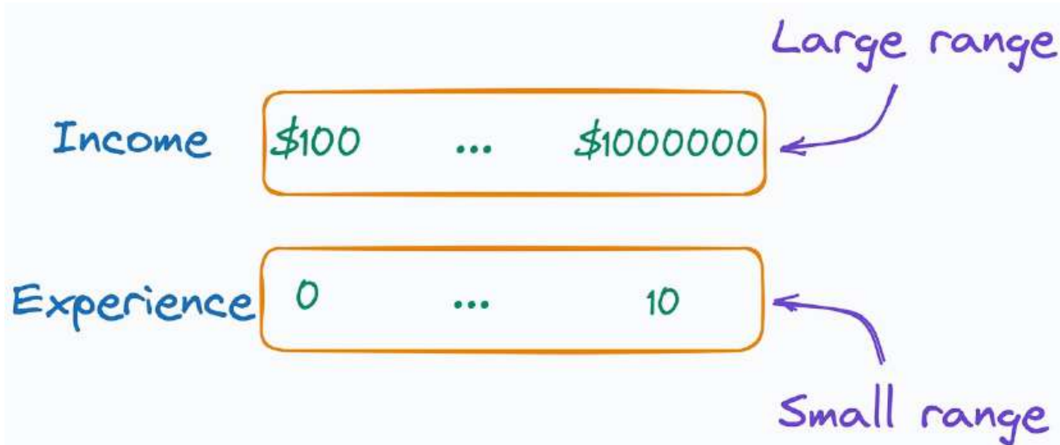
Imagine the amount of data you would need to learn all languages (generative approach) vs. the amount of data you would need to understand some distinctive patterns (discriminative approach).

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

blog.DailyDoseofDS.com

	Test Set Classification Performance		
Algorithm	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

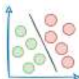
Better performance with feature scaling

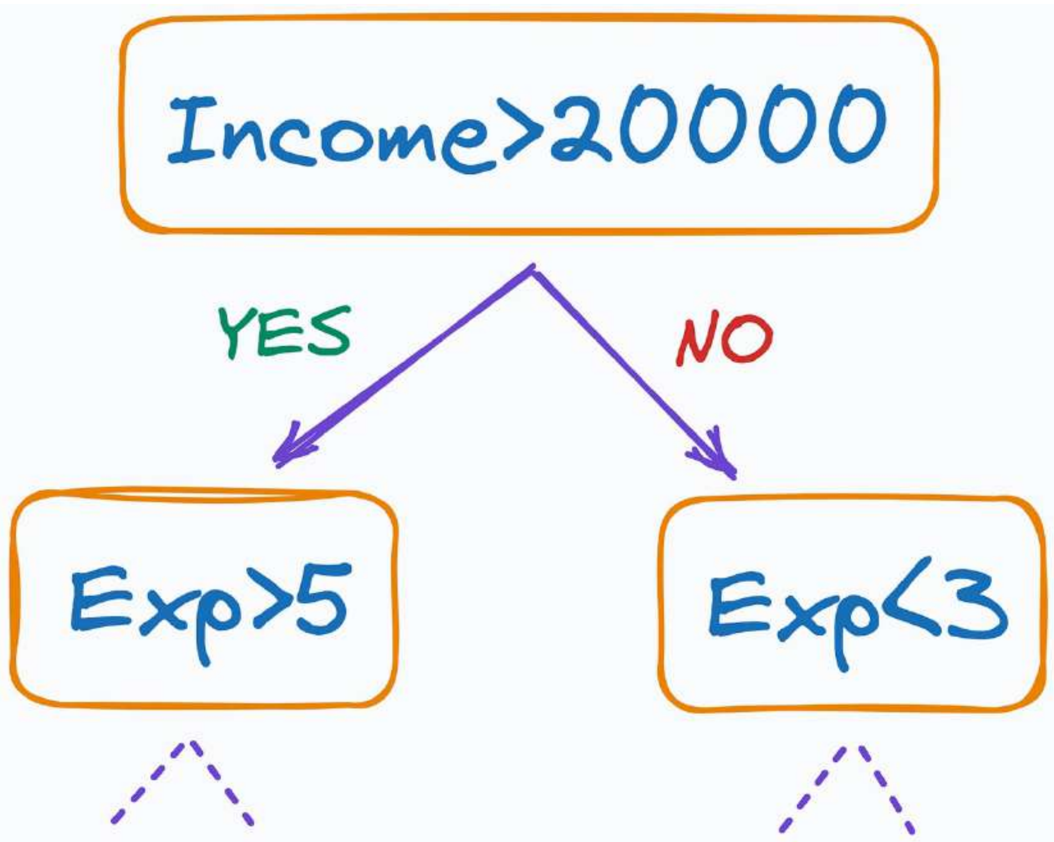
Same performance irrespective of feature scaling

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.





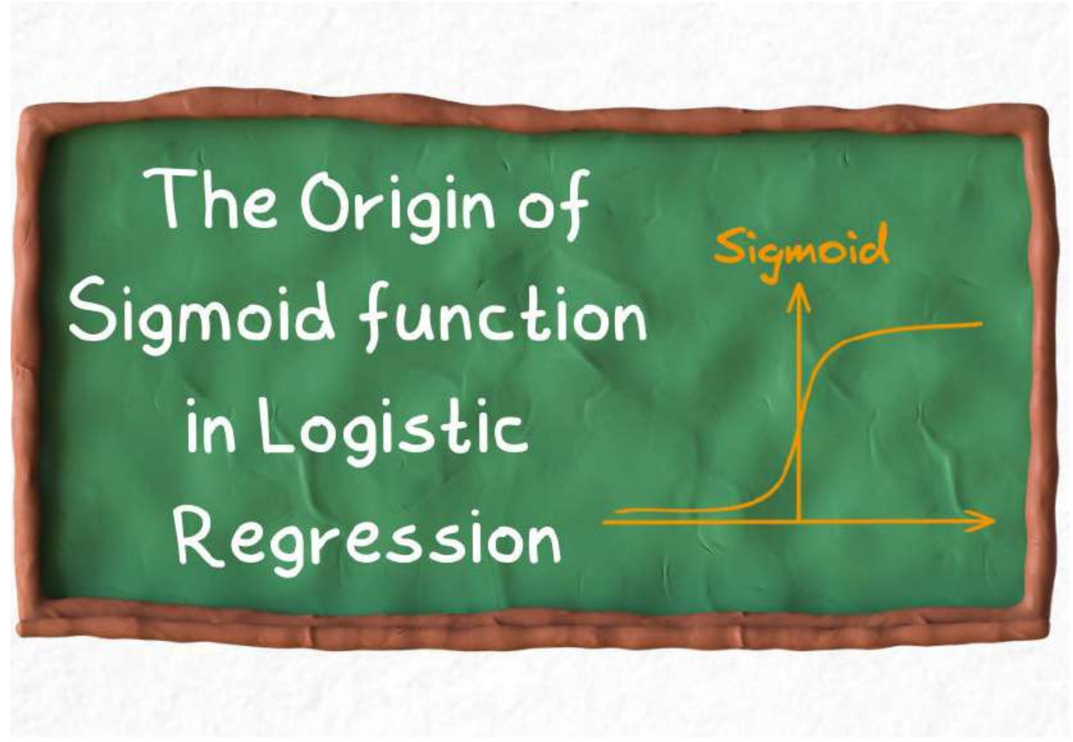
Decision tree

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).

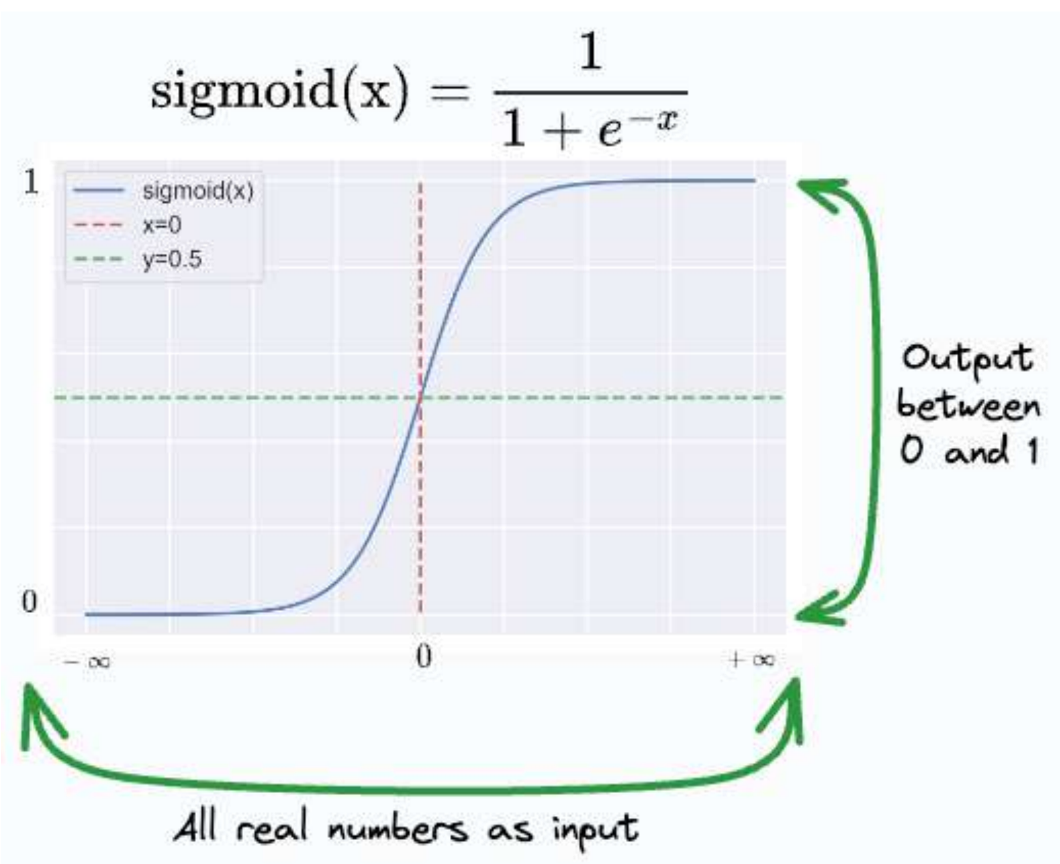
We all know logistic regression does this using the **sigmoid function.**

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

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:

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

$$= \frac{1}{1 + \exp\left(-\frac{(x+2)^2}{2} + \frac{(x-3)^2}{2}\right)}$$

$$= \frac{1}{1 + \exp\left(-5x + \frac{5}{2}\right)}$$

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**

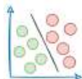
We are covering:

- 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.

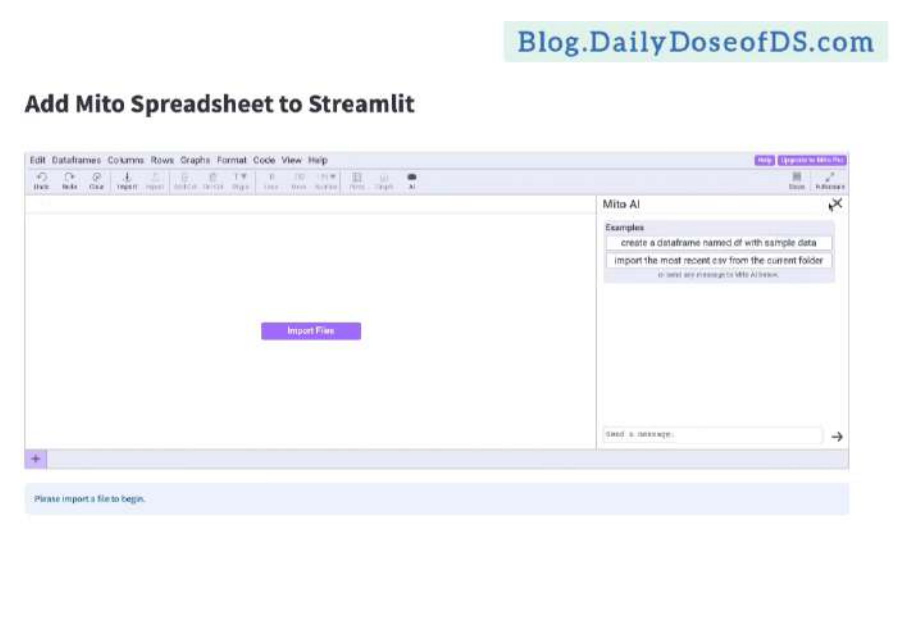
Yet, most no-code tools are often limited in terms of the functionality they support. Thus, flexibility is usually a big challenge while using them.

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:



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:

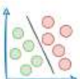
"What proportion of positive predictions were actually positive?"

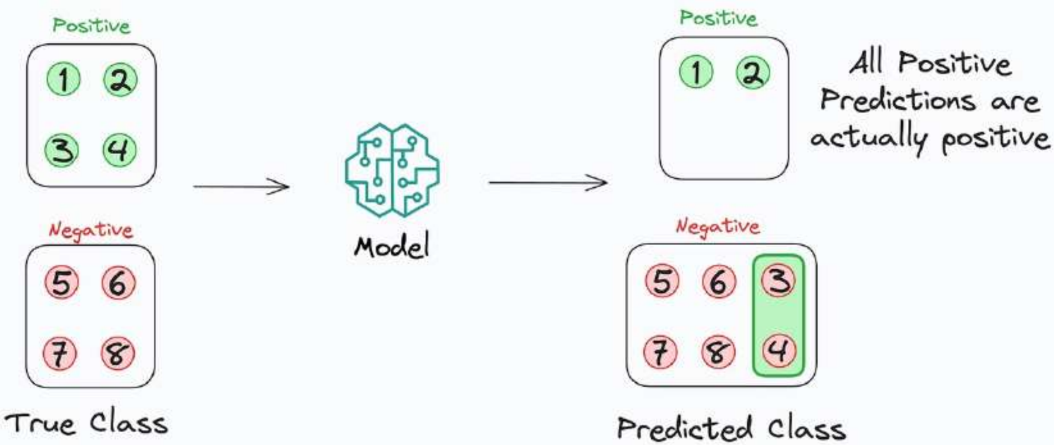
Let's understand that from a mindset perspective.

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:

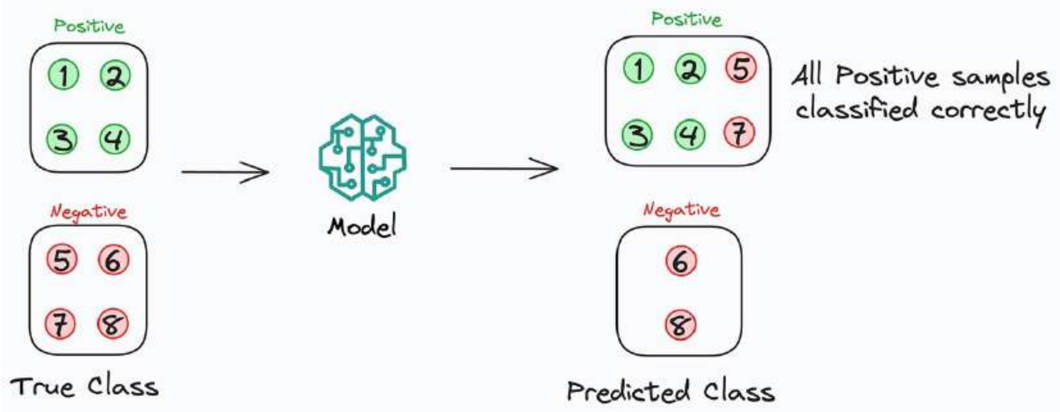
"What proportion of actual positives was identified correctly by the model?"

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.

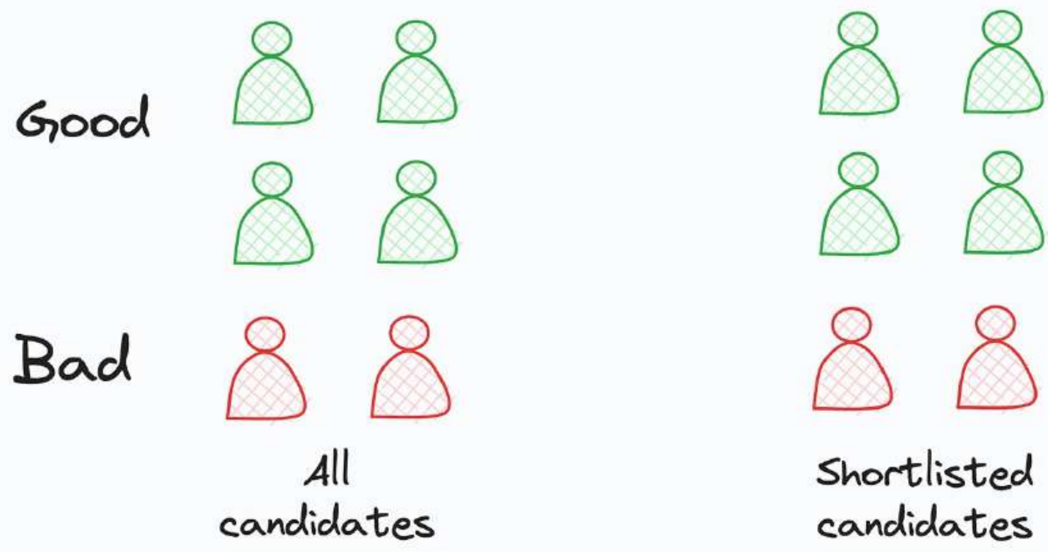
The illustration below is an example of high recall. All positive samples were classified correctly as positive, even though some were actually negative.



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.

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%.

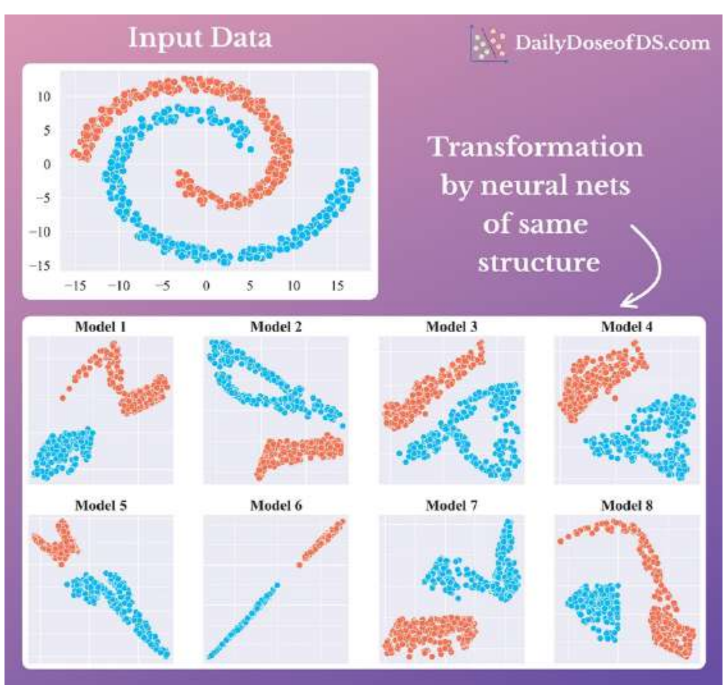
You trained the model again and got the same performance.

Will you call this a reproducible experiment?

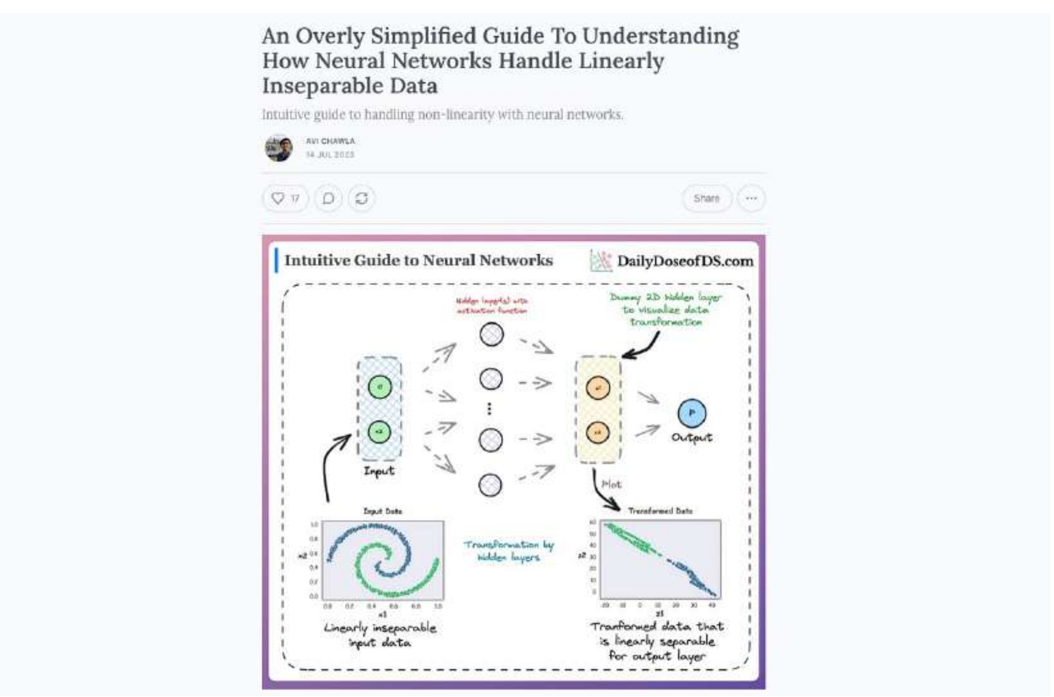
Think for a second before you read further.

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).

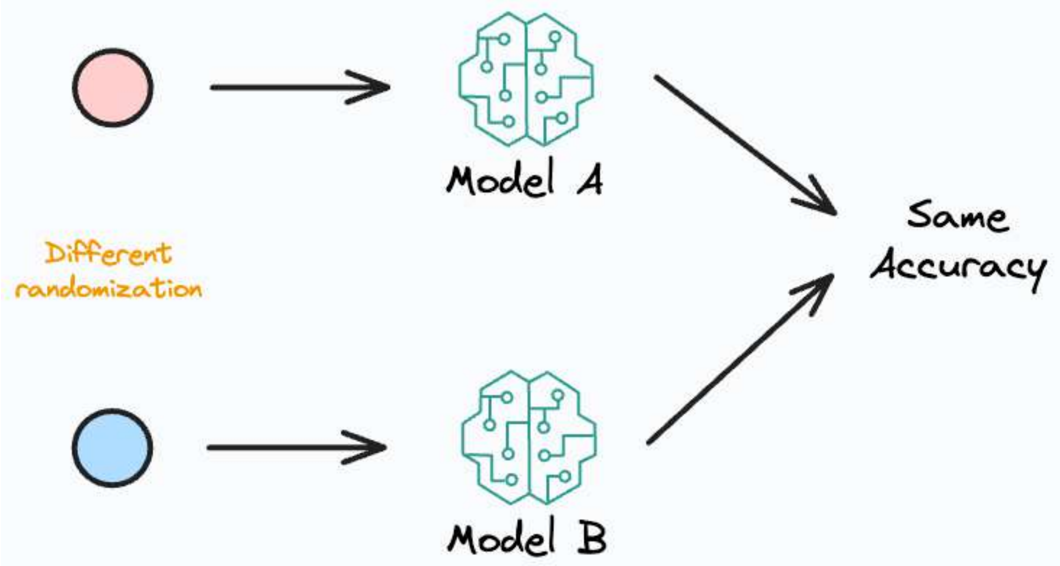
Now will you call this reproducible?

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.

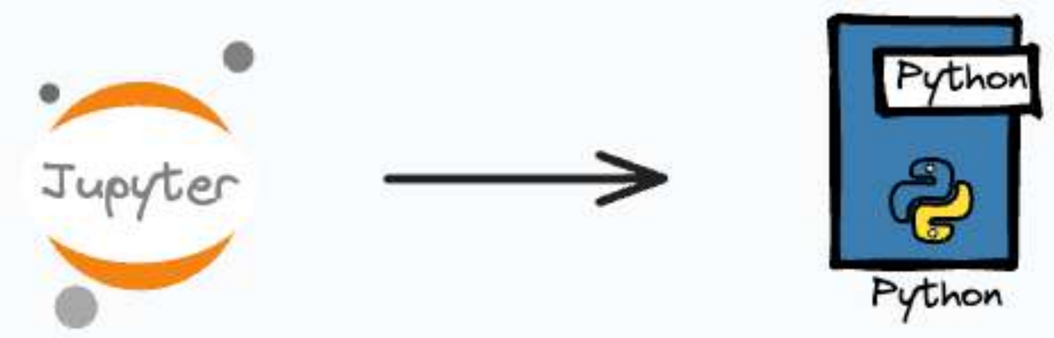
And that is why it is also recommended to set seeds for random generators

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**.

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**.

