Member-only story Featured

ML SIMPLIFIED

Machine Learning Algorithms You Never Knew Existed, But Are Quite Useful

Every heard of Tsetlin Machine!!

Abhay Parashar · Follow

Published in The Pythoneers · 11 min read · 2 days ago

496 3

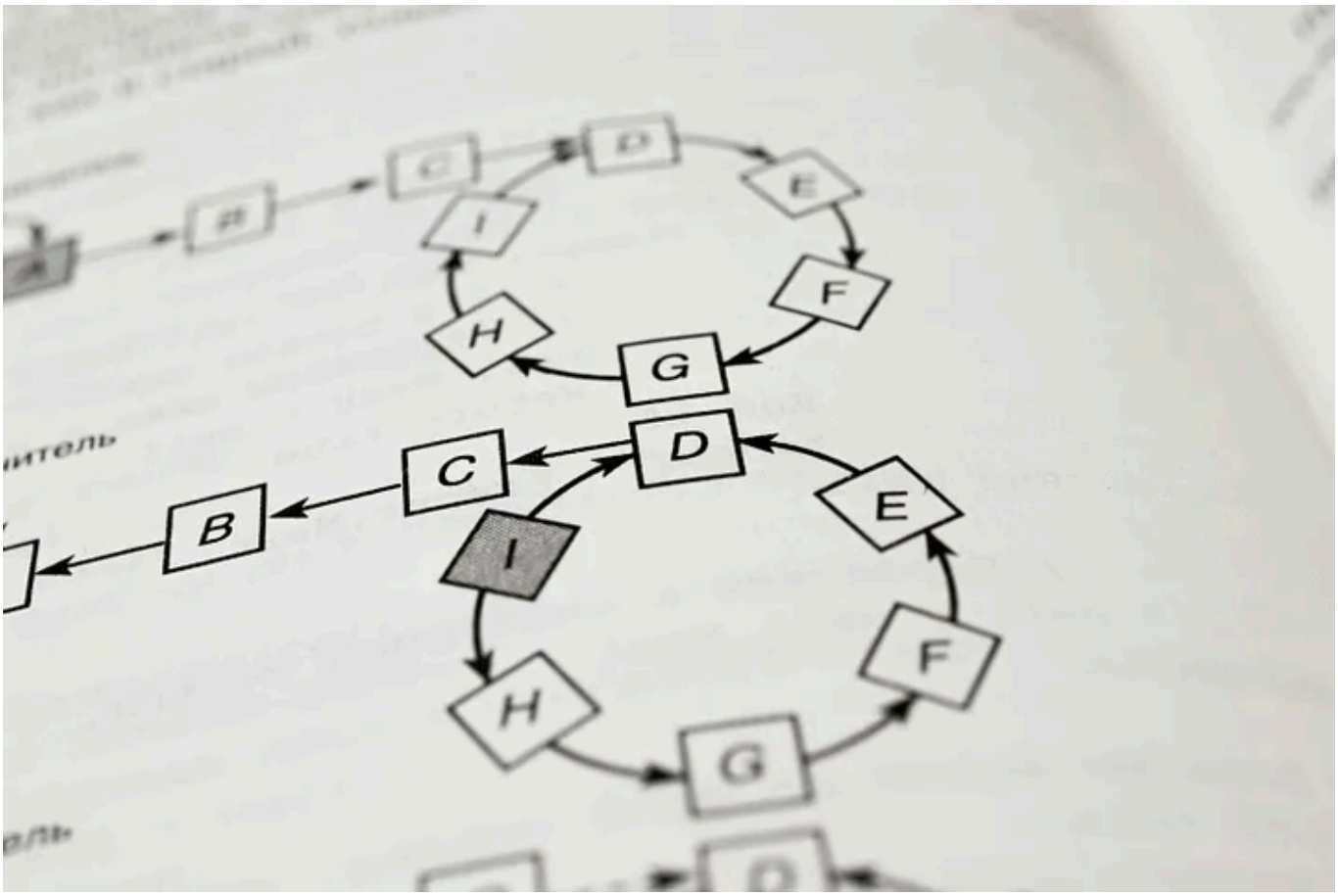


Photo by Андрей Сизов on Unsplash

When we talk about machine learning, the usual suspects — linear regression, decision trees, and neural networks — often steal the spotlight. But beyond these well-known models, there exist several lesser-known yet powerful algorithms that can tackle unique challenges with remarkable efficiency. In this article, we'll explore some of the most underrated yet highly useful machine-learning algorithms that deserve a place in your toolkit.

1. Symbolic Regression

Unlike traditional regression models that assume a predefined equation, Symbolic Regression **discovers the best mathematical expression to fit the data.** In simpler terms: Instead of assuming