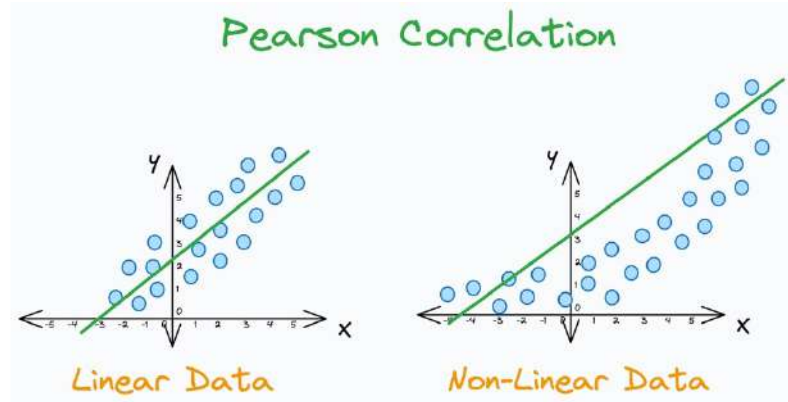
The Biggest Limitation Of Pearson Correlation Which Many Overlook

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:

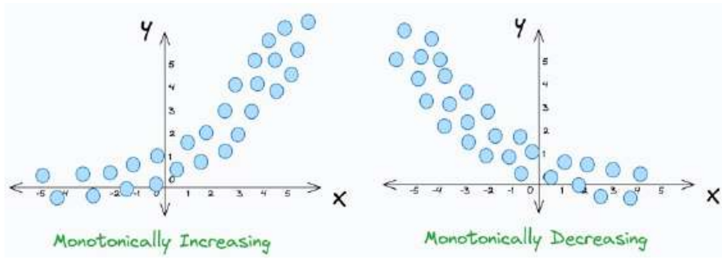
- only measures the linear relationship.
- penalizes a non-linear yet monotonic association.



Pearson correlation only measures the linear relationship

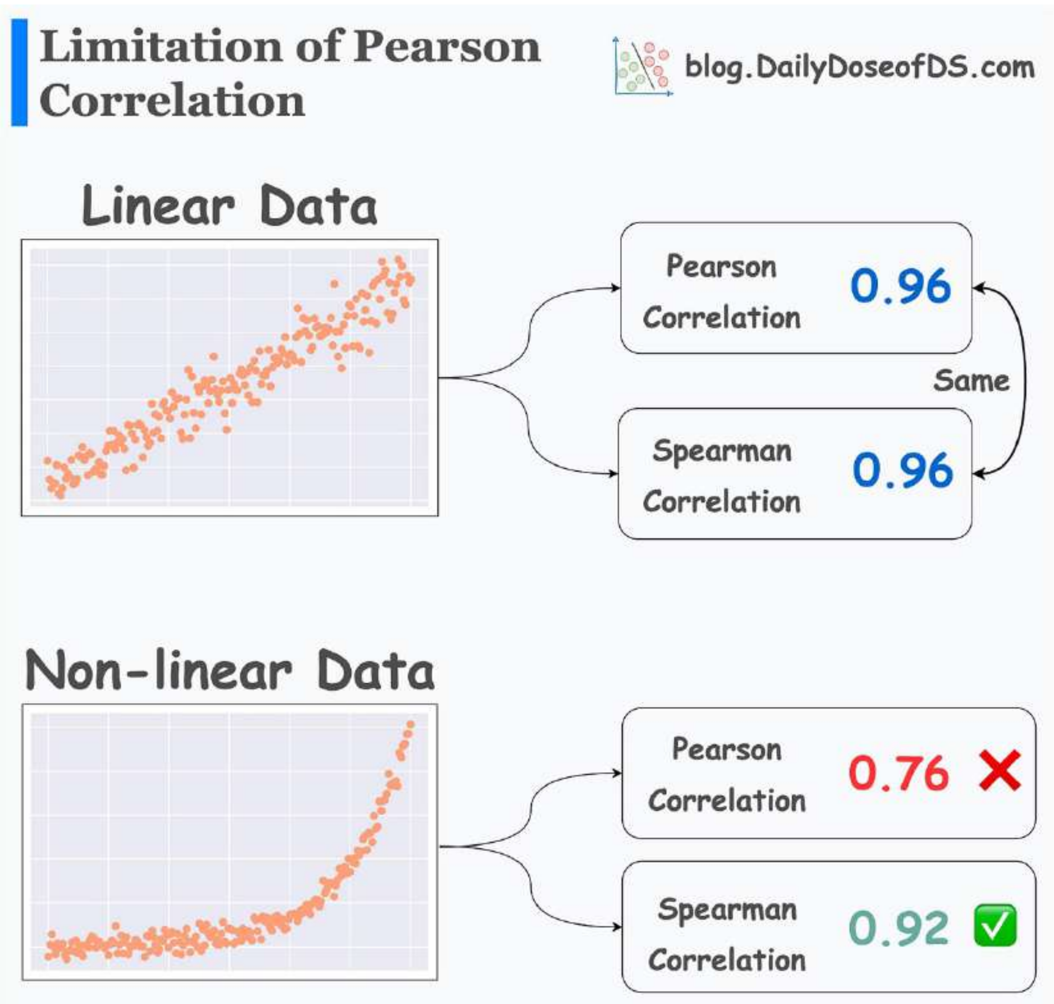
Instead, Spearman correlation is a better alternative.

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



Monotonicity in data

This is evident from the illustration below:



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 Upto 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

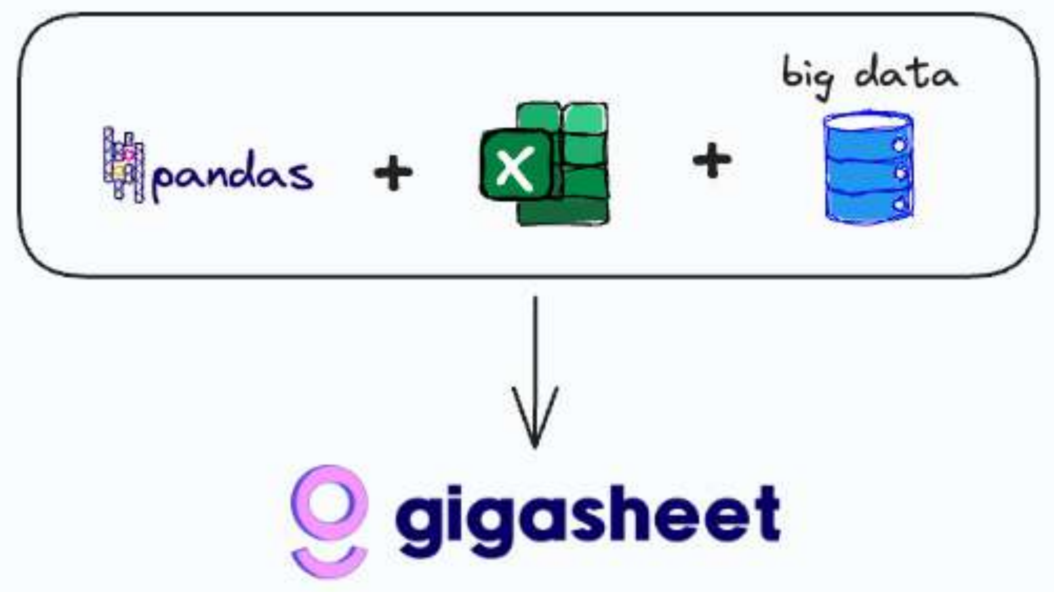
Such cases demand:

- 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**.



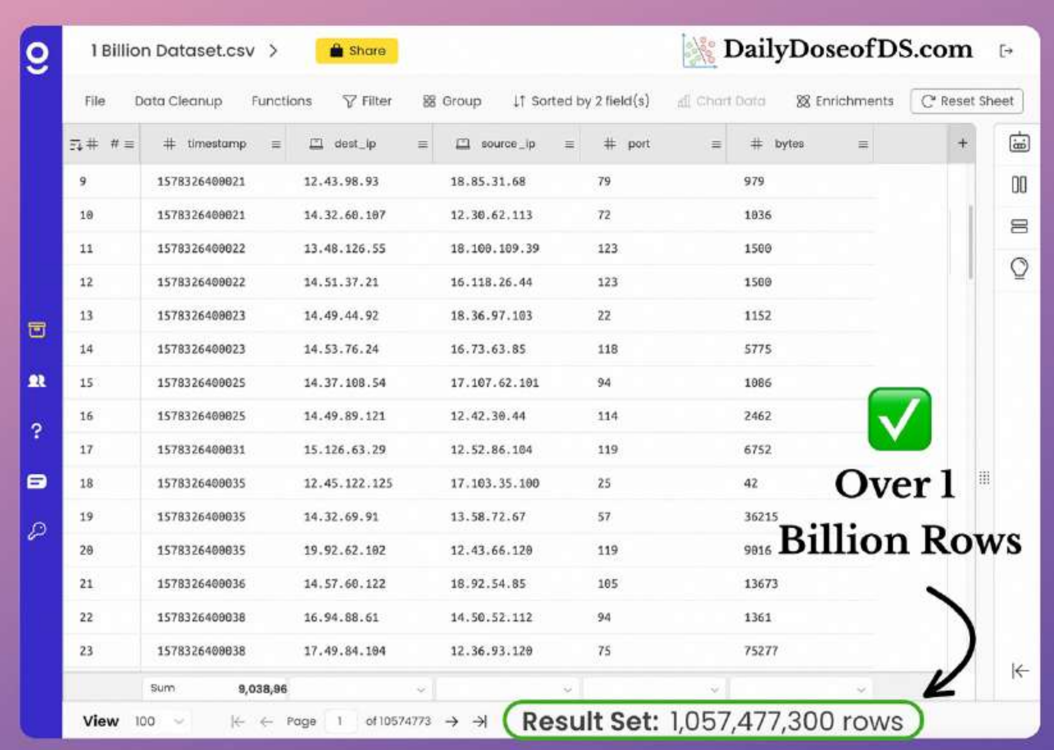
As shown below, I used Gigasheet to load a CSV file with **1 Billion rows** and **47 GB** in size, which is massive.

Loading 1B rows with Gigasheet.

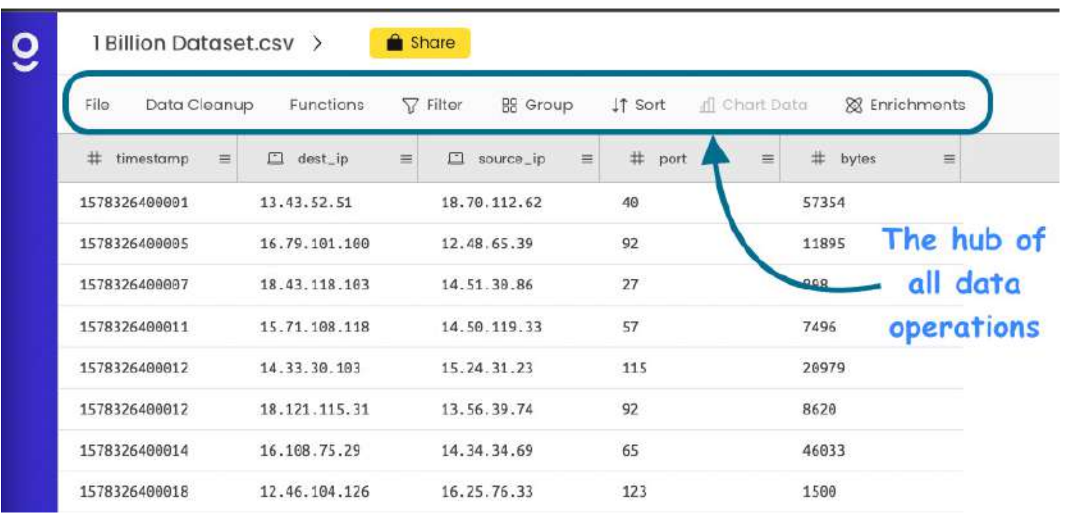
You can perform any data analysis/engineering tasks by simply interacting with a UI.

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.



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



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.

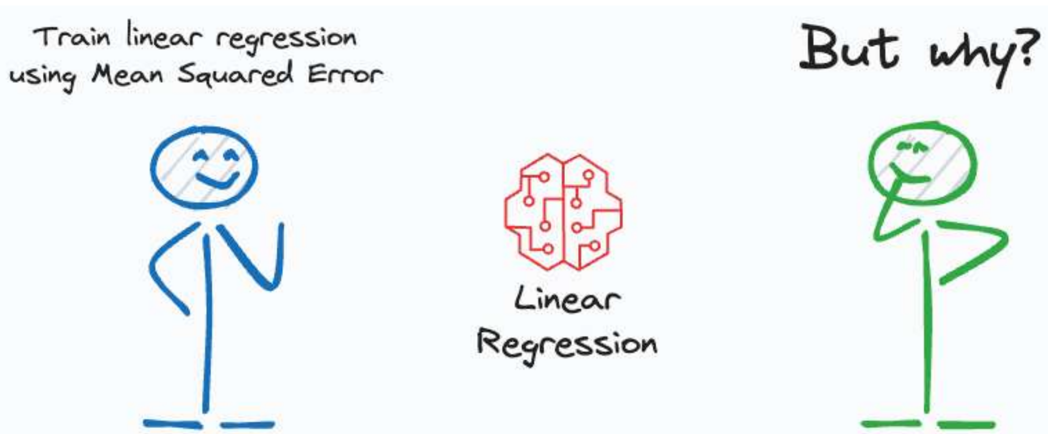
To summarize, Gigasheet immensely simplifies tabular data exploration tasks.

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.**

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:

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

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

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

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

This is called the distribution of y given x; when parametrized by θ

For a specific set of parameters θ, 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 θ. It means that by varying θ, 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.

$$L(\bullet\bullet\bullet\bullet\bullet) = p(\bullet) \cdot p(\bullet) \cdot p(\bullet) \cdot p(\bullet) \cdot p(\bullet)$$

independent observations

product of observing individual observations

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

Thus, we get:

$$L(\theta) = \prod_{i}^{n} p(y_i|x_i;\theta)$$
$$= \prod_{i}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} \exp(-\frac{(y_i - \theta^T x_i)^2}{2\sigma^2})$$

Likelihood function

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

$$log(L(\theta)) = log(\prod_{i}^{n} p(y_i|x_i;\theta))$$
$$= log(\prod_{i}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} \exp(-\frac{(y_i - \theta^T x_i)^2}{2\sigma^2}))$$

Taking the log on both sides in the likelihood function

Simplifying, we get:

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

distribute the logarithm

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

distribute the summation

$$= n \cdot log(\frac{1}{\sqrt{2\pi\sigma^2}}) - \sum_{i}^{n} \frac{(y_i - \theta^T x_i)^2}{2\sigma^2}$$

constant

To reiterate, the objective is to find the θ that maximizes the above expression.

But the first term is independent of θ. 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 σ. 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.

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.

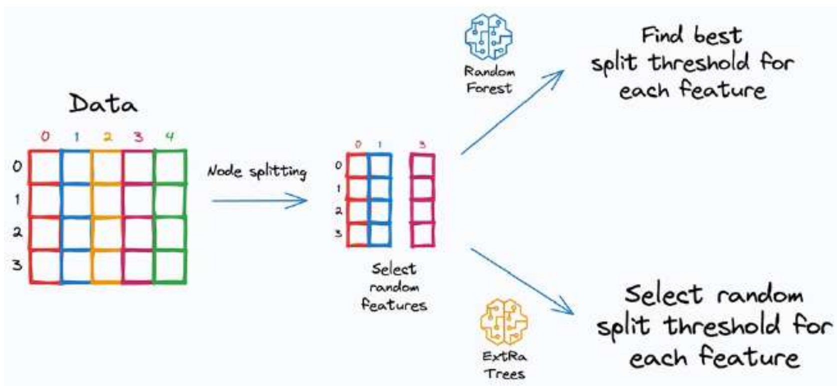
👉 Note:

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

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.

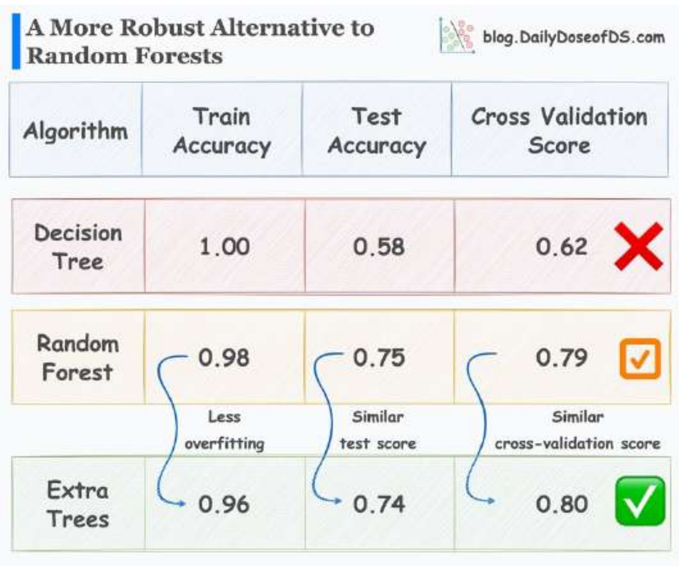


Random Forest vs. ExtRa Trees

- 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

blog.DailyDoseofDS.com

Algorithm	Train Accuracy	Test Accuracy	Cross Validation Score
Decision Tree	1.00	0.58	0.62 ✗
Random Forest	0.98	0.75	0.79 ☑
	Less overfitting	Similar test score	Similar cross-validation score
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.

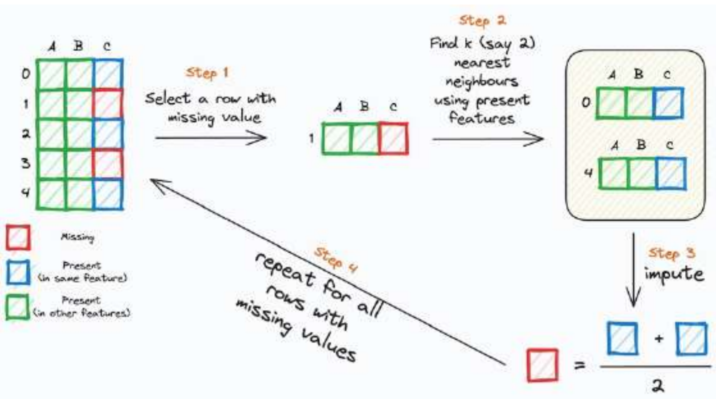
Instead, always try to impute missing values with more precision.

kNN imputer is often a great choice in such cases.

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

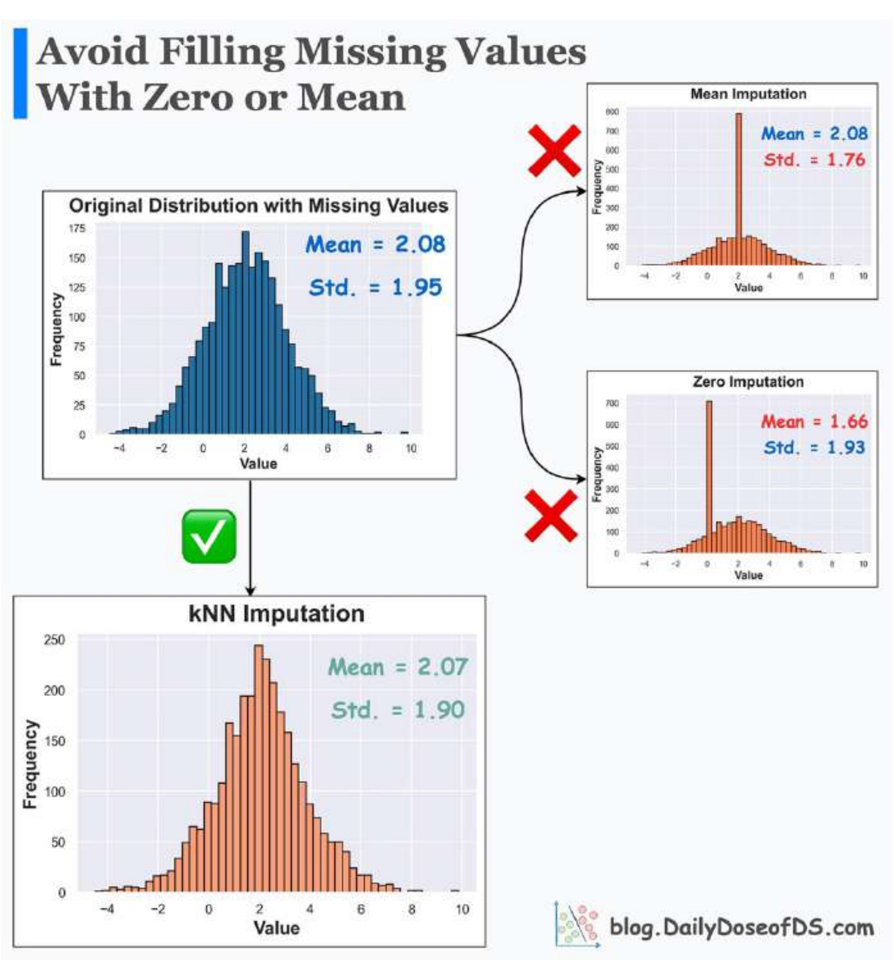
Missing features are imputed by running a kNN on non-missing feature values.

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.

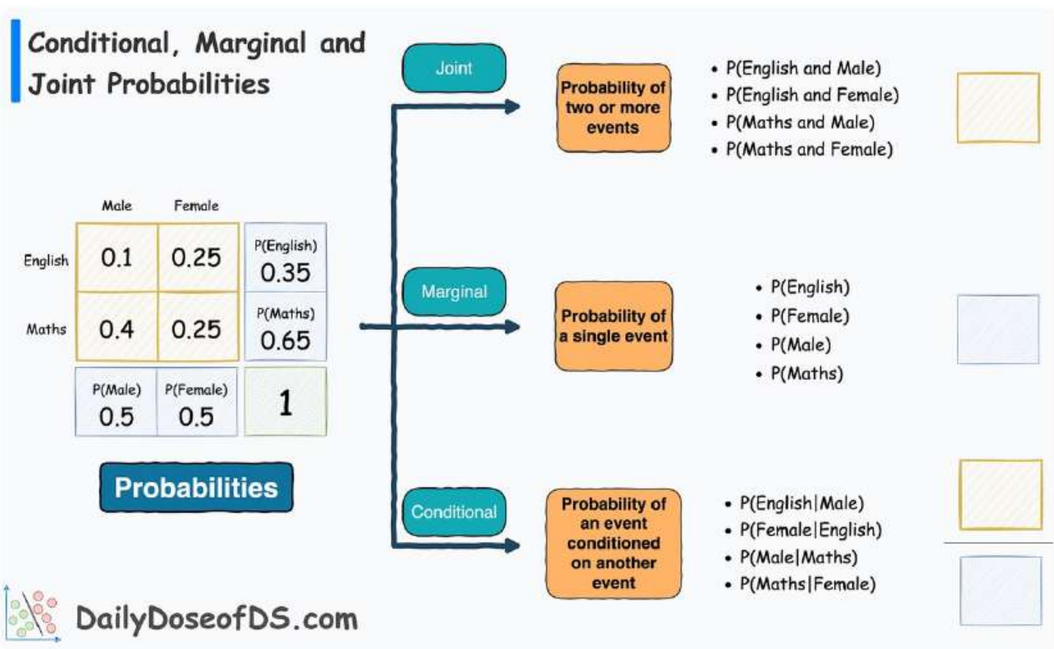


Mean and Zero imputation vs. kNN imputation

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

Get started with kNN imputer: Sklearn Docs.

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

DailyDoseofDS.com

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

Distance to kth Nearest Neighbour

y=0.14

Elbow Point

Distance

Data Point Number

Step 2) Select value at elbow point as Epsilon

DBSCAN(eps = 0.12)

DBSCAN(eps = 0.14)

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

R2 is quite popularly used all across data science and statistics to assess a model.

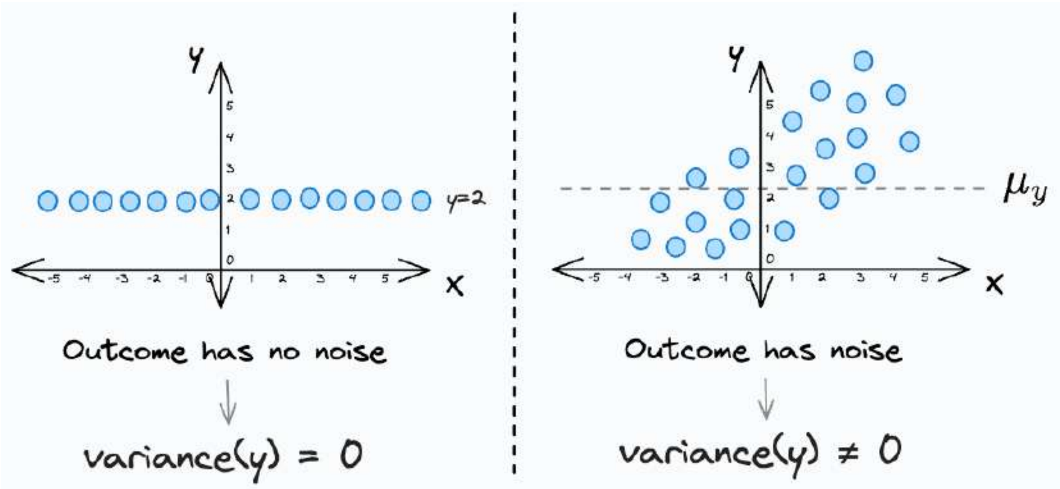
Yet, contrary to common belief, it is often interpreted as a performance metric for evaluating a model, when, in reality, it is not.

Let's understand!

R2 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 R2.

This means that solely optimizing for R2 as a performance measure:

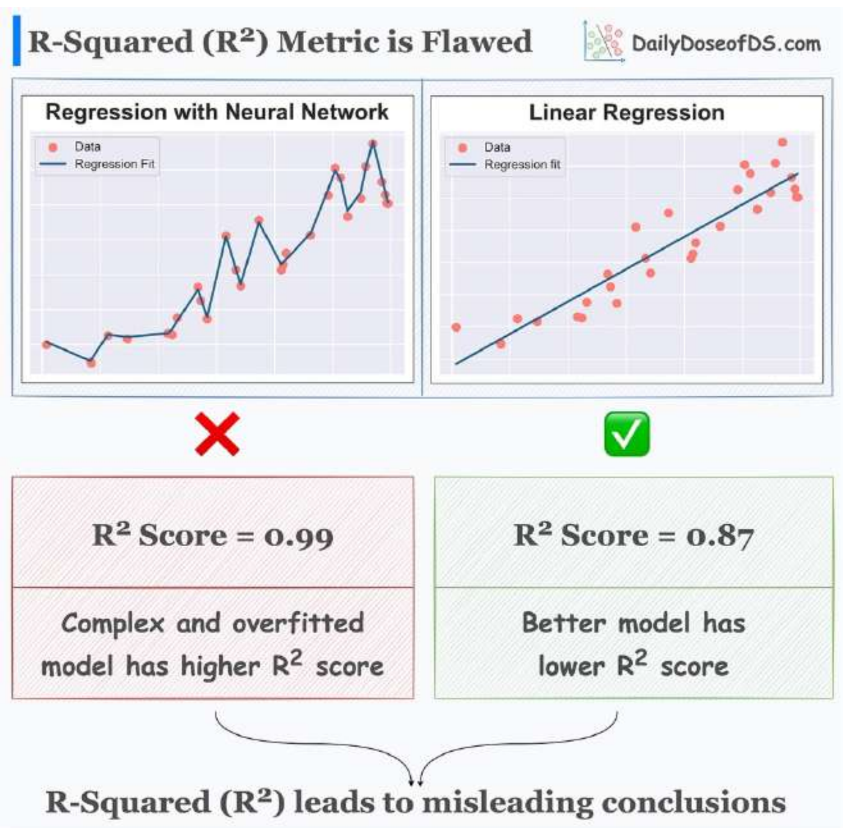
- promotes 100% overfitting.
- leads us to engineer the model in a way that captures random noise instead of underlying patterns.

It is important to note that:

- R2 is NOT a measure of predictive power.
- Instead, R2 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 R2 of 1.
- A better and more generalized model gets a lower R2 score.

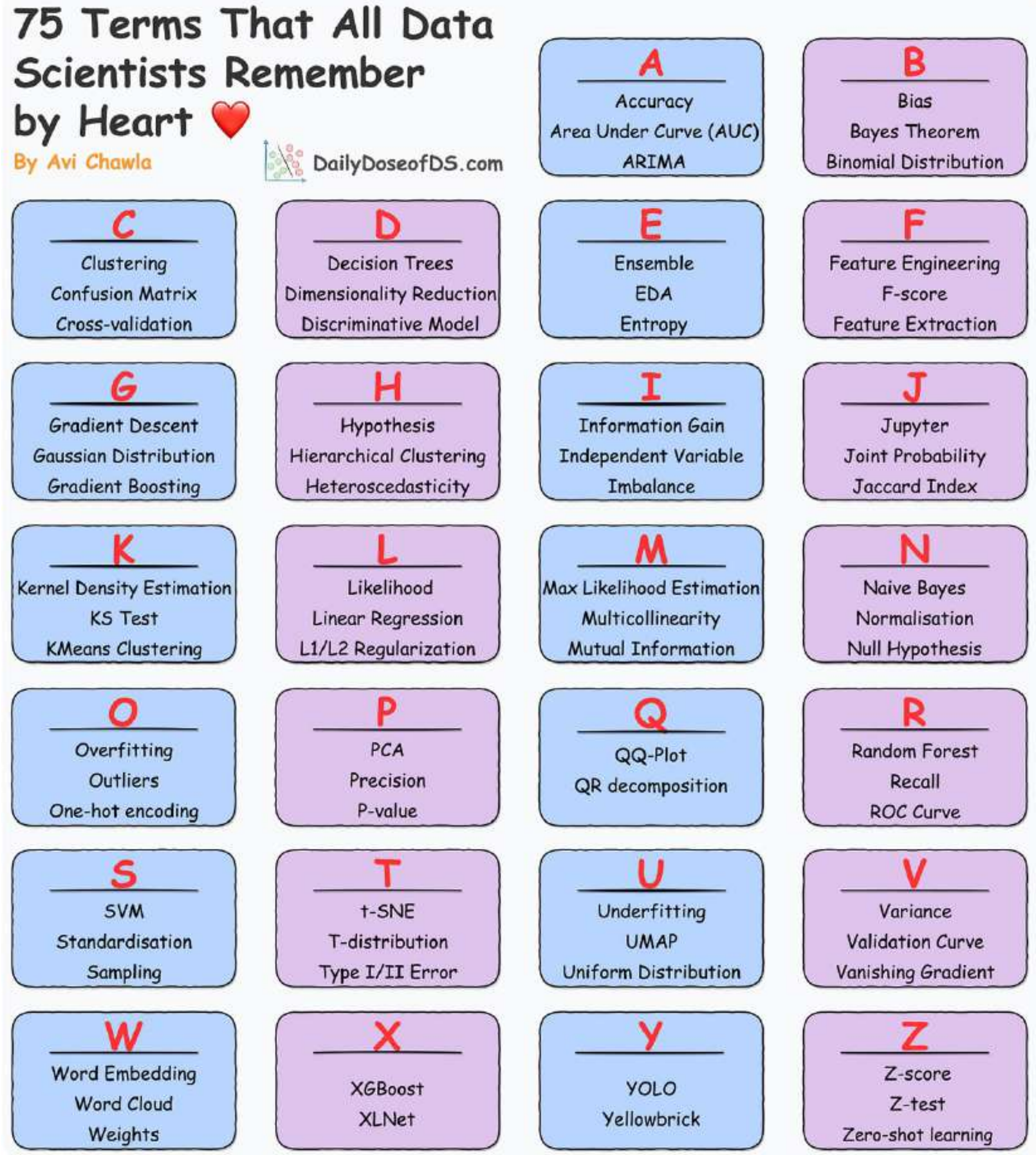
Some other flaws of R2 are:

- R2 always increases as you add more features, even if they are random noise.
- In some cases, one can determine R2 even before fitting a model, which is weird.

👉 Read my full blog on the A-Z of R2, what it is, its limitations, and much more here: **Flaws of R2 Metric**.

👉 Over to you: What are some other flaws in R2?

75 Key Terms That All Data Scientists Remember By Heart



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.

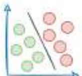
  - **Independent Variable**: Variable that is manipulated in an experiment to observe its effect on the dependent variable.
  - **Imbalance**: Situation where the distribution of classes in a dataset is not equal.
- J:
  - **Jupyter**: Interactive computing environment used for data analysis and machine 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.

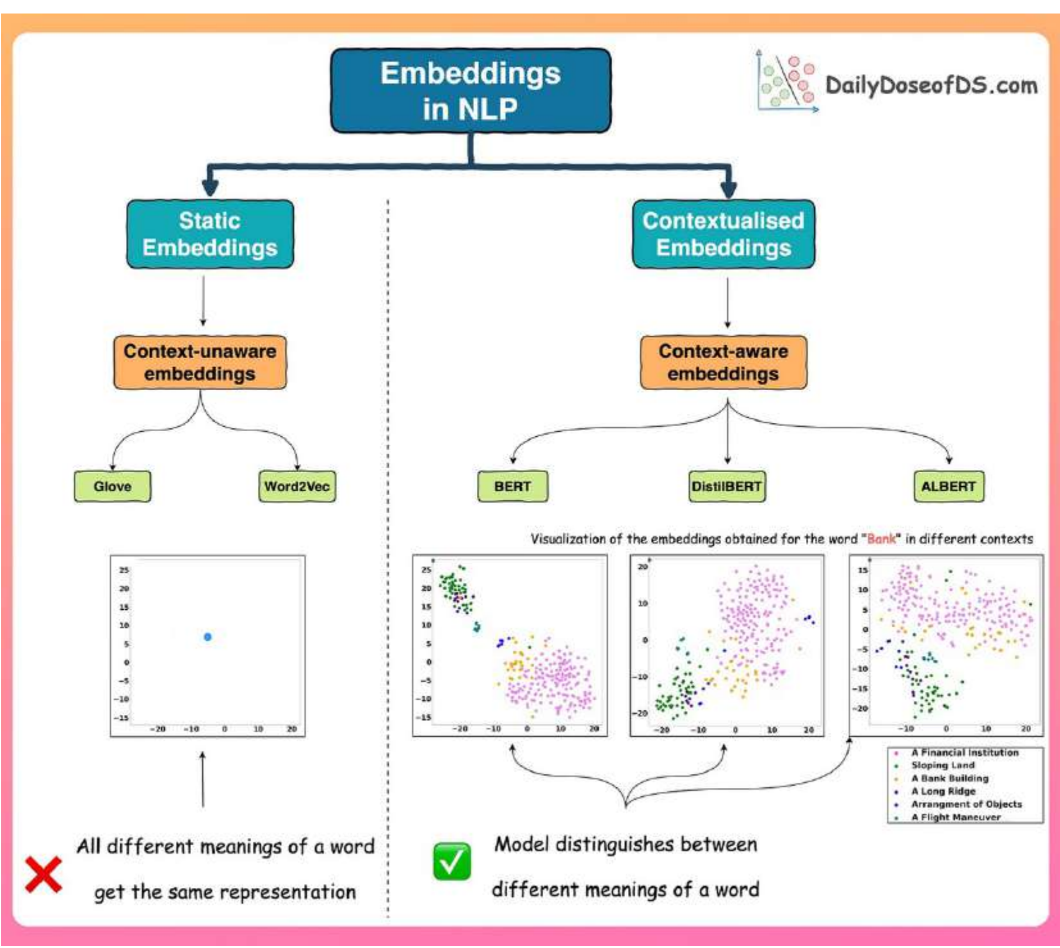
- R:
  - **Random Forest**: Ensemble learning method using multiple decision trees to make predictions.
  - **Recall**: Proportion of true positive predictions among all actual positive instances in a classification model.
  - **ROC Curve (Receiver Operating Characteristic Curve)**: Graph showing the 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.

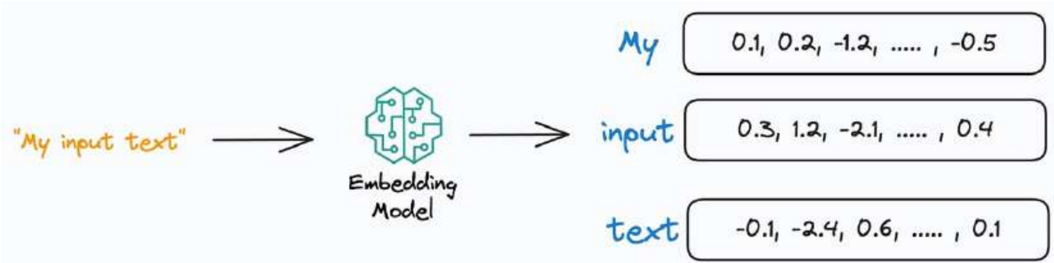
👉 Over to you: Of course, a lot has been left out here. As an exercise, can you add more terms to this?



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.

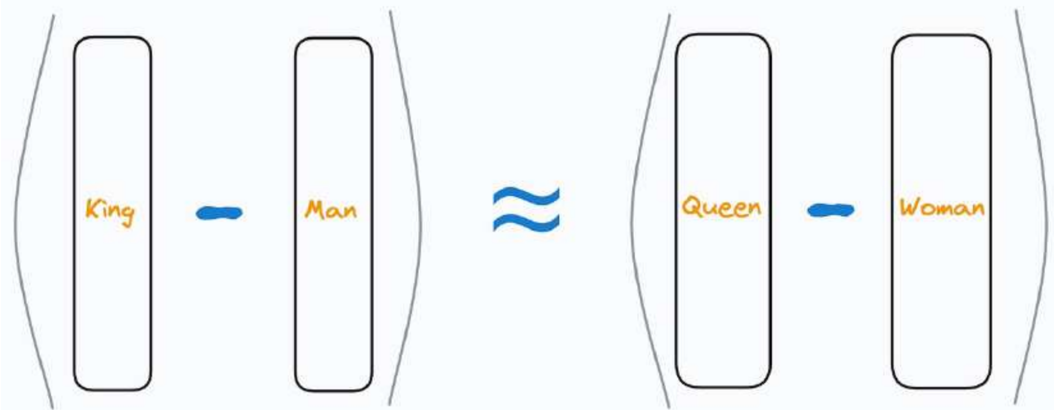
…and other researchers may utilize those embeddings in their projects.

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 (King - Man) + 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.

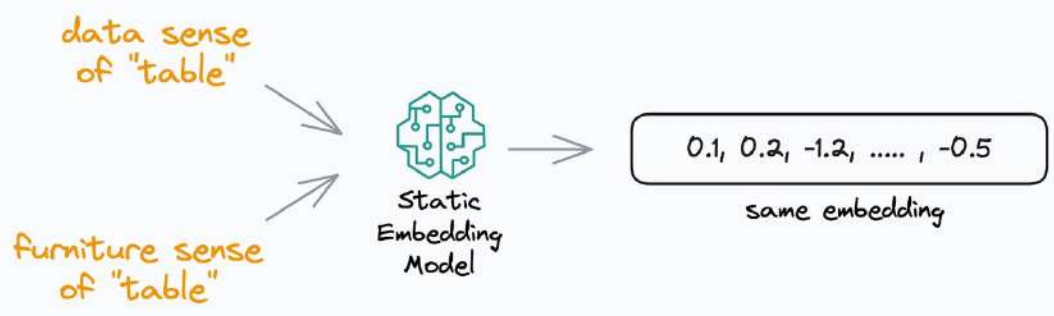
Consider the following two sentences:

- "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.

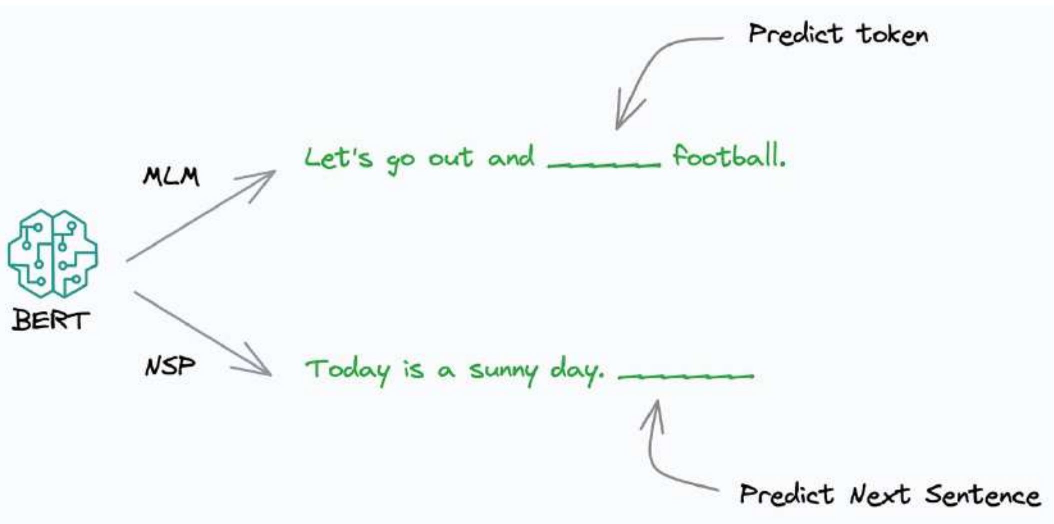


Same embedding for different usages of a word

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

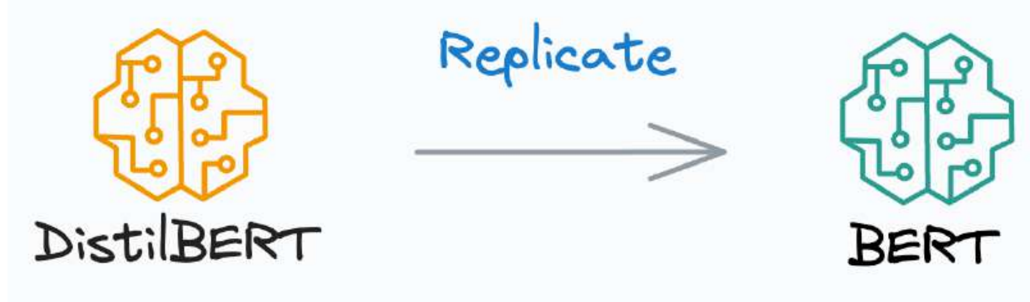
But this changed in the Transformer era, which resulted in contextualized embeddings models powered by Transformers, such as:

- **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.

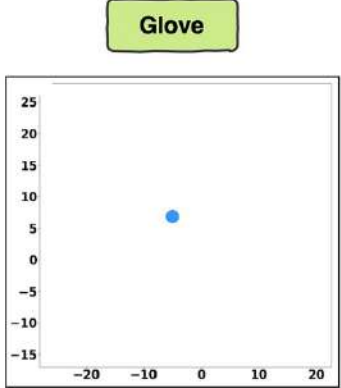
These were capable of generating context-aware representations, thanks to their self-attention mechanism.

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.



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 Priceton’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.

DailyDoseofDS.com

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%
```

Slow KMeans Clustering

KMeans With Data Projecttion

```
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%
```

Similar performance and 10x Faster

As shown, Sparse Random Projection provides:

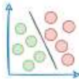
- Similar performance, and
- a MASSIVE run-time improvement of 10x.

This can be especially useful in high-dimensional datasets.

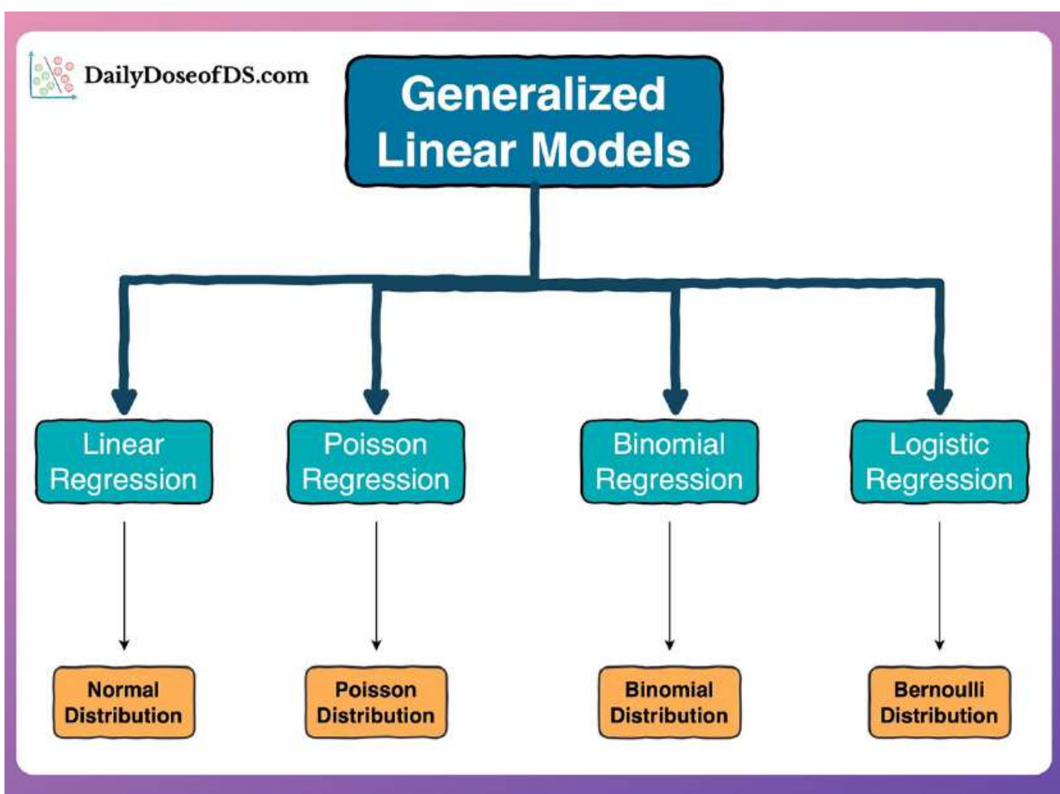
Get started with Sparse Random Projections here: Sklearn Docs.

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.

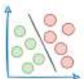
Today, I want to build on that and help you cultivate what I think is one of the MOST overlooked and underappreciated skills in developing linear models.

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

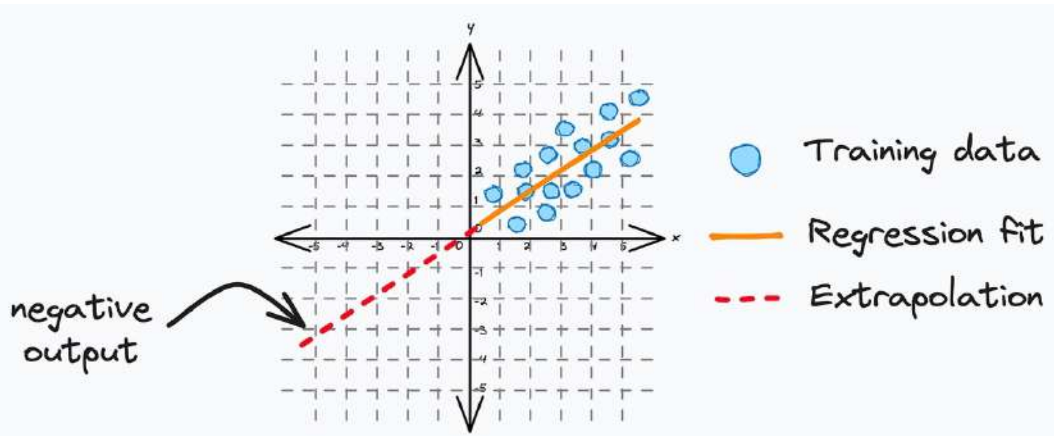
But let's do a quick recap of yesterday's post before we proceed.

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.

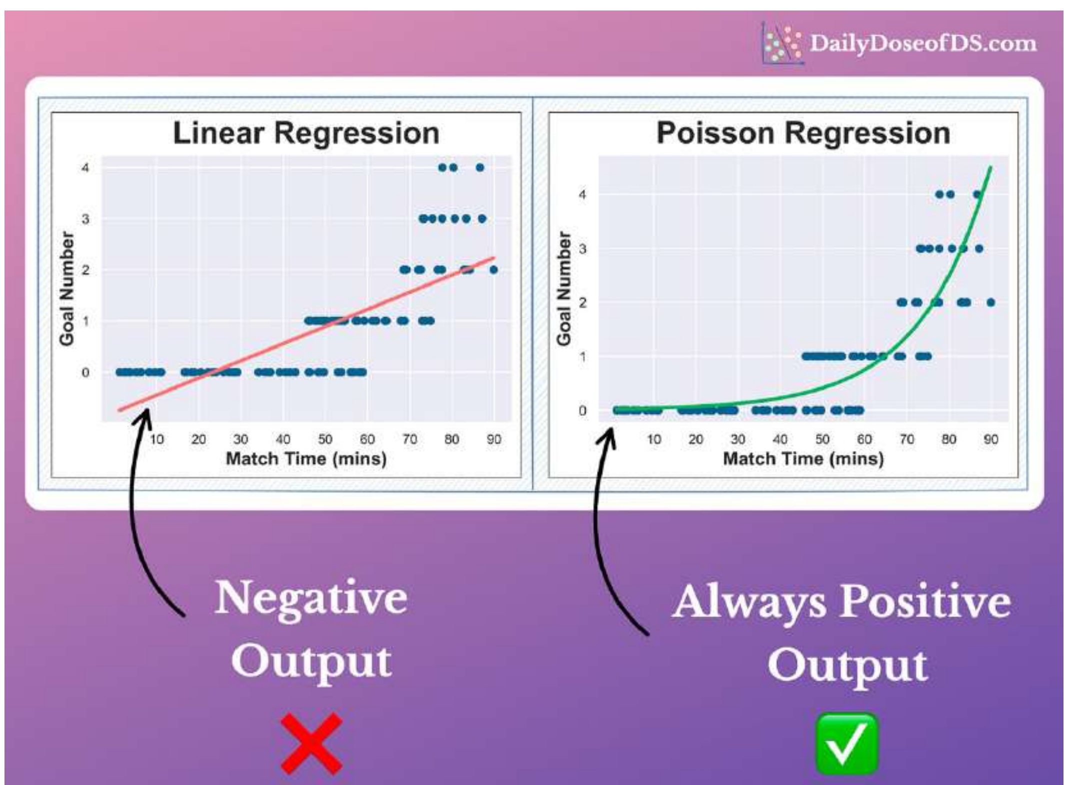
For instance:

- 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:



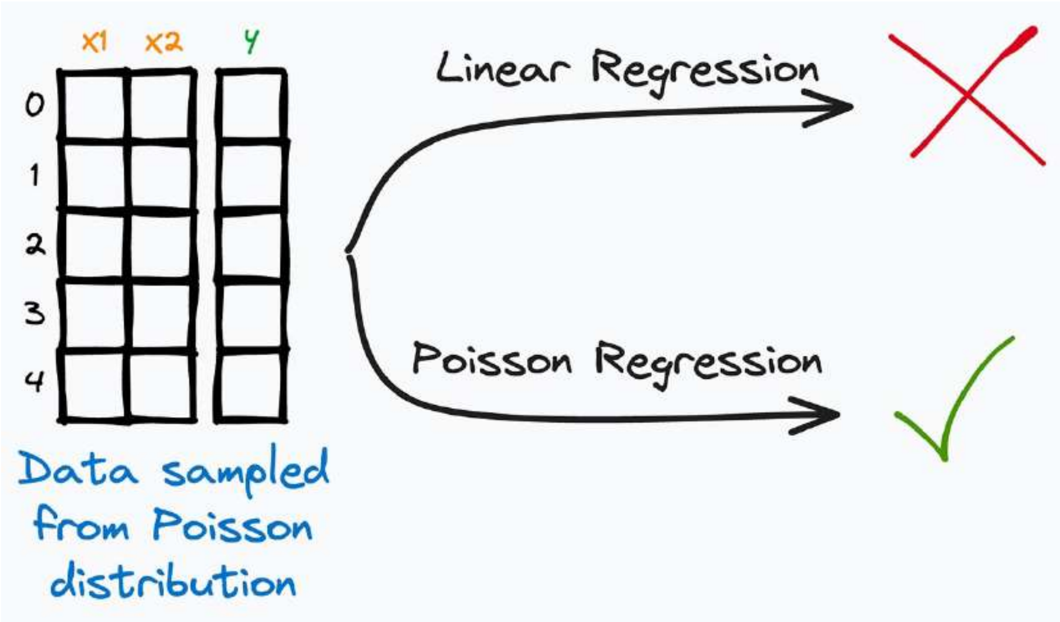
Please read yesterday's post for in-depth info: [Poisson Regression: The Robust Extension of Linear Regression](#).

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.

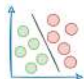


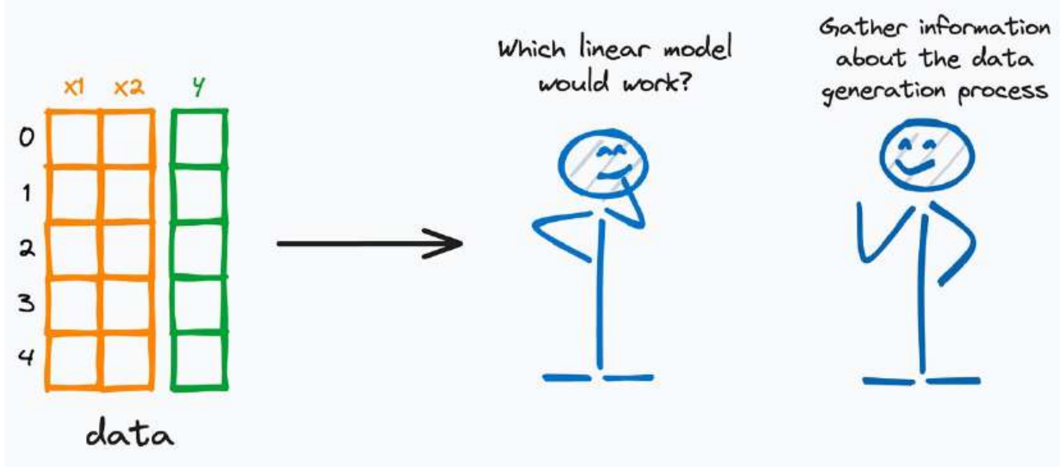
Instead, it came from a Poisson distribution, which is why Poisson regression worked better.

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

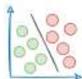
Thus, developing a habit of stepping back and thinking about the data generation process will give you so much clarity in the modeling stages.

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.

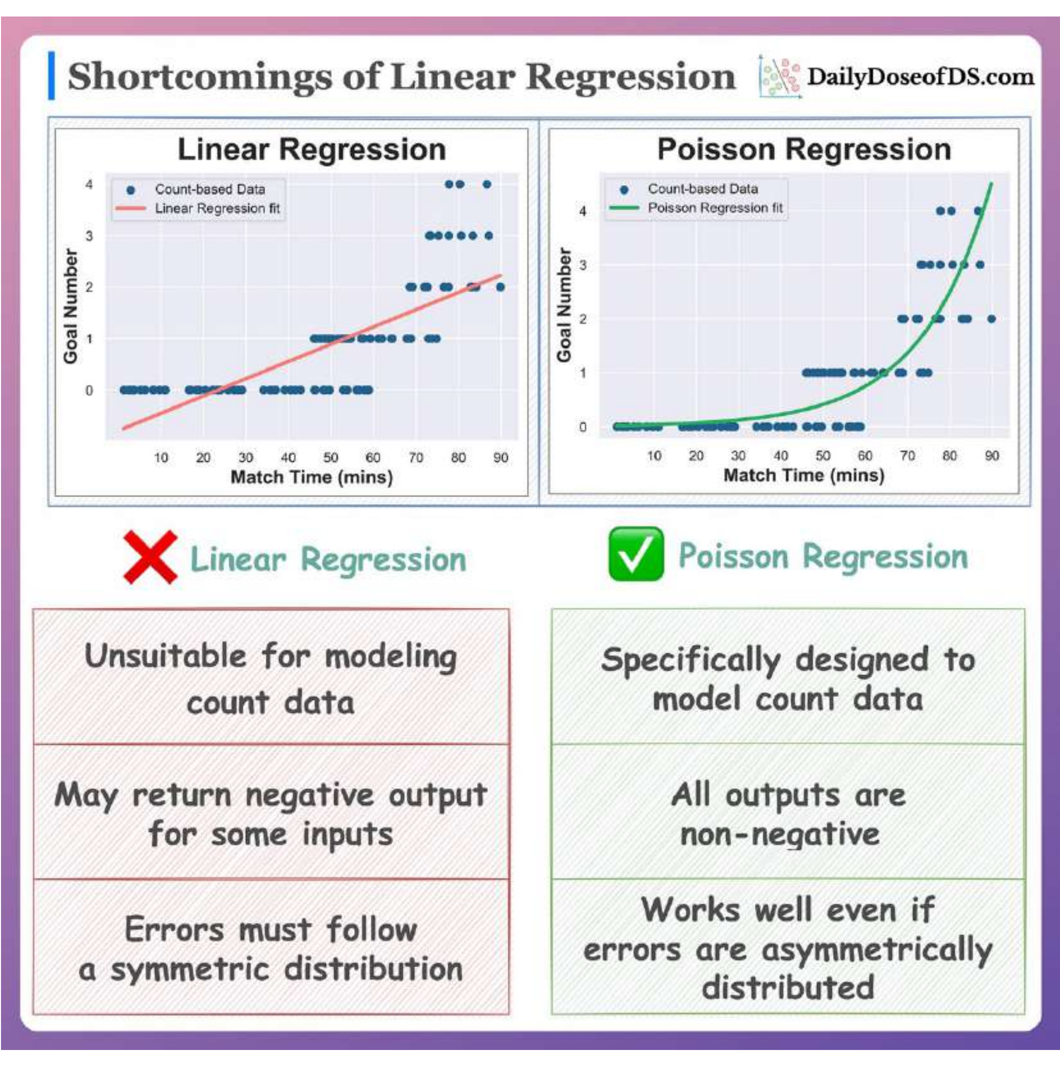
Consequently, you'd know which algorithm to use and, most importantly, **why**.

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.

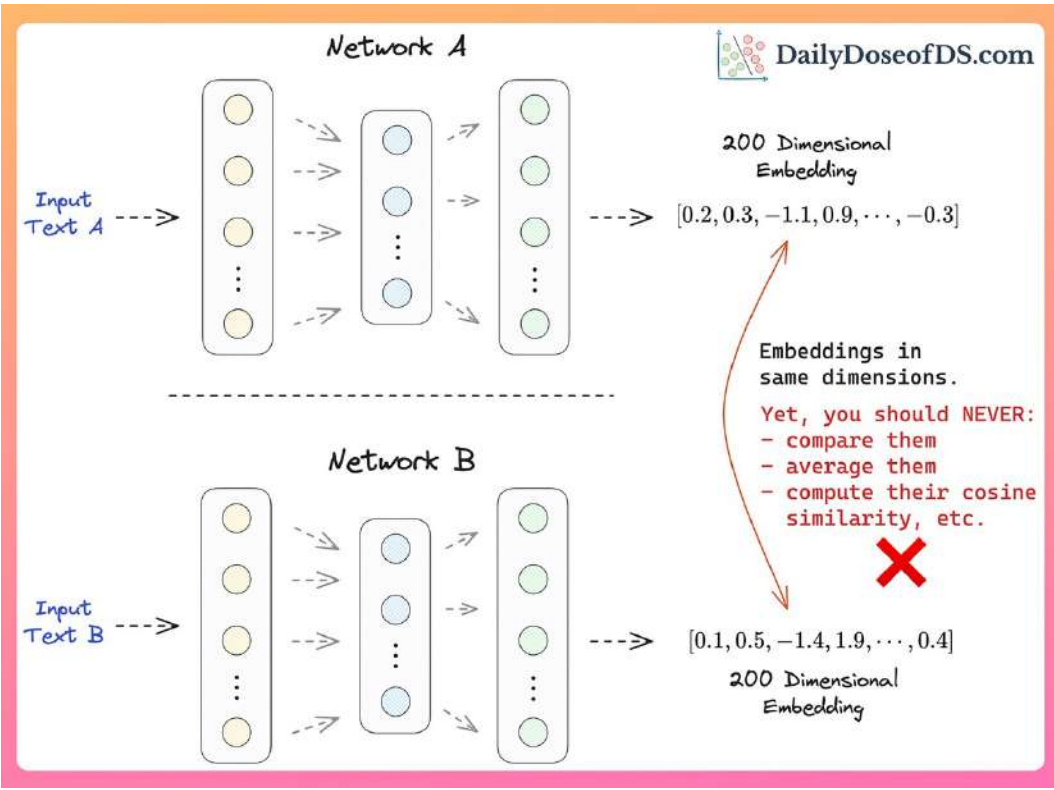


Poisson Regression: The Robust Extension of Linear Regression



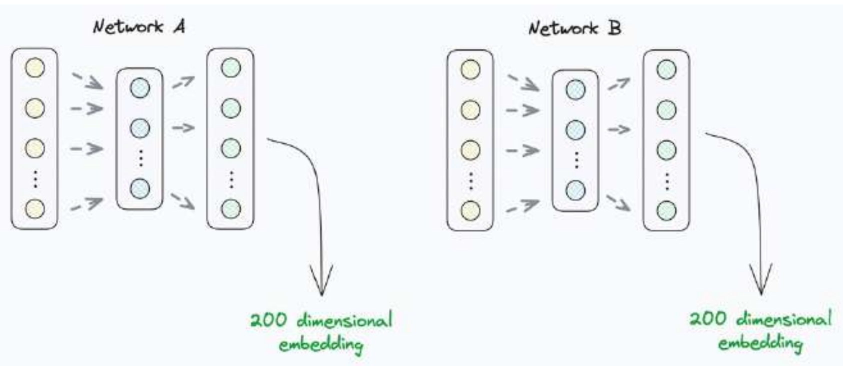
Read the full issue here: https://www.blog.dailydoseofds.com/p/poisson-regression-the-robust-extension

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?

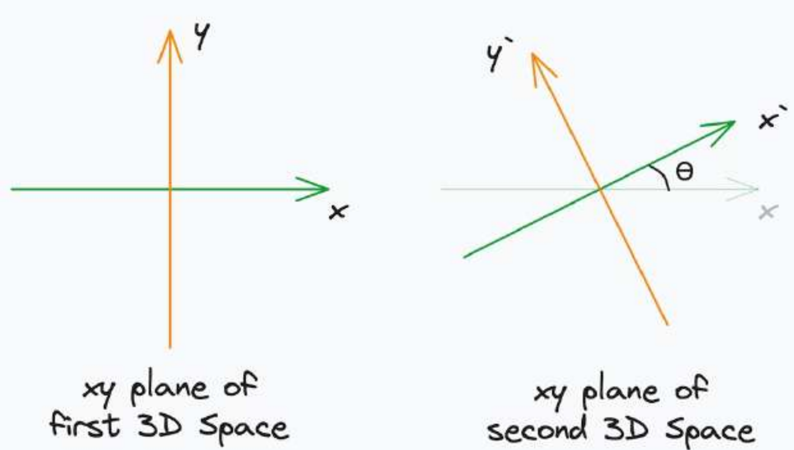
Even though these embeddings have the same length, they are out of space.

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.

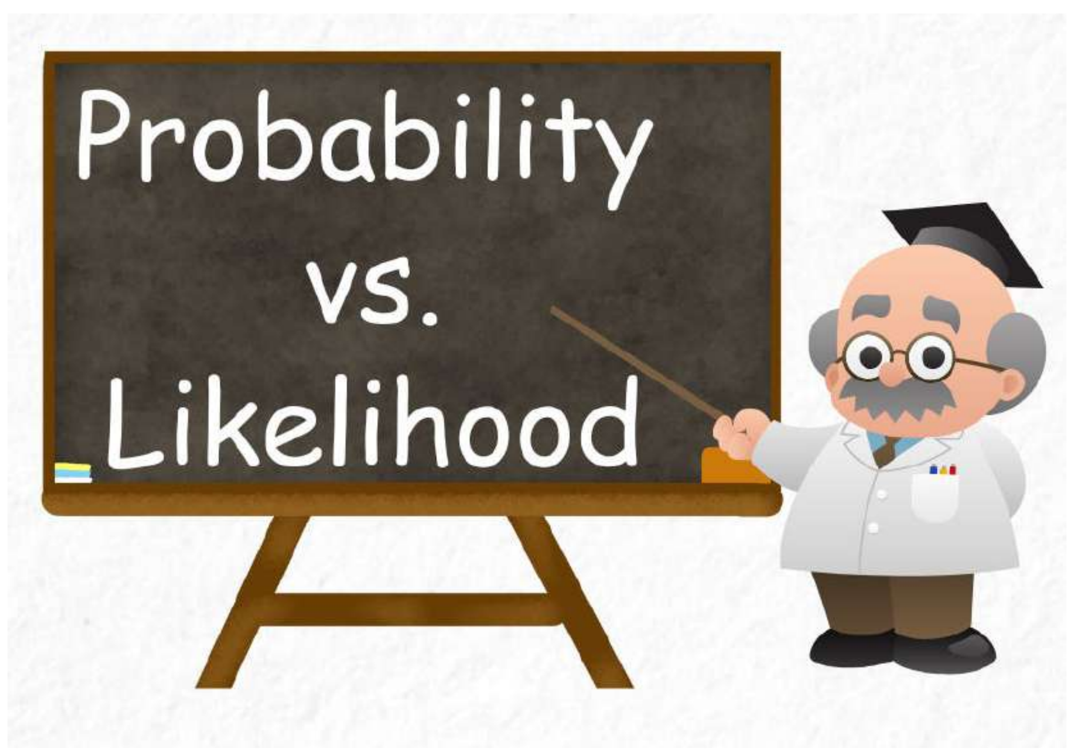
This is a mistake that may cause some serious trouble in your ML pipeline.

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 /ˌprɒb.əˈbɪl.ə.ti/ US /ˌprɑː.bəˈbɪl.ə.t̬i/

Add to word list

C1

the level of possibility of something happening or being true:

- *What is the probability **of** winn**ing**?*
- *The probability **of** gett**ing** 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).*

Anyway.

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

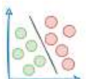
Let's understand!

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

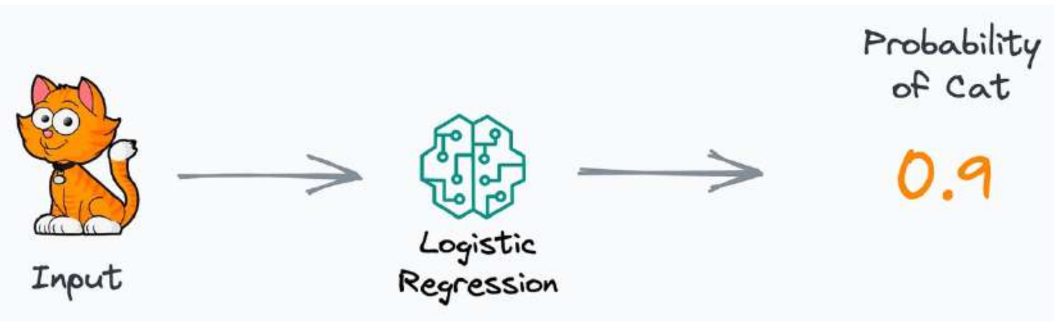
For instance, what is the:

- 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.

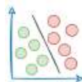
Likelihood, on the other hand, is about explaining events that have already occurred.

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

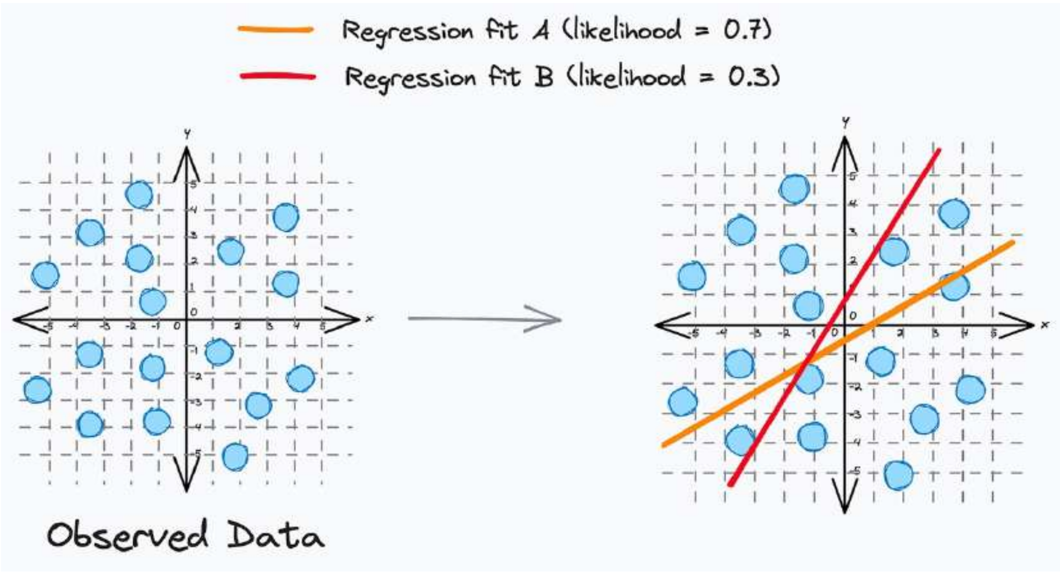
Likelihood helps us determine if we can trust the parameters in a model based on the observed data.

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.

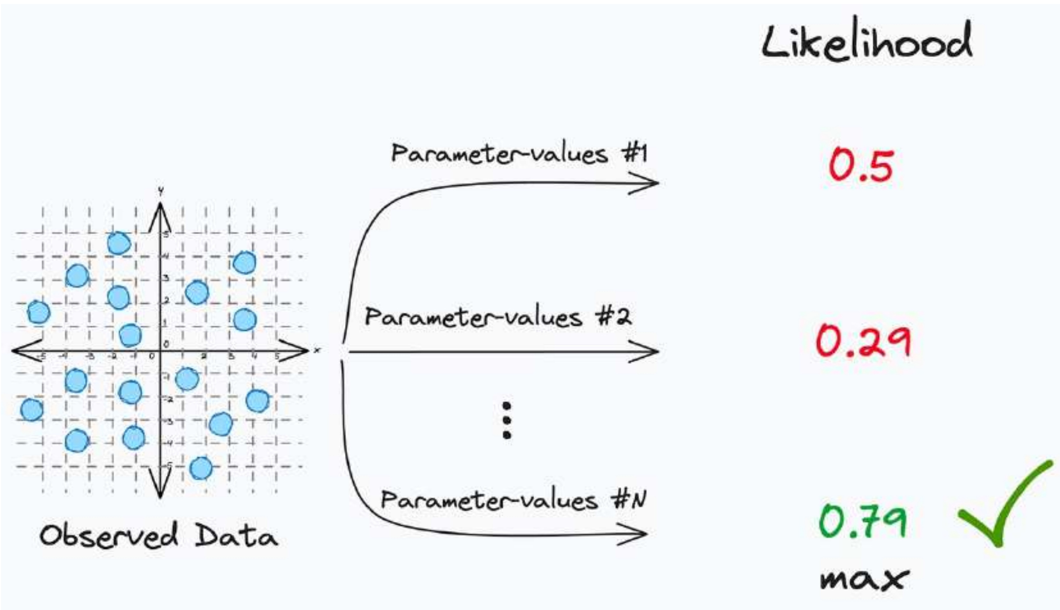


Here, you will ask questions like:

- 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 (θ) 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 "HHHHHHHTHH".
- **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:

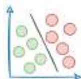
- We determine the possibility of an event.
- We know the parameters associated with the event and assume them to be trustworthy.

In Likelihood:

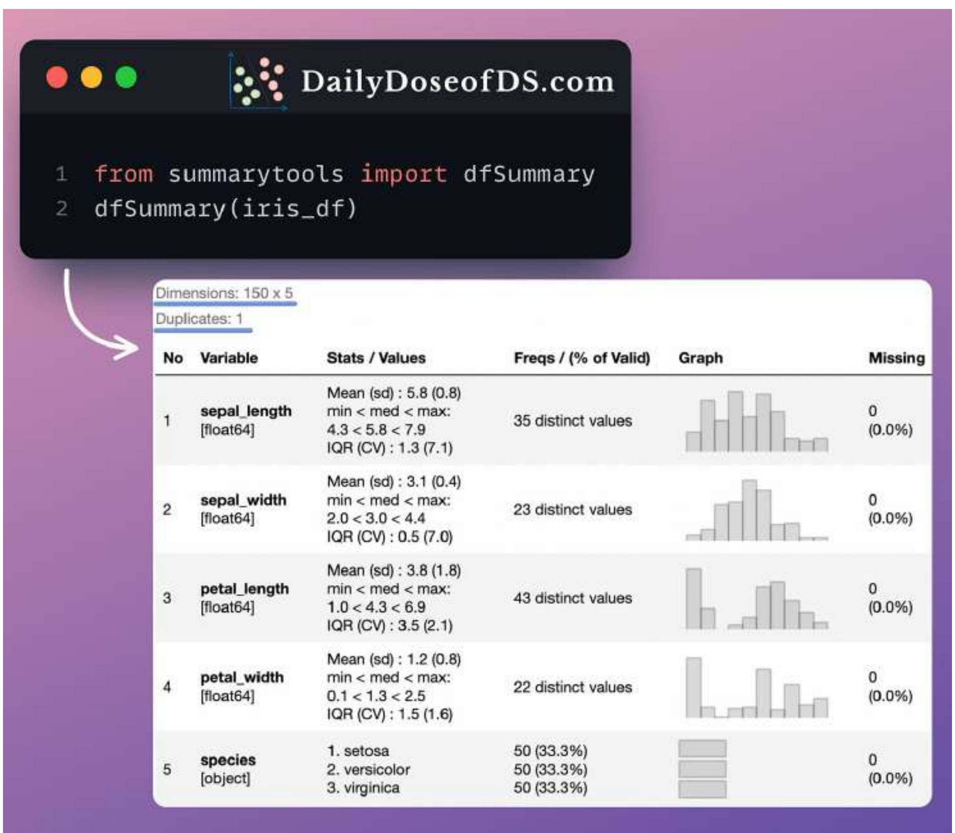
- We have some observations.
- We have an explanation (or parameters).
- Likelihood helps us quantify whether the explanation is trustworthy.

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.



DailyDoseofDS.com

```
from summarytools import dfSummary
dfSummary(iris_df)
```

Dimensions: 150 x 5
Duplicates: 1

No	Variable	Stats / Values	Freqs / (% of Valid)	Graph	Missing
1	**sepal_length** [float64]	Mean (sd) : 5.8 (0.8) min < med < max: 4.3 < 5.8 < 7.9 IQR (CV) : 1.3 (7.1)	35 distinct values		0 (0.0%)
2	**sepal_width** [float64]	Mean (sd) : 3.1 (0.4) min < med < max: 2.0 < 3.0 < 4.4 IQR (CV) : 0.5 (7.0)	23 distinct values		0 (0.0%)
3	**petal_length** [float64]	Mean (sd) : 3.8 (1.8) min < med < max: 1.0 < 4.3 < 6.9 IQR (CV) : 3.5 (2.1)	43 distinct values		0 (0.0%)
4	**petal_width** [float64]	Mean (sd) : 1.2 (0.8) min < med < max: 0.1 < 1.3 < 2.5 IQR (CV) : 1.5 (1.6)	22 distinct values		0 (0.0%)
5	**species** [object]	1. setosa 2. versicolor 3. virginica	50 (33.3%) 50 (33.3%) 50 (33.3%)		0 (0.0%)

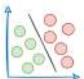
Summarytools is a Jupyter-based tool that provides a standardized and comprehensive data summary.

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

NumPy Array Creation Methods

Method	Description
np.array(<list>)	NumPy array from Python list
np.array(<list-of-lists>)	NumPy array from list of lists
np.array(<pandas-series>)	NumPy array from PD Series
df.values	NumPy array from DataFrame
np.zeros(<size>)	NumPy array of all zeros
np.ones(<size>)	NumPy array of all ones
np.eye(<size>)	Identity NumPy array
np.arange(<start>,<stop>,<step>)	Equally spaced NumPy array with specific step
np.linspace(<start>,<stop>,<count>)	Equally spaced NumPy array with specific size
np.random.randint(<low>,<high>,<size>)	NumPy array of random ints
np.random.random(<size>)	NumPy array of random floats

NumPy Array Manipulation Methods

Method	Description
array.reshape(<new-shape>)	Reshape NumPy Array
array.transpose() OR array.T	Transpose NumPy Array
np.concatenate(<np-arrays>, <axis>)	Concatenate NumPy Arrays
np.flatten(<Nd-np-array>)	Flatten a NumPy Array
np.unique(<np-array>, <axis>)	Find unique elements
array.tolist()	NumPy Array to List

Search Methods

Method	Description
np.argmax(<np-array>, <axis>)	Max Element Index
np.argmin(<np-array>, <axis>)	Min Element Index
np.where(<condition>,<true-return-value>, <false-return-value>)	Conditional Search and Replacement
np.nonzero(<np-array>)	Index of non-zero elements

Mathematical Operations

Method	Description
np.sin(<np-array>)	Trigonometric Functions
np.cos(<np-array>)	
np.tan(<np-array>)	
np.floor(<np-array>)	Element-wise floor value
np.ceil(<np-array>)	Element-wise ceiling value
np.rint(<np-array>)	Round to nearest int
np.round_(<np-array>, <decimal-places>)	Round to decimal places
np.exp(<np-array>)	Element-wise exponent
np.log(<np-array>)	Element-wise logarithm
np.sqrt(<np-array>)	Element-wise square root
np.sum(<np-array>, <axis>)	Sum along an axis
np.mean(<np-array>, <axis>)	Mean along an axis
np.std(<np-array>, <axis>)	Std. dev along an axis

Matrix and Vector Operations

Method	Description
np.dot(<np-array1, np-array2>)	Dot Product
np.matmul(<np-array1, np-array2>)	Matrix Multiplication
np-array1 @ np-array2	
np.linalg.norm(<np-array>)	Vector Norm

Sorting Methods

Method	Description
np.sort(<np-array>, <axis>)	Sort Array
np.argsort(<np-array>, <axis>)	Return the order of indices that sort the array

DailyDoseofDS.com

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:

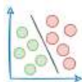
20% of your inputs contribute towards generating 80% of your outputs.

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

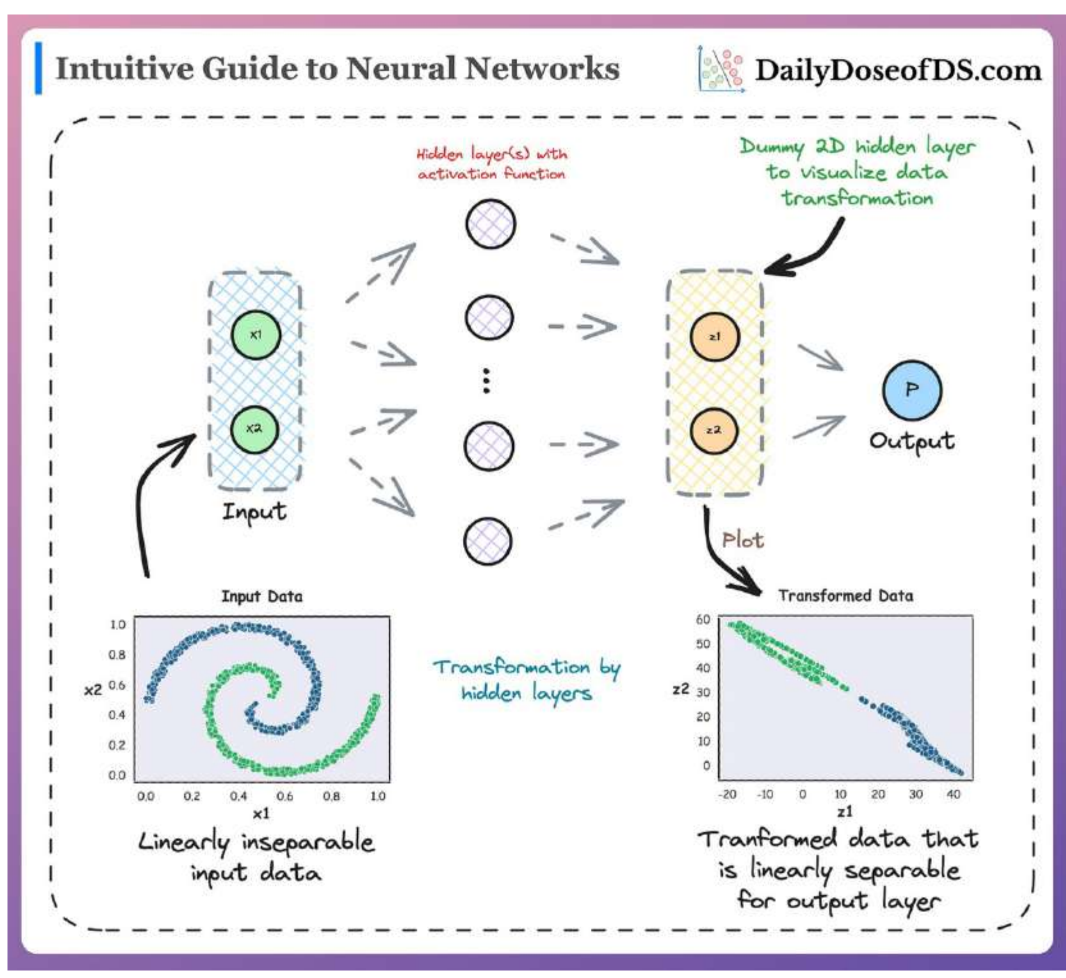
The above visual depicts the 40 most commonly used methods for NumPy.

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:

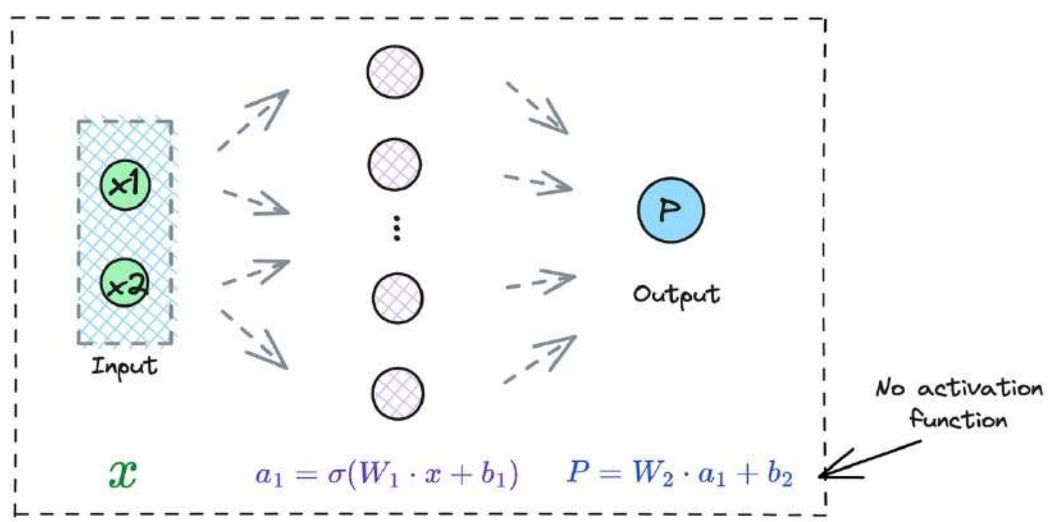
a_l : output activation of current layer

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

b_l : bias $\quad W_l$: weights

σ : activation function

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

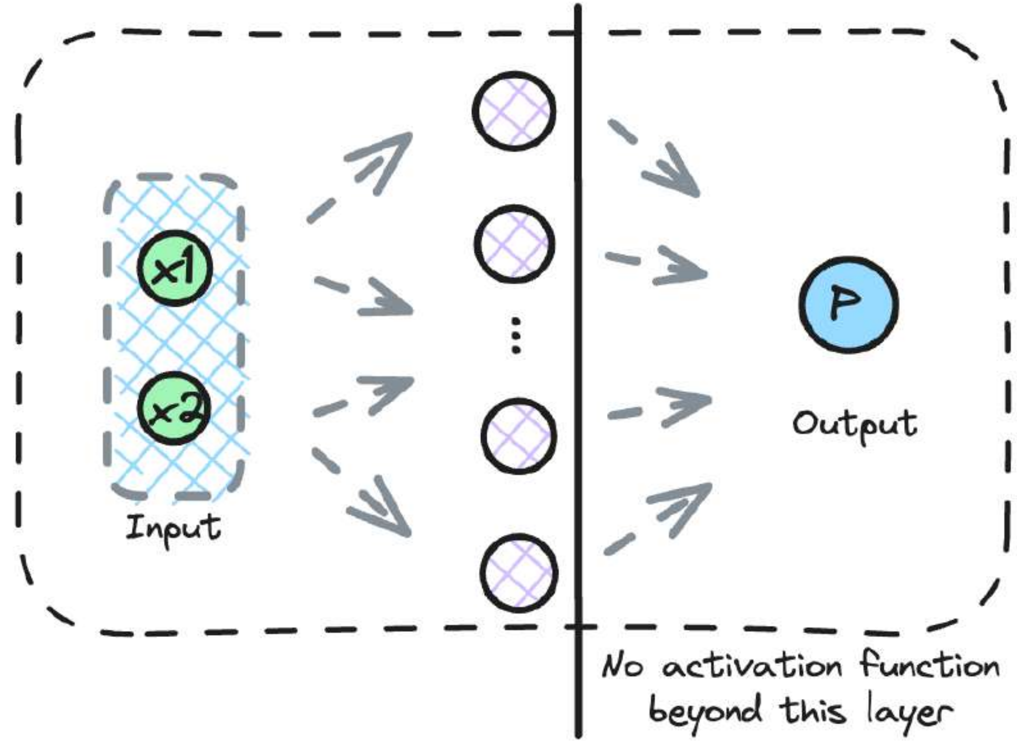


The data is transformed at the hidden layer along with an activation function.

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

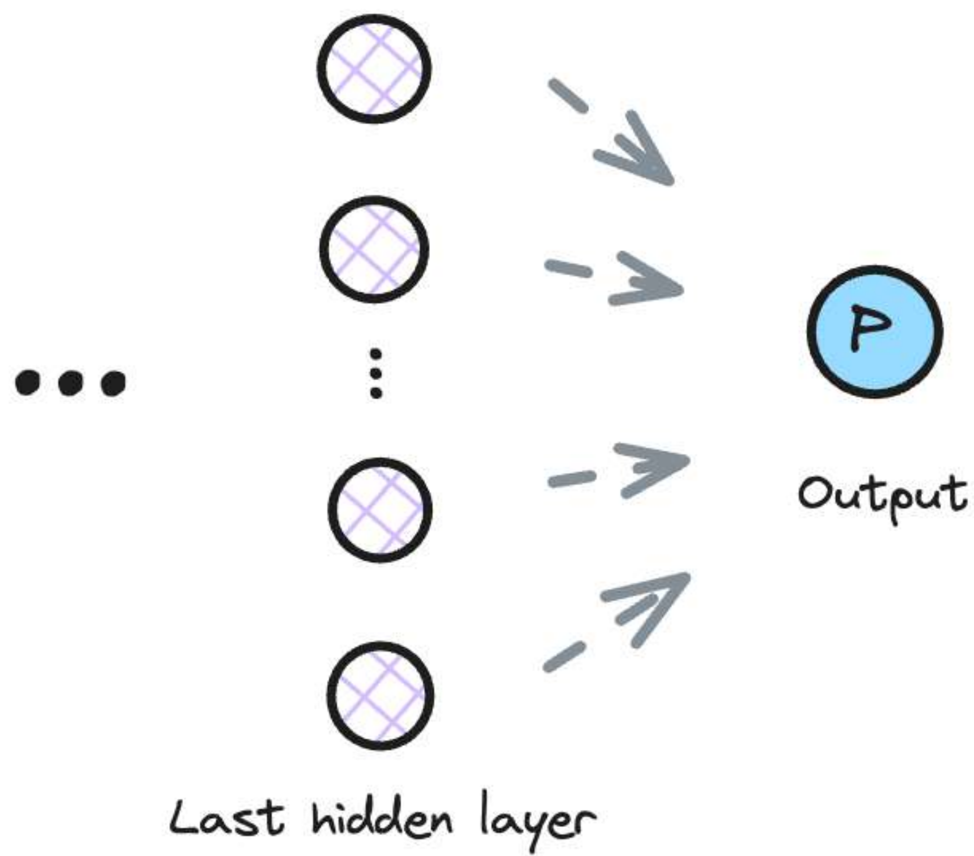
It’s time to notice something here.

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.

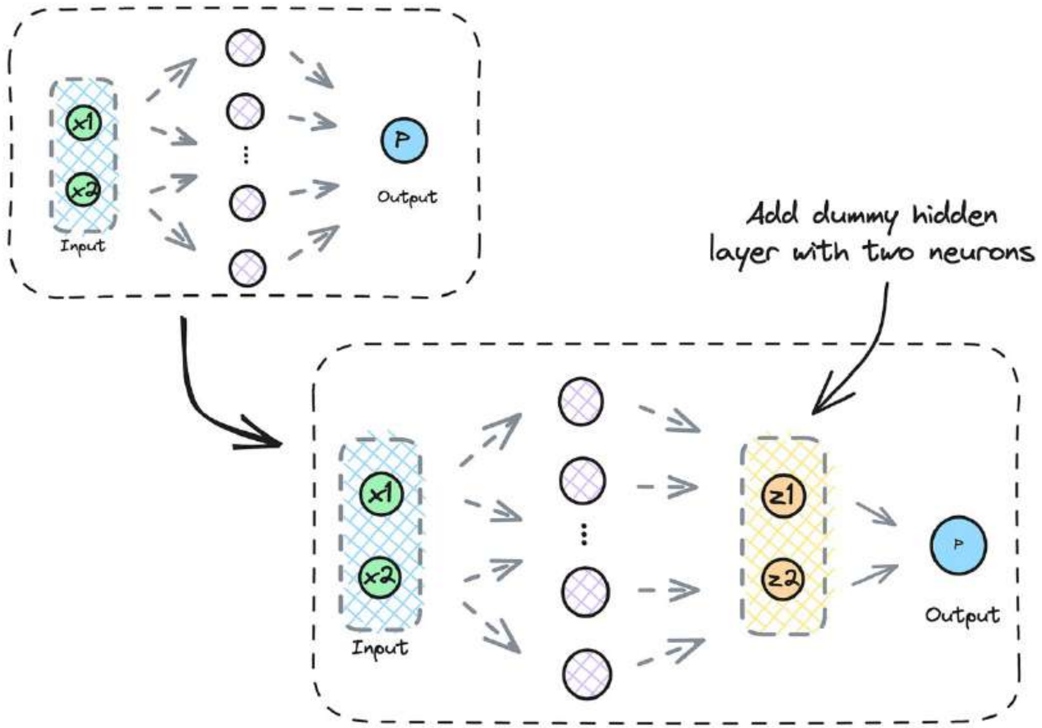
To summarize…

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.



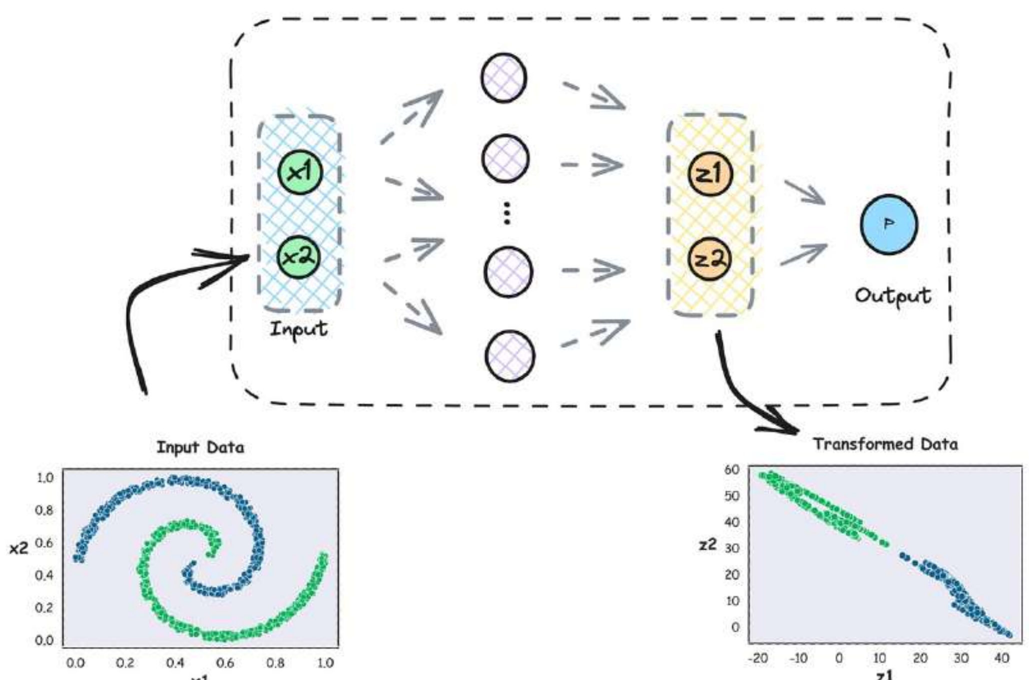
Why two neurons?

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.

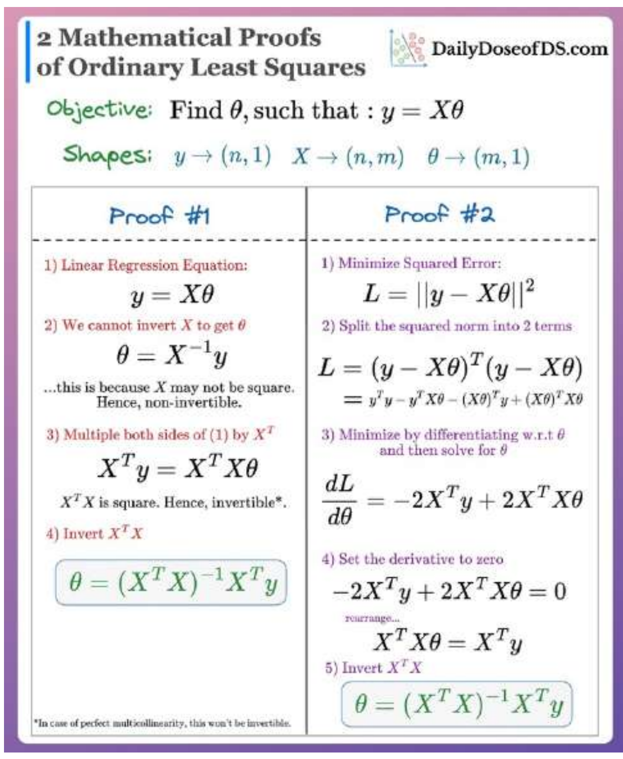
This transformed data can be easily handled by the output classification layer.

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



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

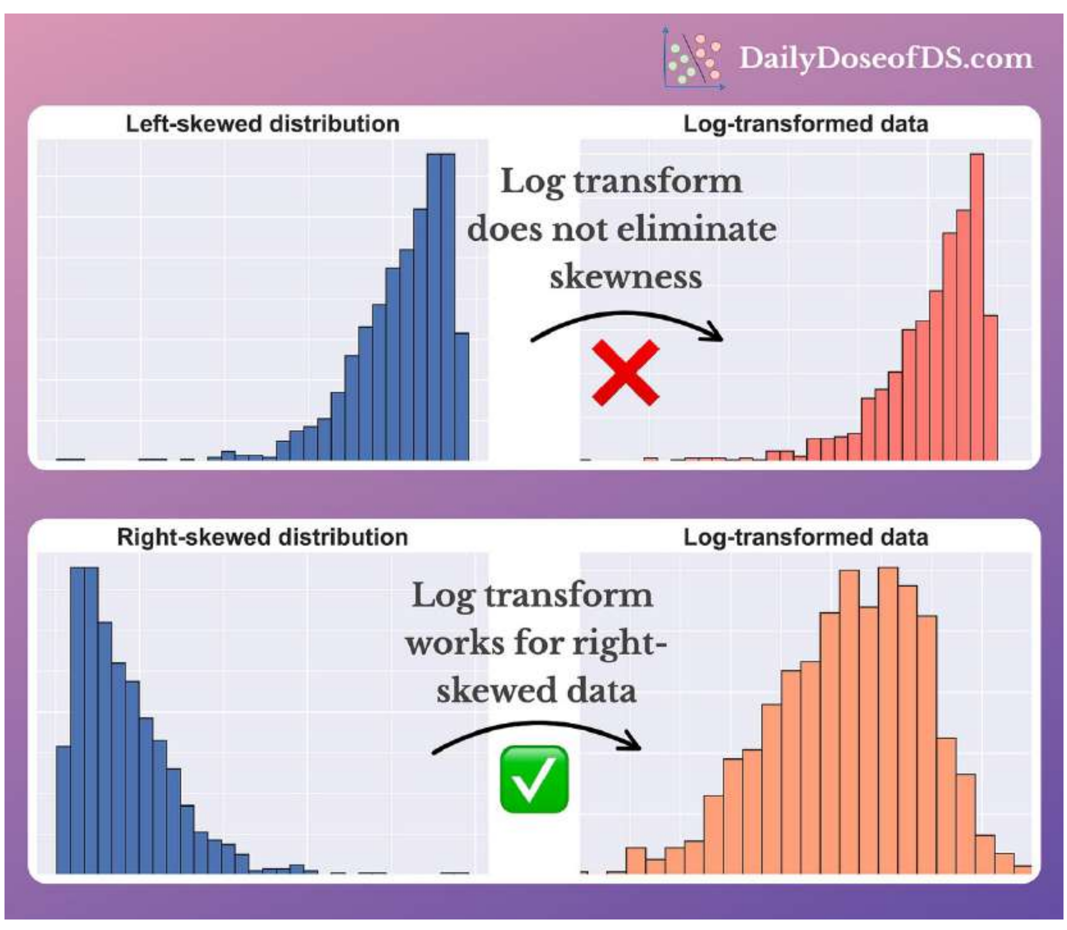
However, in addition to gradient descent, linear regression can model data using another technique called ordinary least squares (OLS).

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.

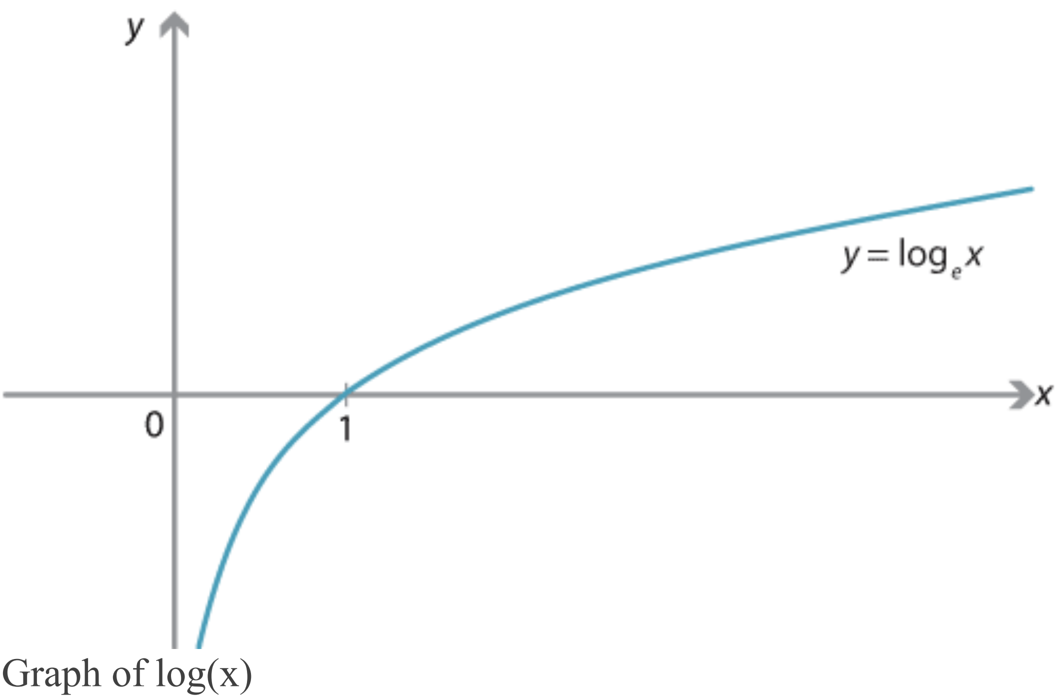
Yet, it is not always the ideal solution for eliminating skewness.

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.

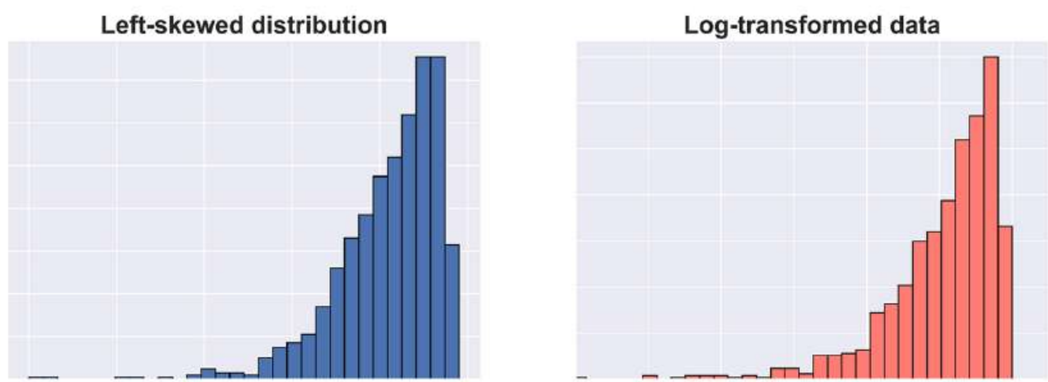
It is because the log function grows faster for lower values. Thus, it stretches out the lower values more than the higher values.



Graph of log(x)

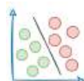
Thus,

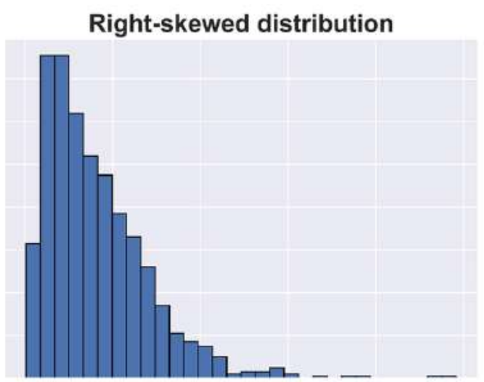
- In case of left-skewness:

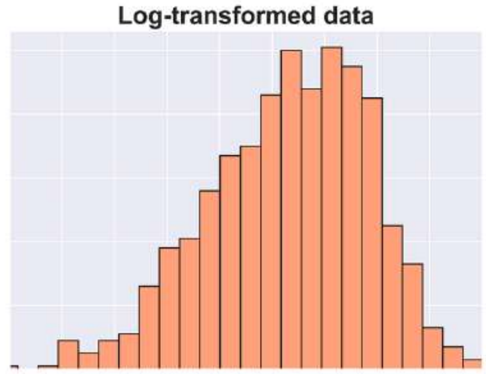


Left-skewness with log transform

  - 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:







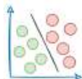
Right-skewness with log transform

- 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.

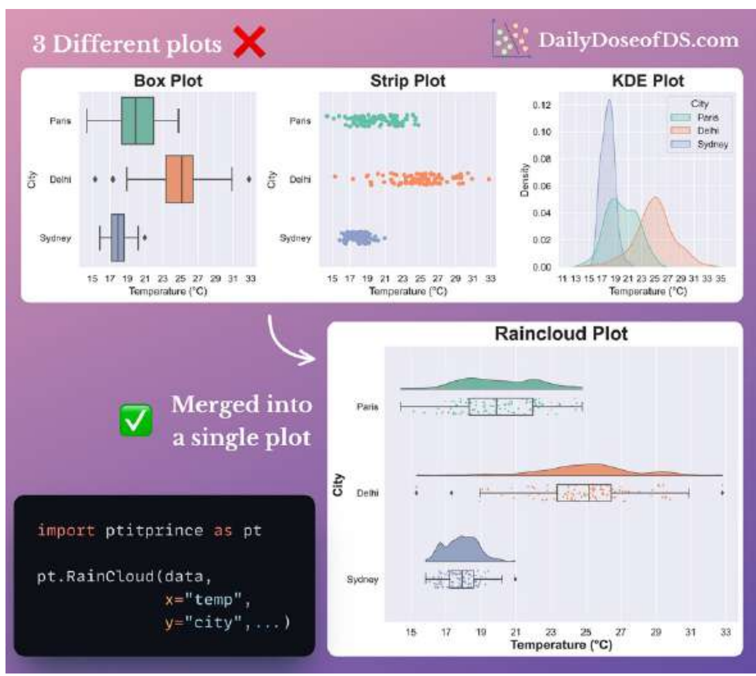
There are a few things you can do:

- 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?



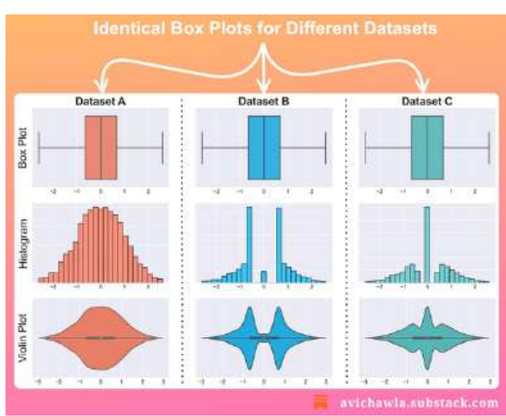
Raincloud Plots: The Hidden Gem of Data Visualisation



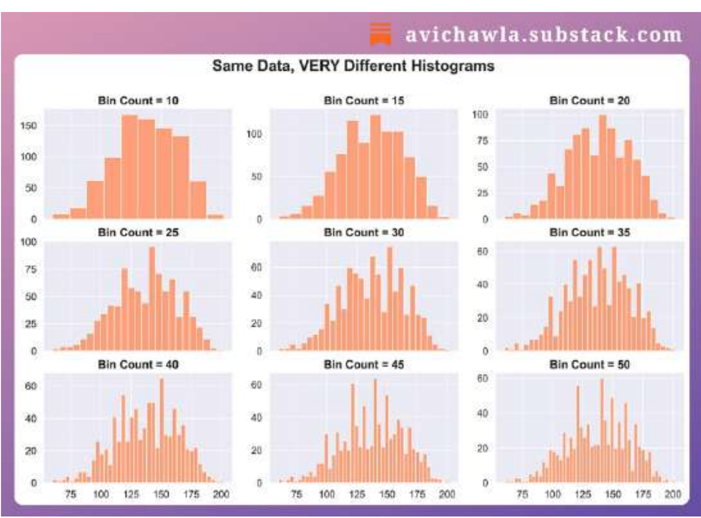
Visualizing data distributions using box plots and histograms can be misleading at times.

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](#).



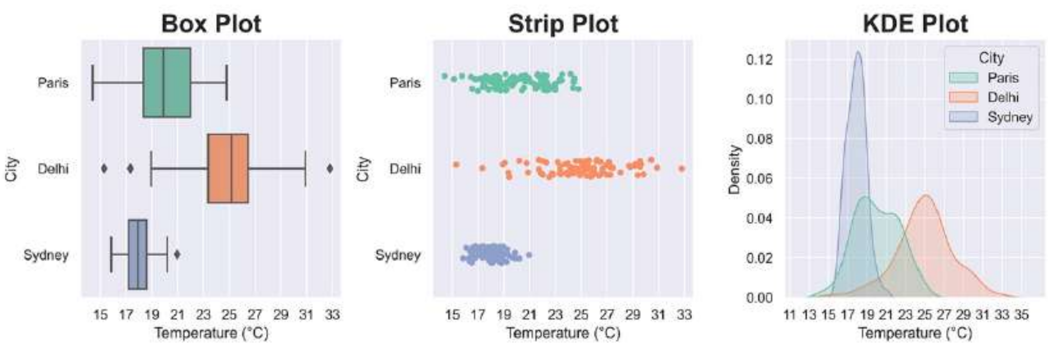
  - We get the same box plot with three different datasets.
- Altering the number of bins changes the shape of a histogram.
  - Read this post here.



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

Paris
Delhi
Sydney
City
15 17 19 21 23 25 27 29 31 33
Temperature (°C)

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:

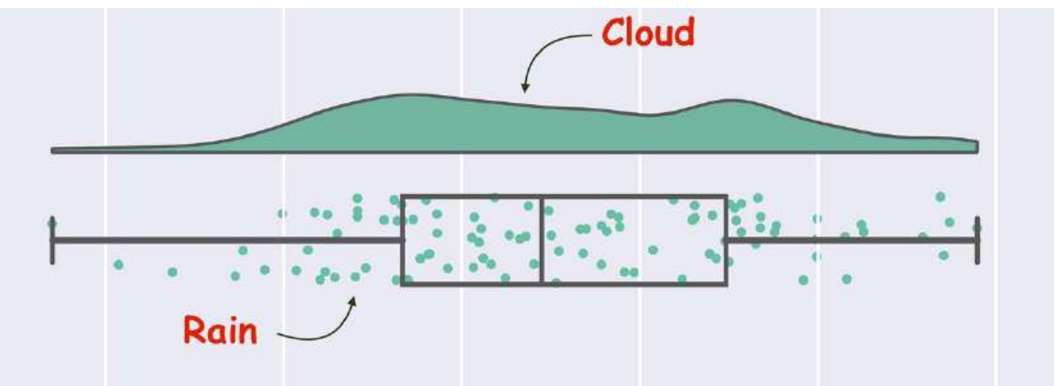
- Combine multiple plots to prevent incorrect/misleading conclusions

- 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.

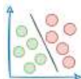
R users can use Raincloud Plots library: GitHub.

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



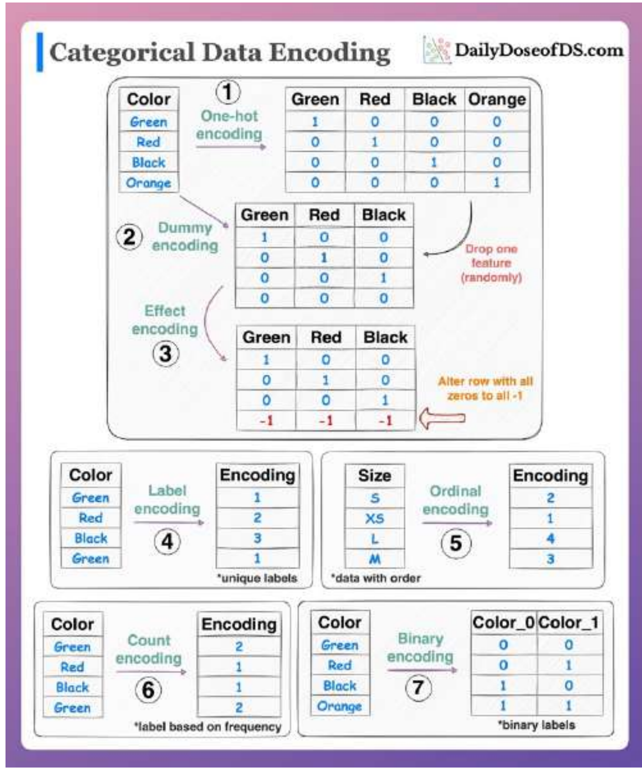
The origin of the name “Raincloud plot”

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



blog.DailyDoseofDS.com

7 Must-know Techniques For Encoding Categorical Feature



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

These primarily include:

a. Categorical

b. Numerical

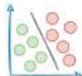
While numerical features can be directly used in most ML models without any additional preprocessing, categorical features require encoding to be represented as numerical values.

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?**

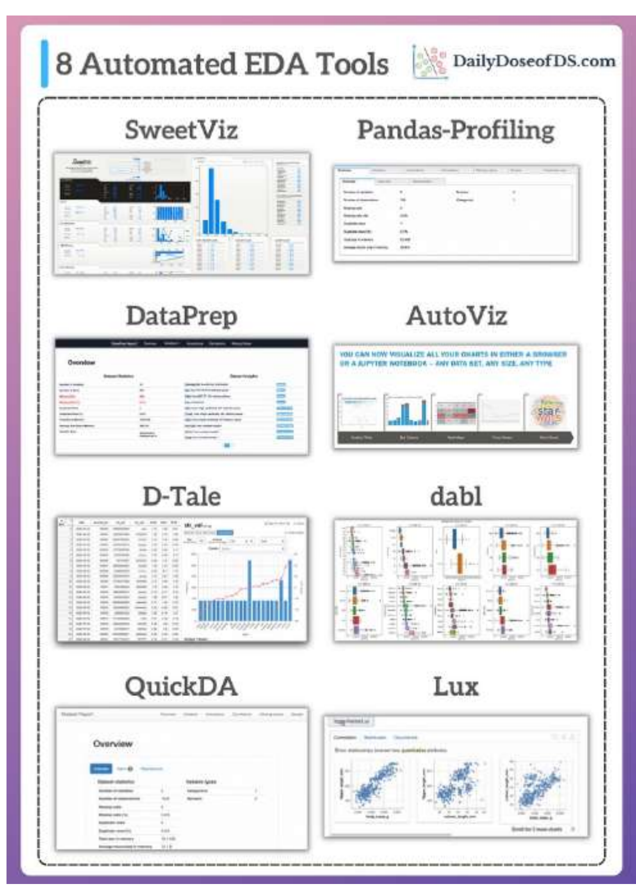
If categorical features do need some additional processing, being aware of the common techniques to encode them is crucial.

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

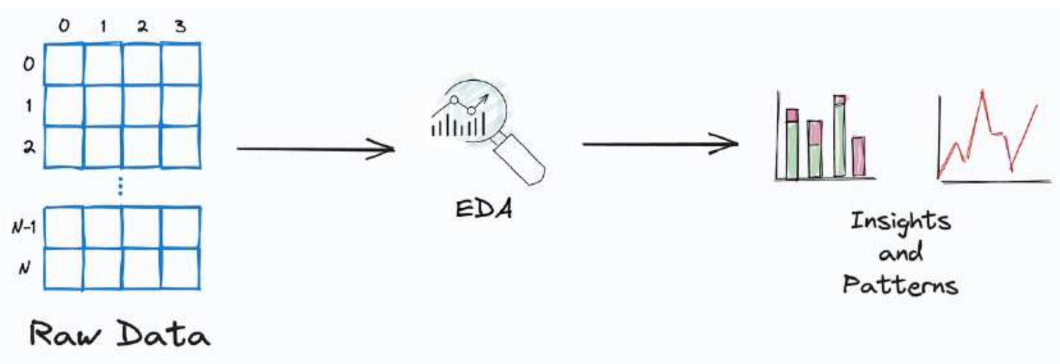


Automated EDA Tools That Let You Avoid Manual EDA Tasks



EDA is a vital step in all data science projects.

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

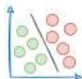


By uncovering hidden insights and patterns, one can make informed decisions about subsequent steps in the project.

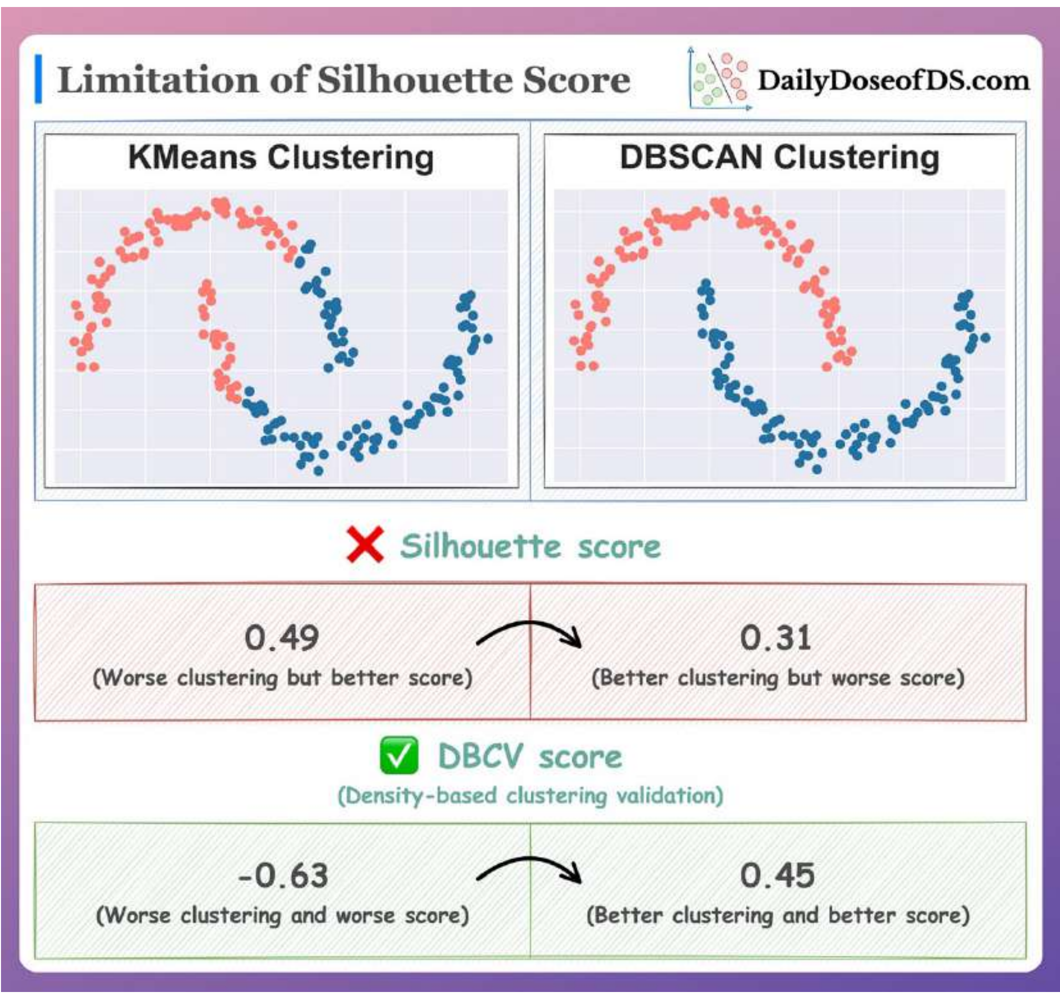
Despite its importance, it is often a time-consuming and tedious task.

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.

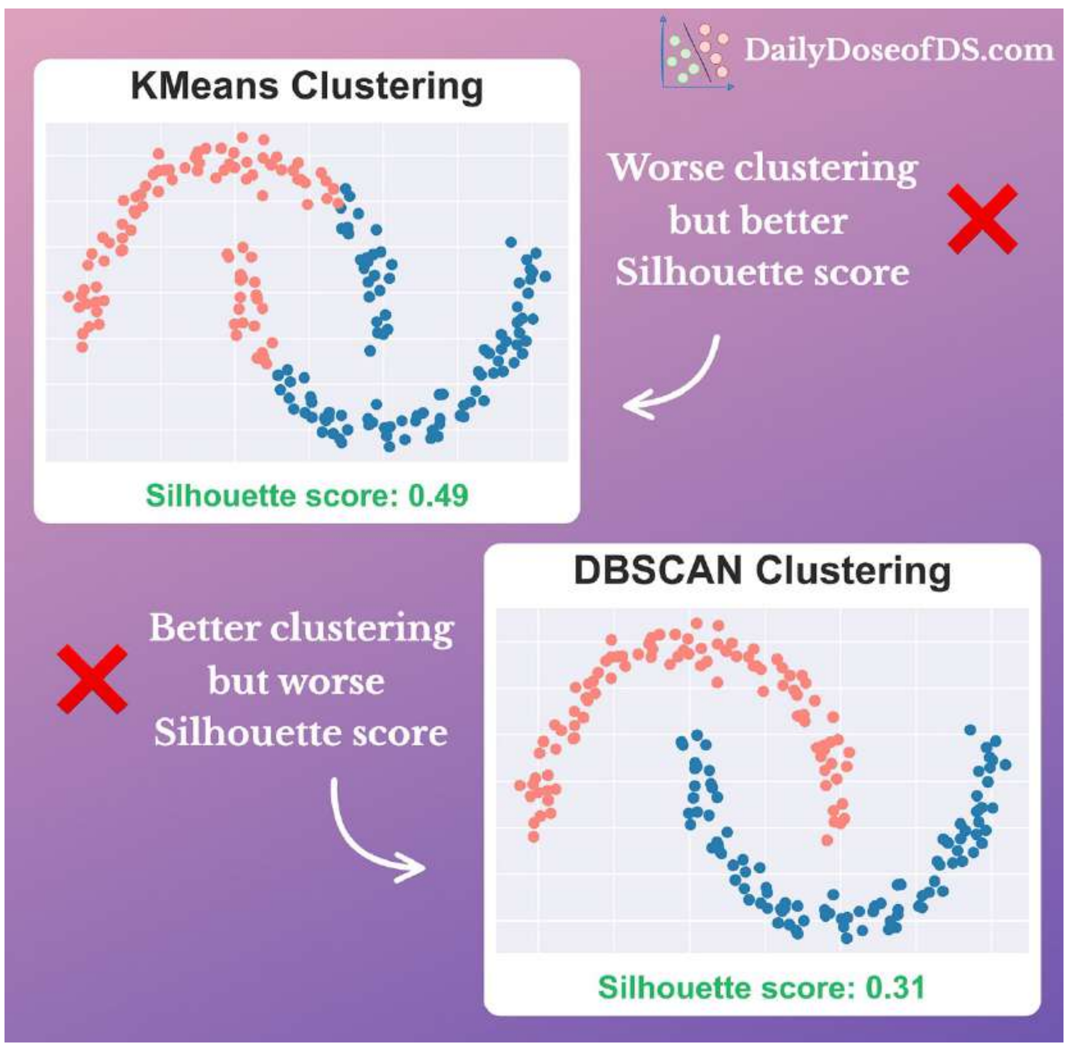
At times, it is also preferred in place of the elbow curve to determine the optimal number of clusters. (I have covered this before if you wish to recap or learn more).

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:

a. The density **within** a cluster

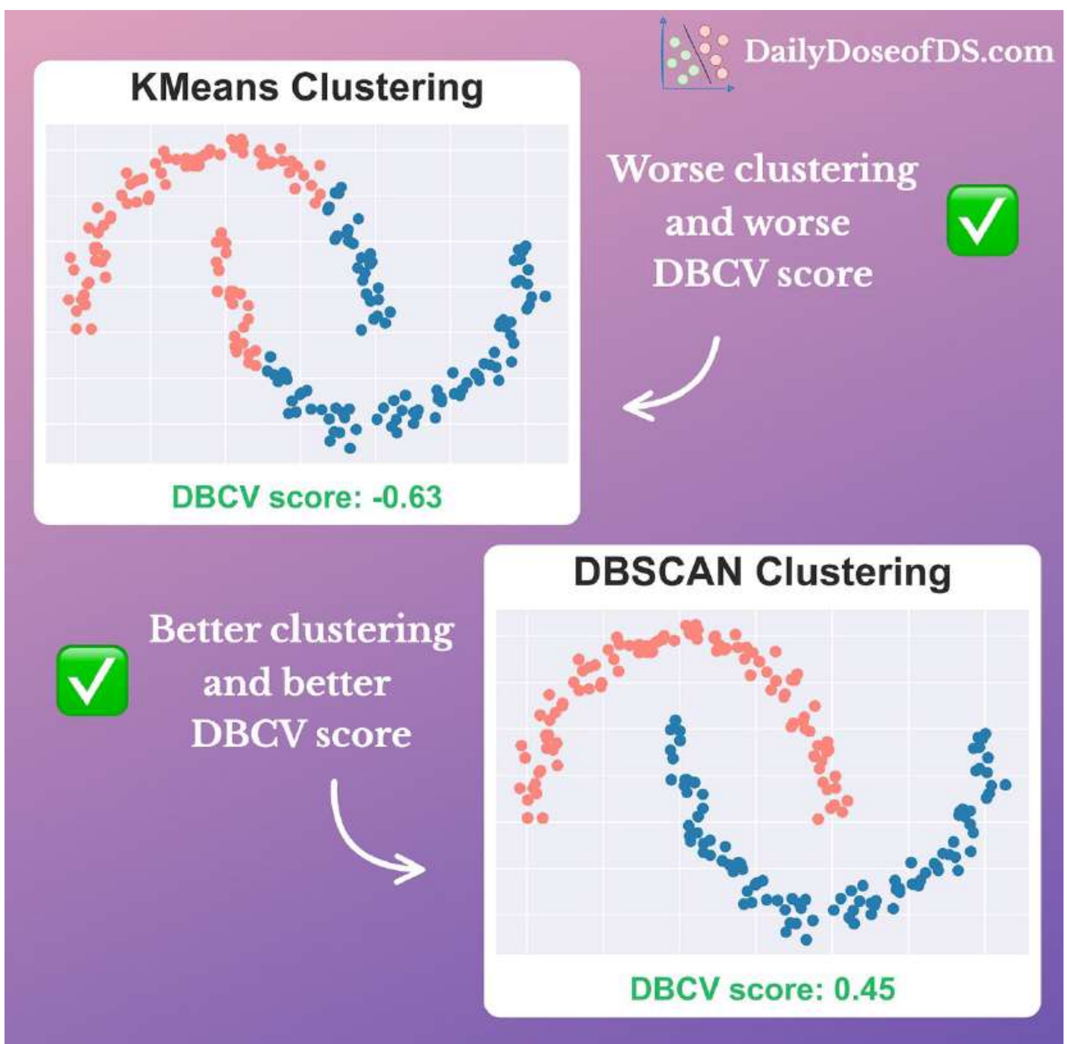
b. The density **between** clusters

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.

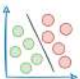
The effectiveness of DBCV is also evident from the image below:



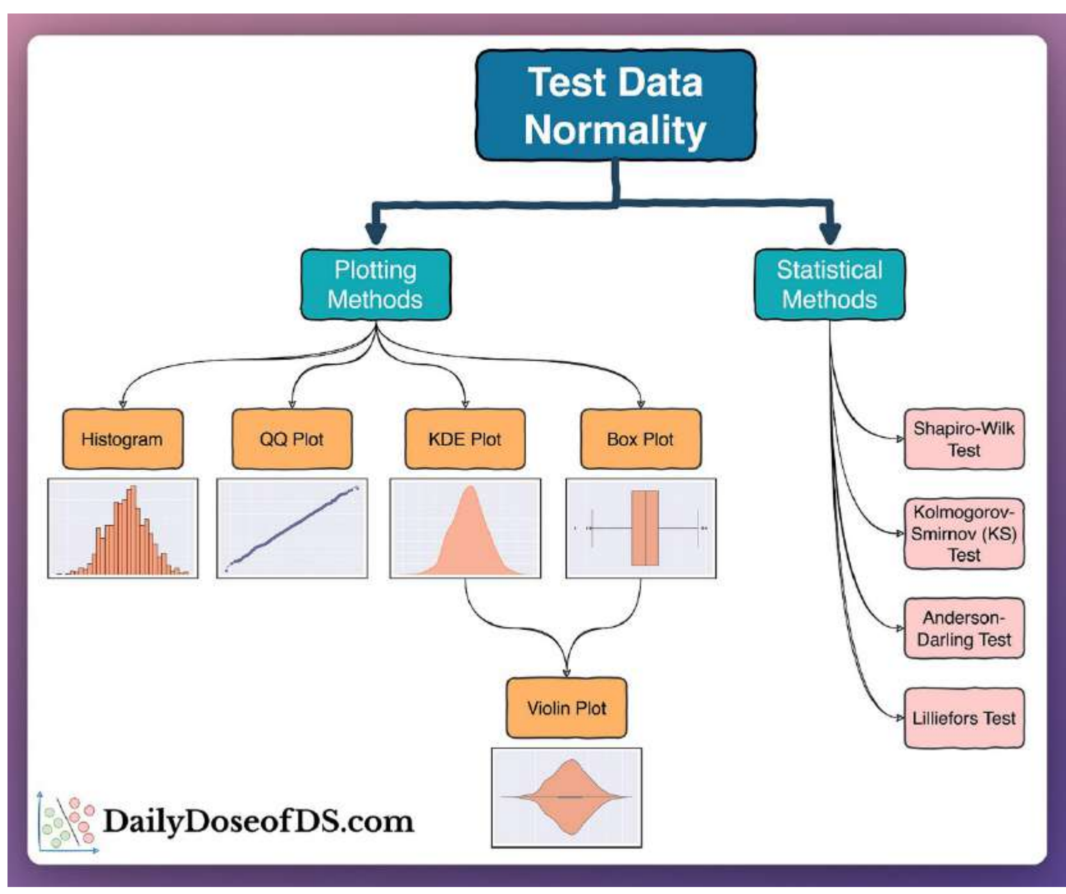
This time, the score for the clustering output of KMeans is worse, and that of density-based clustering is higher.

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.

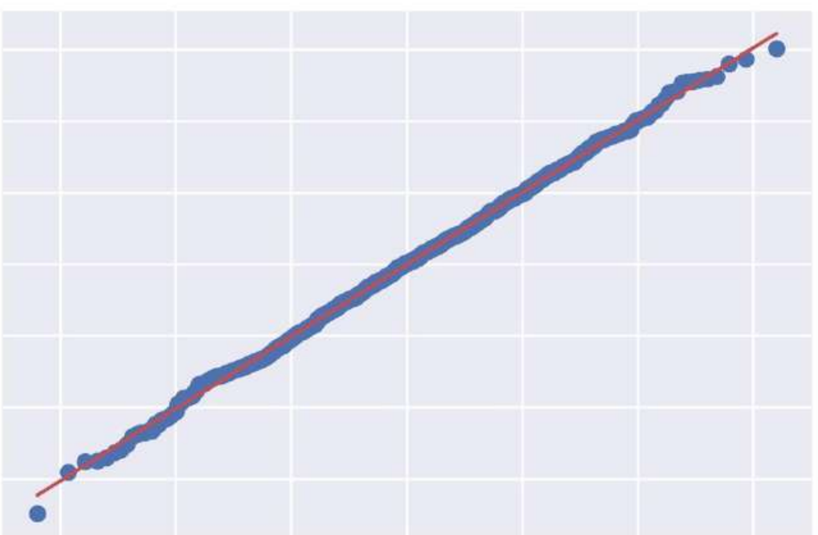
Thus, being aware of the ways to test normality is extremely crucial for data scientists.

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

#1) Plotting methods:

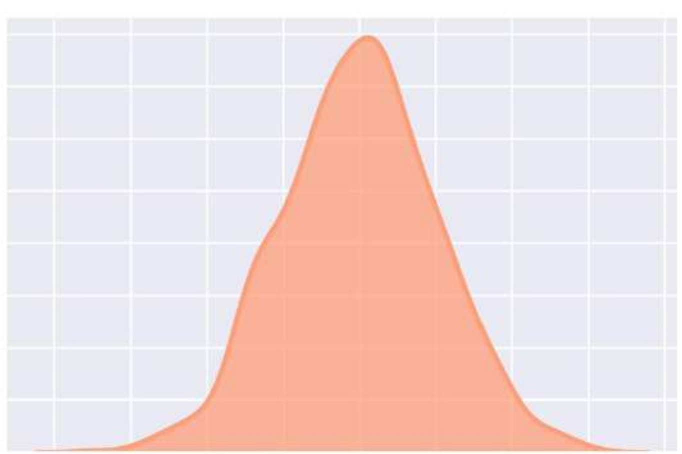
Histogram

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.

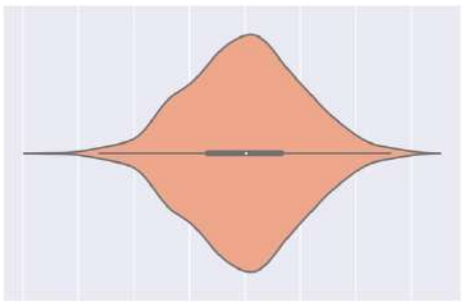
KDE (Kernel Density Estimation) Plot:



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.

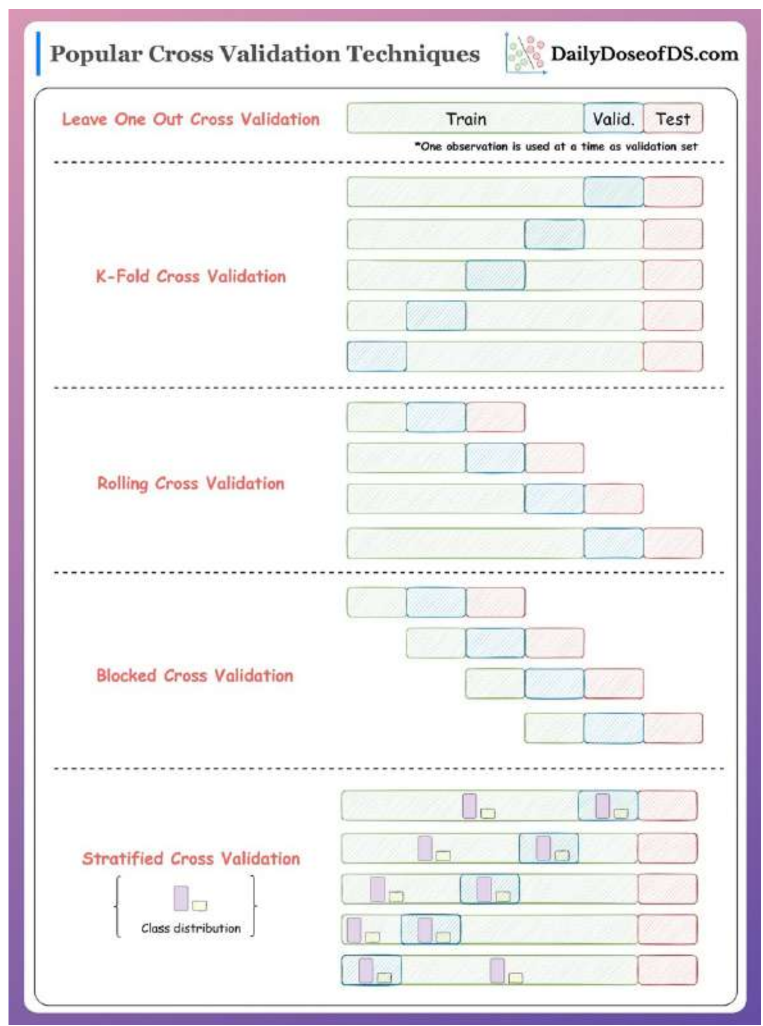
c. However, if the parameters are unknown, Lilliefors is recommended.

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?

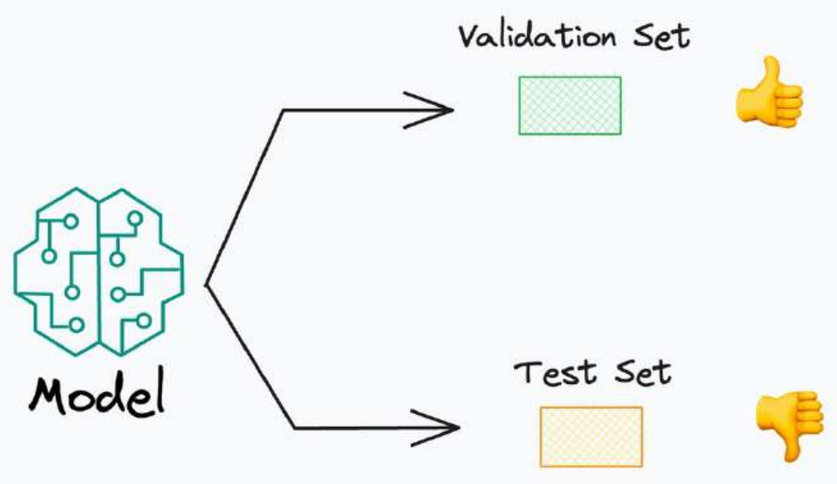
A Visual Guide to Popular Cross Validation Techniques



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.



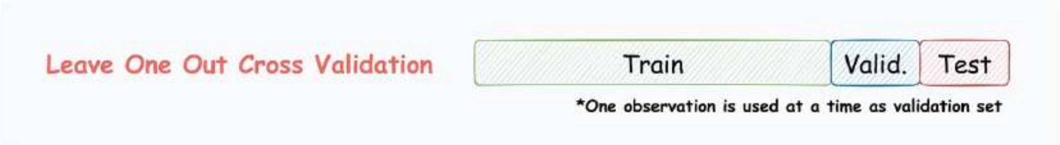
That is why we often use cross-validation instead of simple single-set validation.

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.

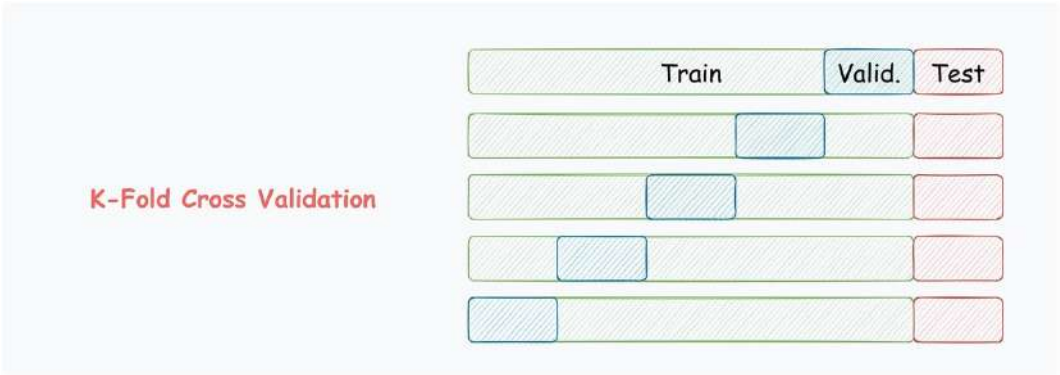
The image above presents a visual summary of five of the most commonly used cross-validation techniques.

Leave-One-Out Cross-Validation



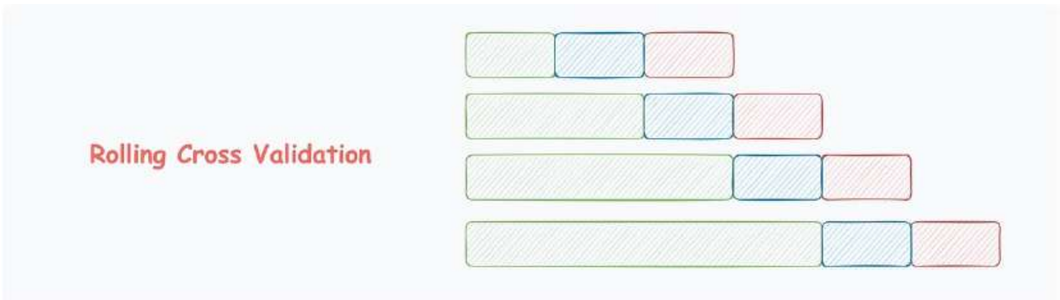
1. Leave one data point for validation.
2. Train the model on the remaining data points.
3. Repeat for all points.
4. This is practically infeasible when you have tons of data points. This is because number of models is equal to number of data points.
5. We can extend this to Leave-p-Out Cross-Validation,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

K-Fold Cross-Validation



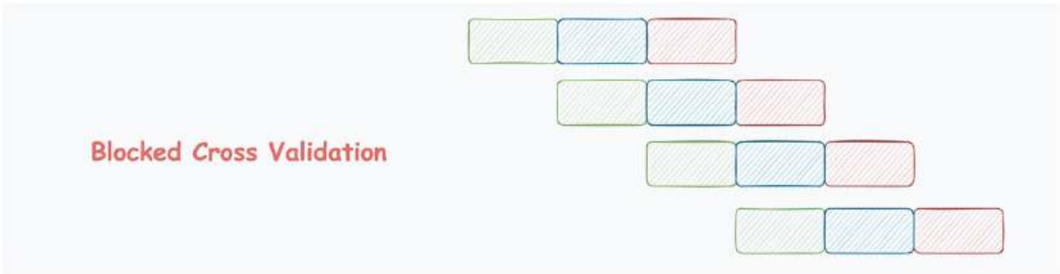
1. Split data into k equally-sized subsets.
2. Select one subset for validation.
3. Train the model on the remaining subsets.
4. Repeat for all subsets.

Rolling Cross-Validation



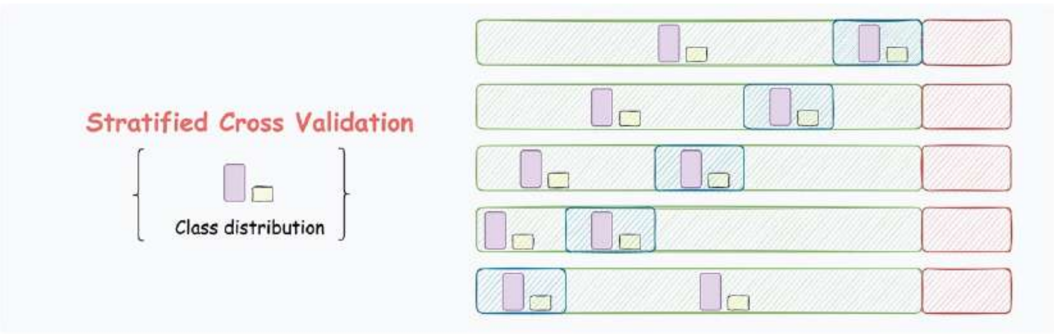
1. Mostly used for data with temporal structure.
2. Data splitting respects the temporal order, using a fixed-size training window.
3. The model is evaluated on the subsequent window.

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.

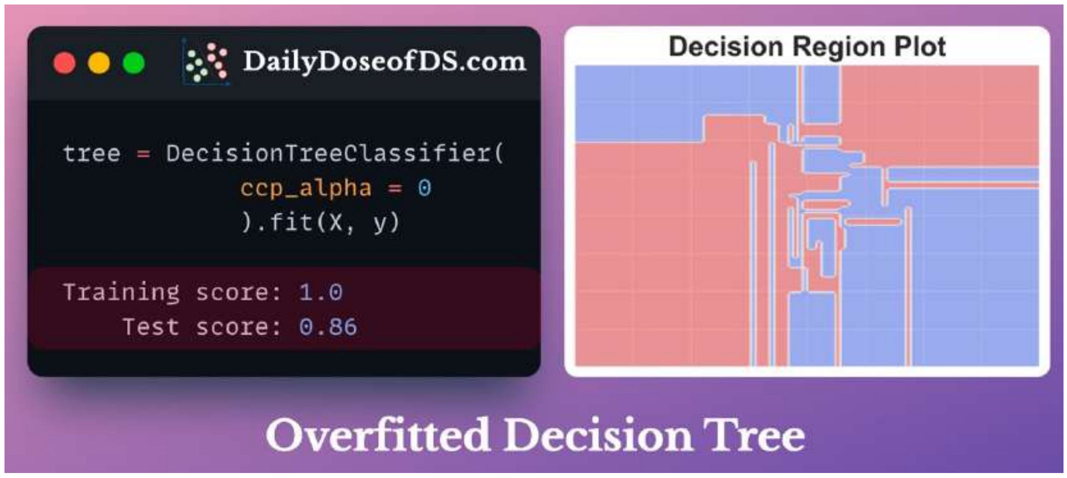
👉 Over to you: What other cross-validation techniques have I missed?

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.

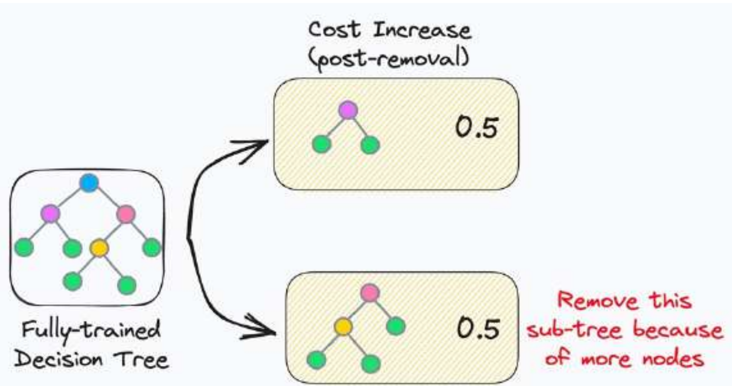
CCP considers a combination of two factors for pruning a decision tree:

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.



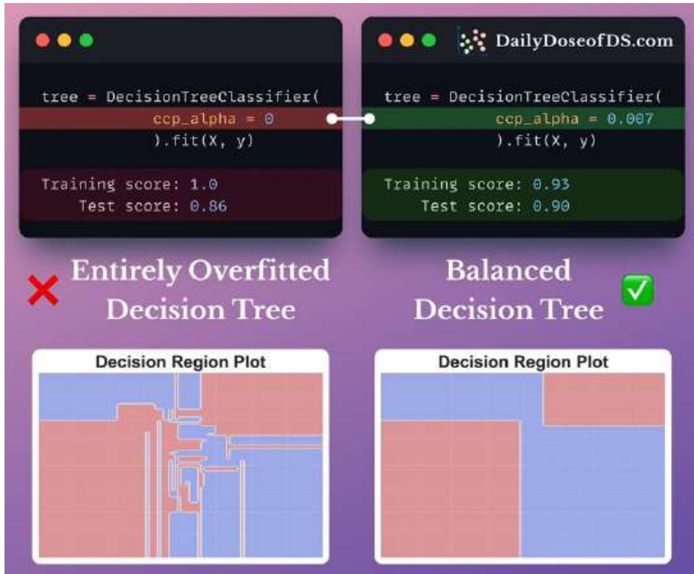
Cost-complexity pruning at the same increase in misclassification cost.

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

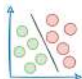
1. large value of `ccp_alpha` → results in underfitting
2. small value of `ccp_alpha` → results in overfitting

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

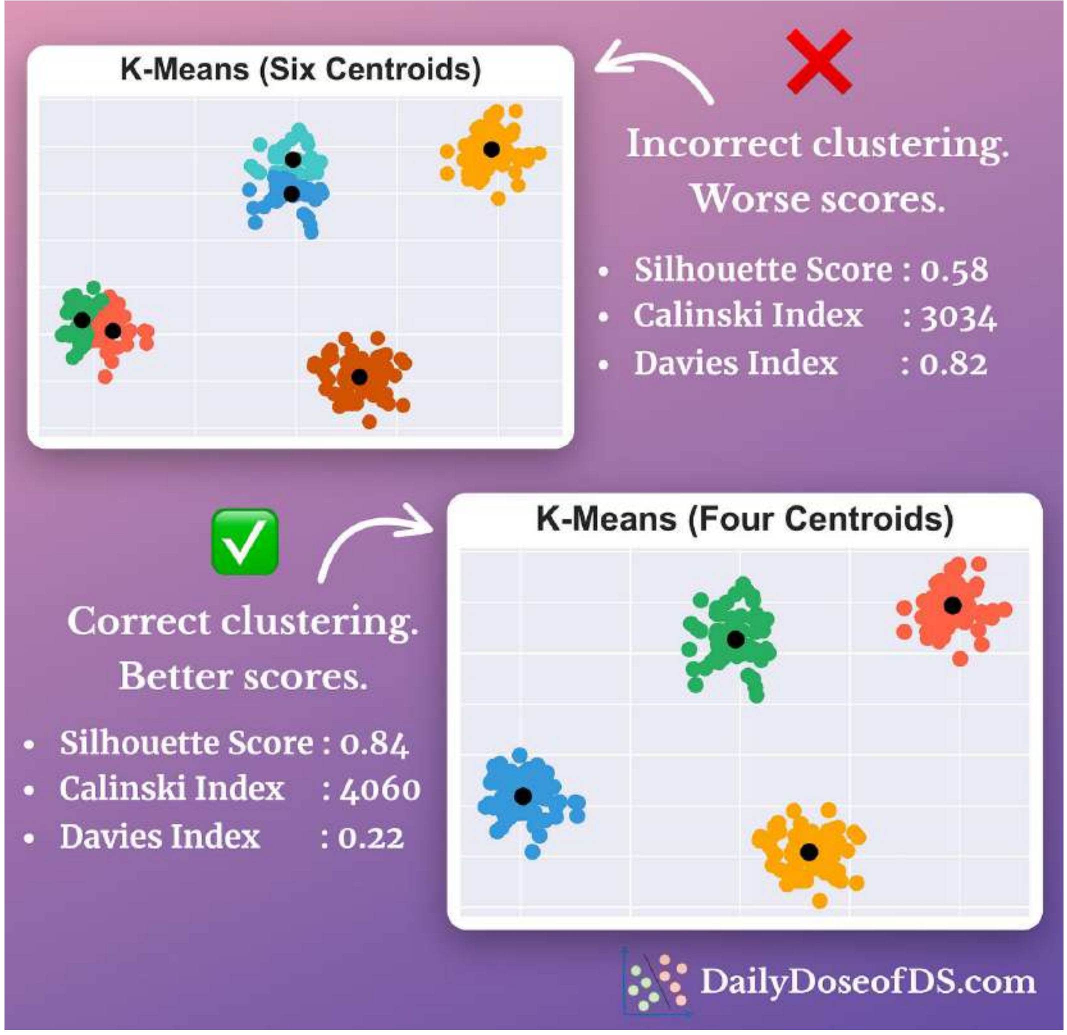
The effectiveness of cost-complexity-pruning is evident from the image below:



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



Evaluate Clustering Performance Without Ground Truth Labels



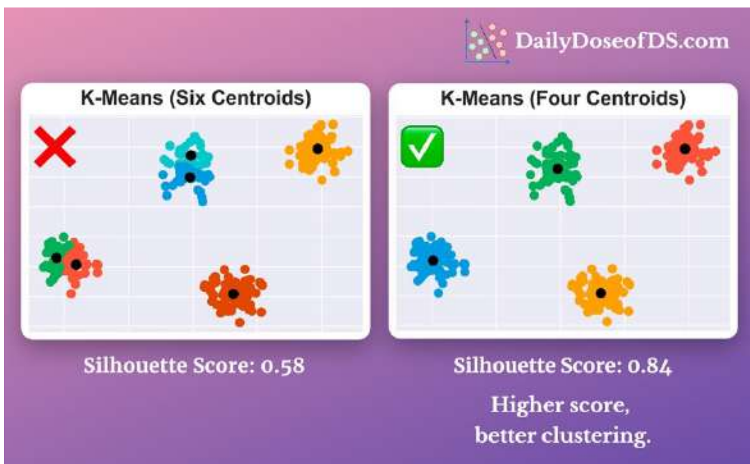
In the absence of ground truth labels, evaluating clustering performance is difficult.

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.

Silhoutte 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.

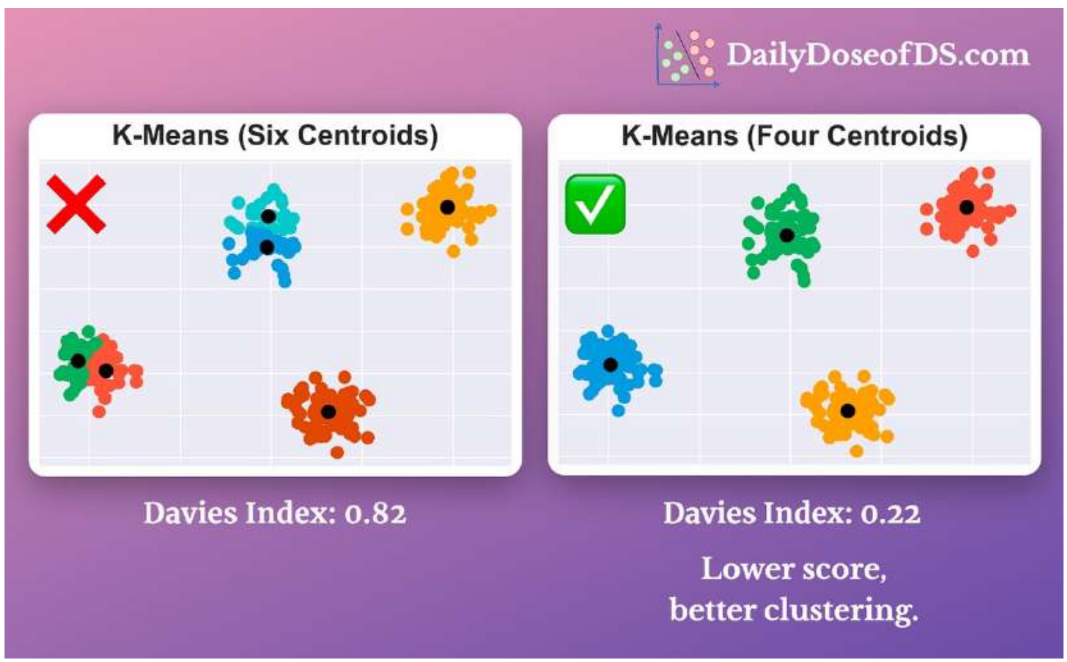
I covered this here if you wish to understand Silhoutte Coefficient with diagrams: The Limitations Of Elbow Curve And What You Should Replace It With.

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

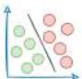
Luckily, they are neatly integrated with sklearn too.

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

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.

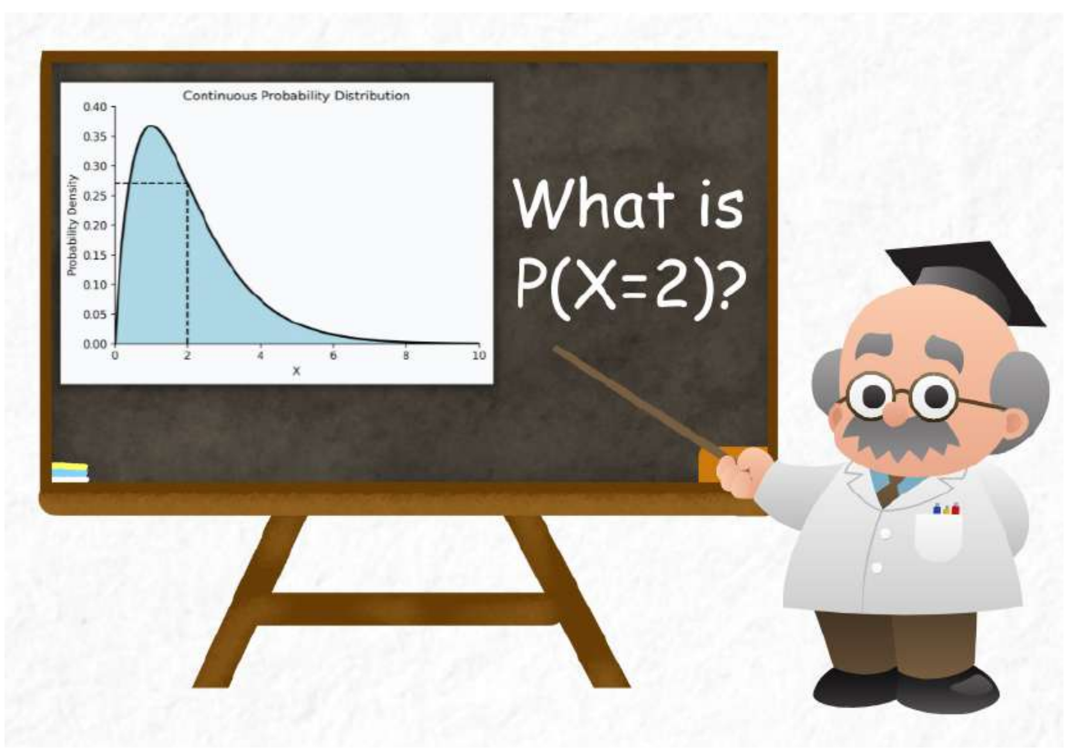
In most cases, the transition will require minimal code updates.

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.

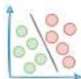
👉 Over to you: What are some other faster alternatives to Pandas that you are aware of?

The Most Common Misconception About Continuous Probability Distributions



This issue has many mathematical formulations.

Pleaase 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.

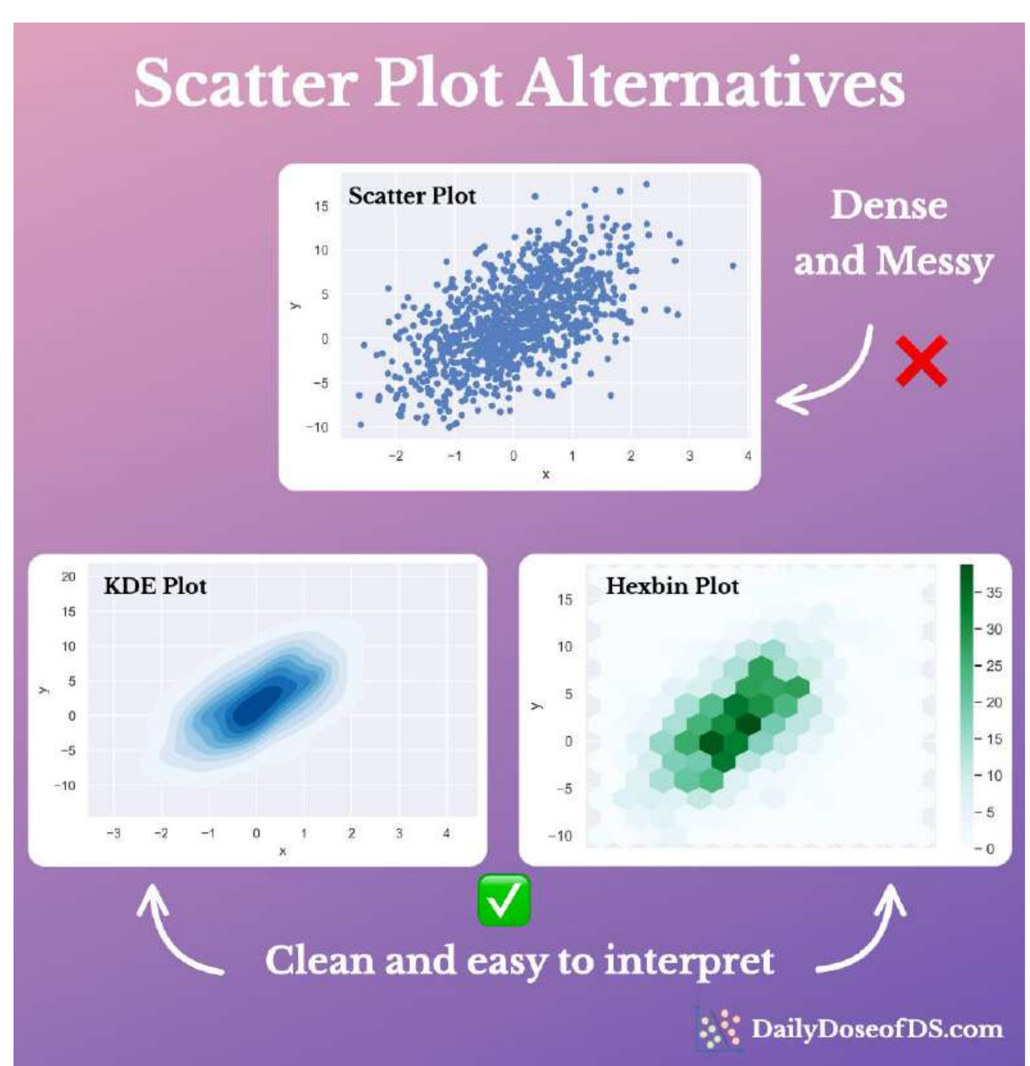
Scatter, bar, and line plots are the three most commonly used plots to visualize data.

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.

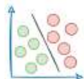
Bar plot alternative

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.



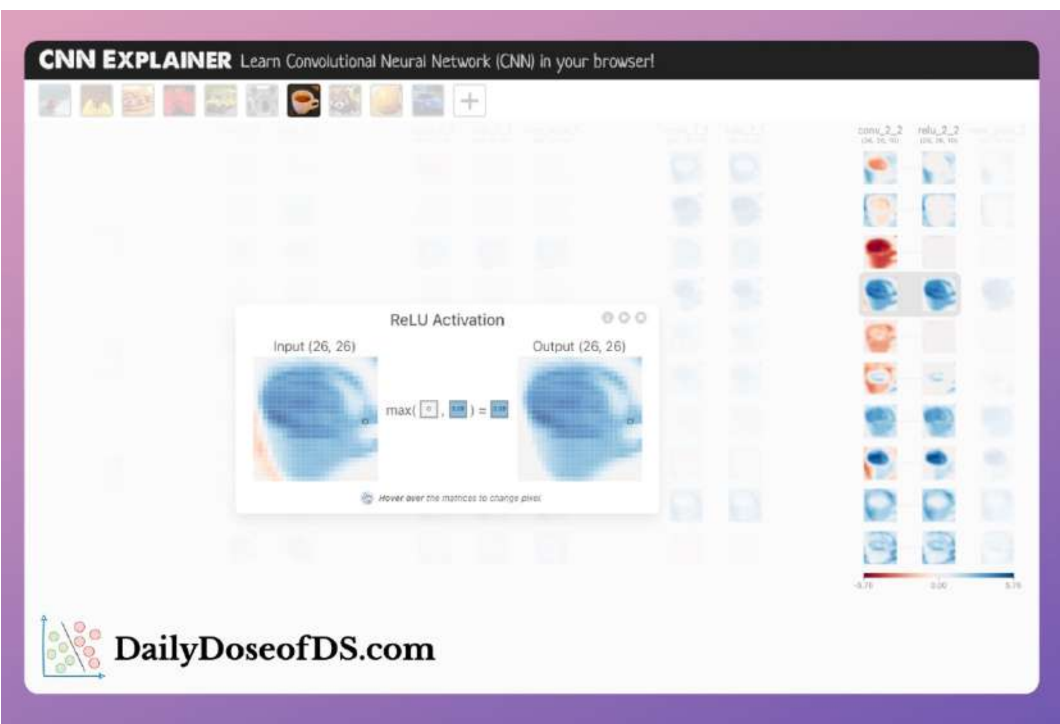
Bar/Line plot alternative — Waterfall chart

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



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.

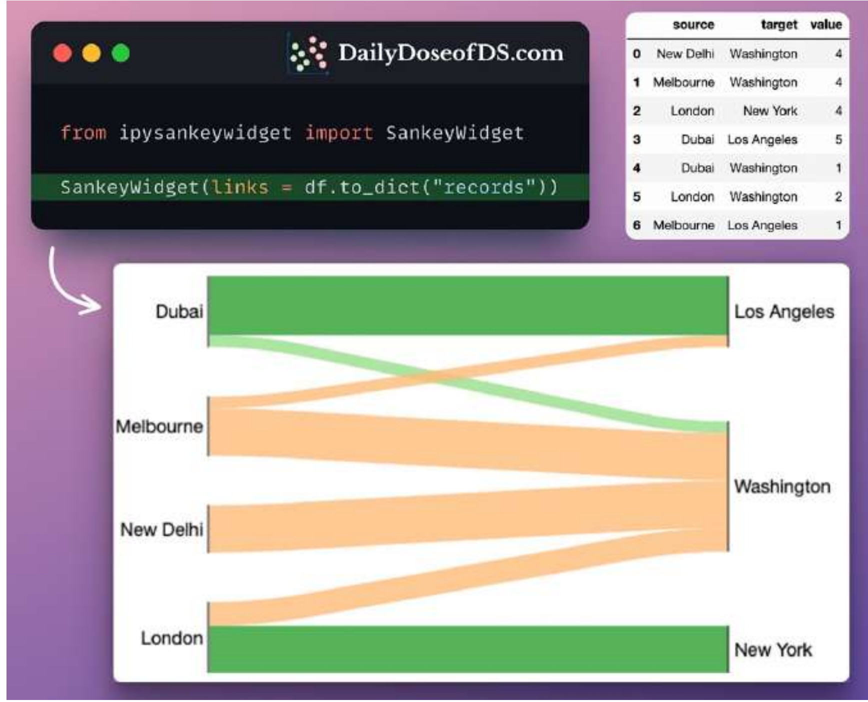
It is an incredible interactive tool to visualize the internal workings of a CNN.

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



	source	target	value
0	New Delhi	Washington	4
1	Melbourne	Washington	4
2	London	New York	4
3	Dubai	Los Angeles	5
4	Dubai	Washington	1
5	London	Washington	2
6	Melbourne	Los Angeles	1

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:

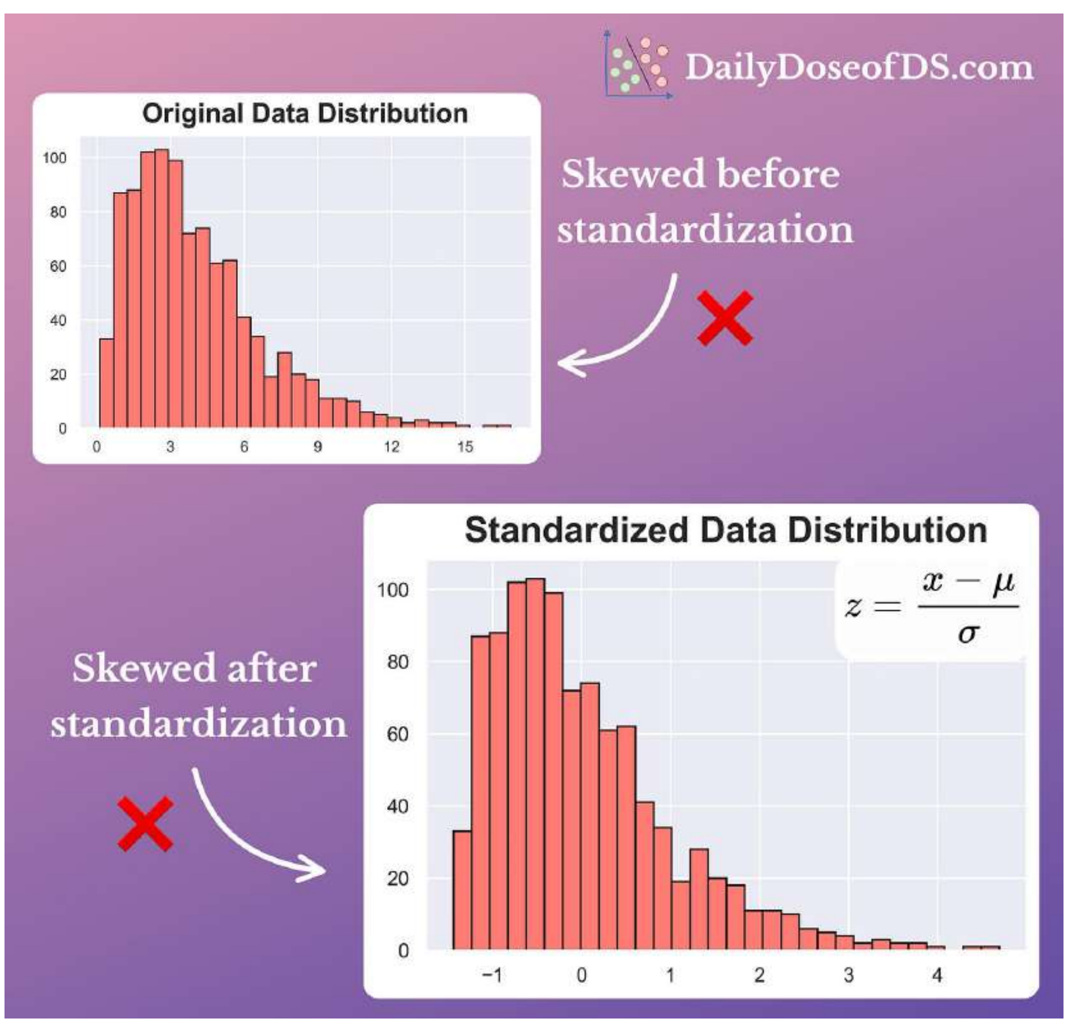
1. Washington hosts flights from all origins
2. New York only receives passengers from London
3. Majority of flights in Los Angeles come from Dubai
4. All flights from New Delhi go to Washington

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.

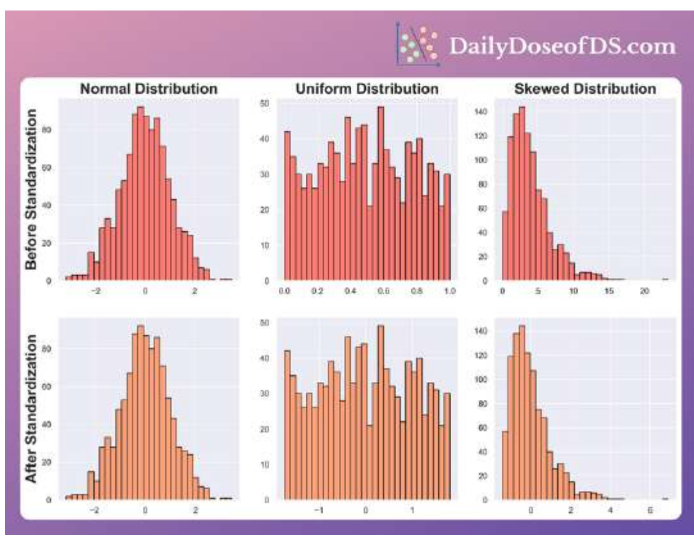
Yet, contrary to common belief, they NEVER change the underlying distribution.

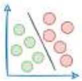
Instead, they just alter the range of values.

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:



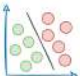


If you intend to eliminate skewness, scaling/standardization won't help.

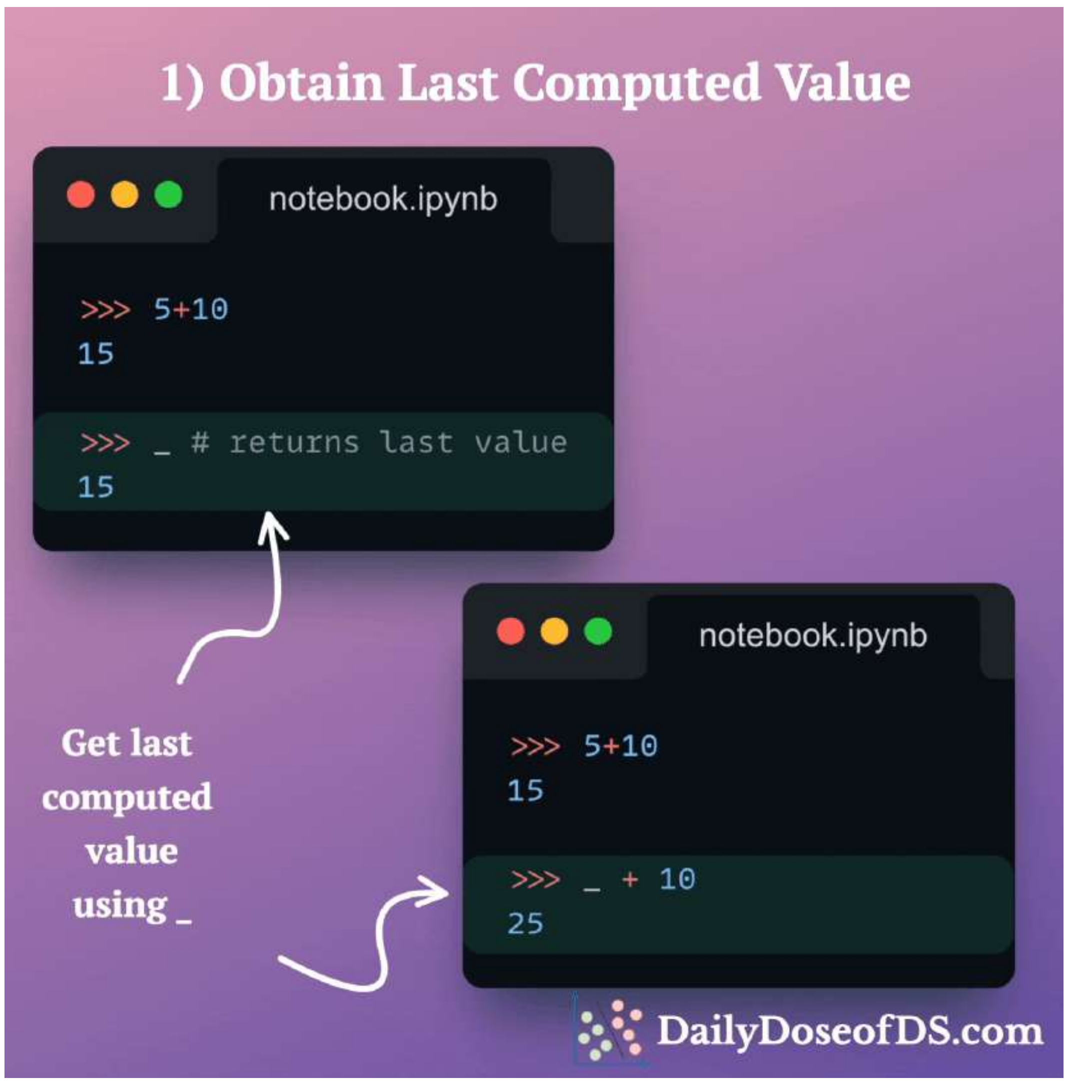
Try feature transformations instead.

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

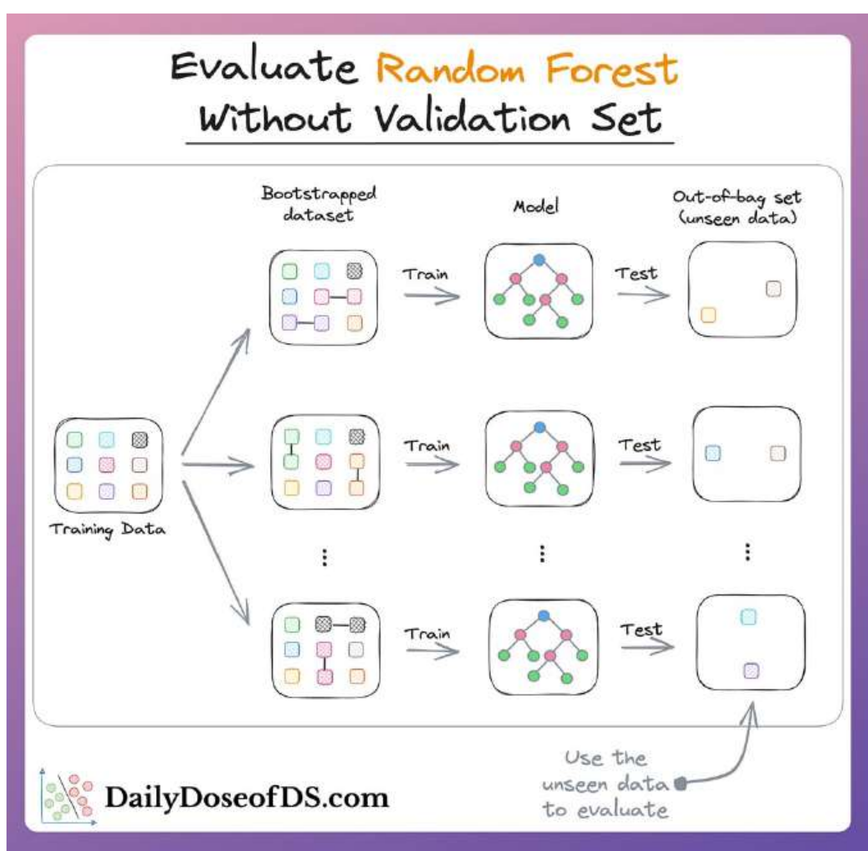


Underscore offers many functionalities in Python.

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:

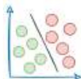
1. If you have less data, you can prevent data splitting
2. It's computationally faster than using, say, cross-validation
3. It ensures that there is no data leakage, etc.

Luckily, out-of-bag validation is neatly tied in sklearn's random forest implementation too.

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-rerun 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?