$$y = mx + b$$

or

$$y = ax^2 + bx + c$$

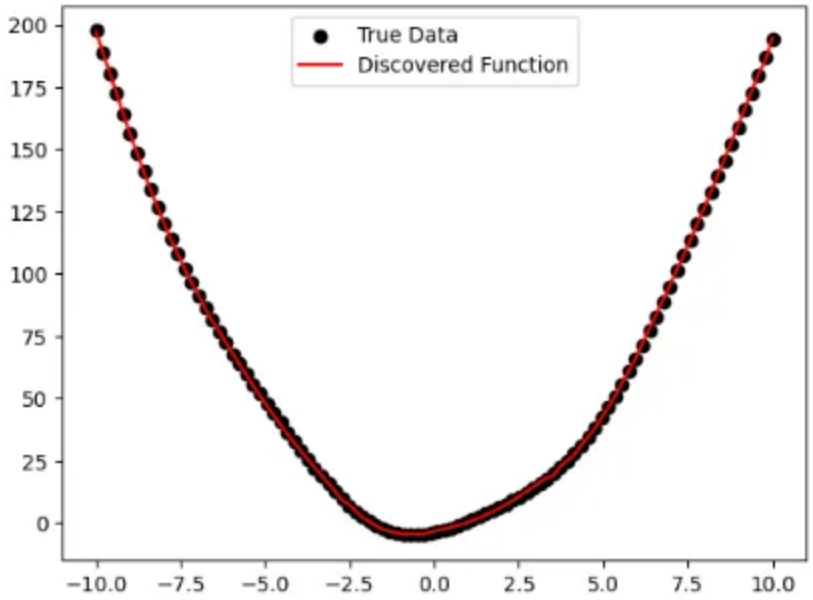
Symbolic Regression discovers the actual underlying equation as-

$$y=3\sin(x)+2x^2-4$$

It uses **genetic programming**, which evolves models over generations through mutation and crossover (similar to natural selection).

```
# !pip install gplearn

import numpy as np
import matplotlib.pyplot as plt
from gplearn.genetic import SymbolicRegressor

# Generate Sample Data
X = np.linspace(-10, 10, 100).reshape(-1, 1)
y = 3*np.sin(X).ravel() + 2*X.ravel()**2 - 4

# Initialize the Symbolic Regressor
sr = SymbolicRegressor(population_size=2000,
                       generations=20,
                       stopping_criteria=0.01,
                       function_set=('add', 'sub', 'mul', 'div', 'sin', 'cos', '
                       p_crossover=0.7,
                       random_state=42)

# Fit the model
sr.fit(X, y)

# Make Predictions
y_pred = sr.predict(X)

#
plt.scatter(X, y, color='black', label='True Data')
plt.plot(X, y_pred, color='red', label='Discovered Function')
```

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



See how well Symbolic Regression has performed fitting the sample data

Where Can You Use This in Your Projects?

1. **Predicting Physical Laws:** If you have data from physics experiments, symbolic regression can rediscover the original physical law.
2. **Stock Market Prediction:** It can derive an equation to model stock prices.
3. **Medical Research:** It can find the relationship between drugs and patient recovery.
4. **Data Science Competitions:** It's a hidden gem in Kaggle competitions!

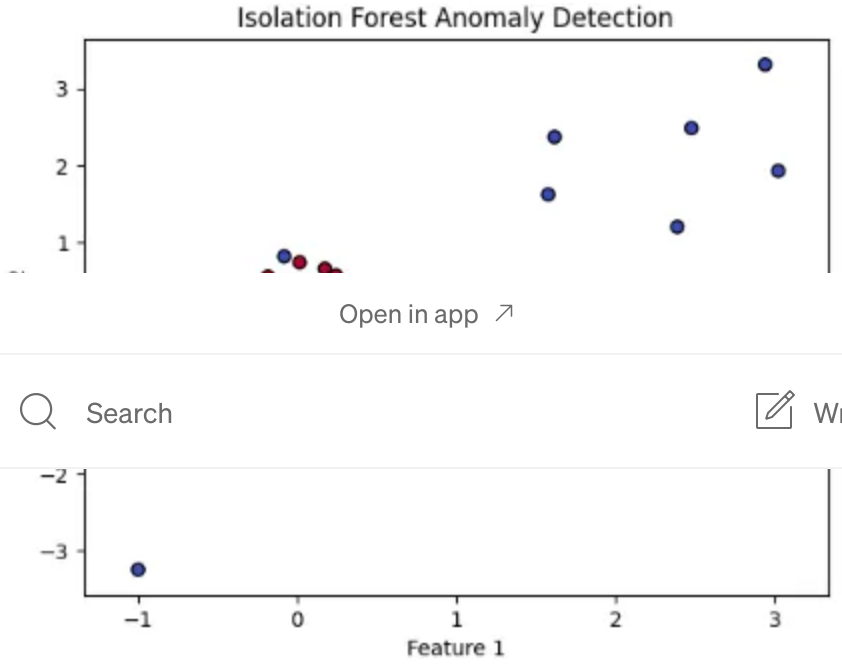
---

2. Isolation Forest (iForest)

Its an tree-based **anomaly detection** algorithm that isolates outliers faster than usual clustering or density-based methods (like DBSCAN or One-Class SVM). Instead of profiling normal data, it actively isolates **outliers** based on how quickly a point stands out in a randomly partitioned space.

It works well on **high-dimensional data and d**oes **not require labeled data**, that makes it suitable for unsupervised learning.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.ensemble import IsolationForest

# Generate Synthetic Data (Normal Data)
rng = np.random.RandomState(42)
X = 0.3 * rng.randn(100, 2)  # 100 normal points

# Add Some Anomalies (Outliers)
X_outliers = rng.uniform(low=-4, high=4, size=(10, 2))  # 10 outliers

# Combine Normal Data & Outliers
X = np.vstack([X, X_outliers])


iso_forest = IsolationForest(n_estimators=100, contamination=0.1, random_state=4
y_pred = iso_forest.fit_predict(X)

plt.scatter(X[:, 0], X[:, 1], c=y_pred, cmap='coolwarm', edgecolors='k')
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.title("Isolation Forest Anomaly Detection")
plt.show()
```



Outliers are presented in Blue color.

When To Use ?

1. Identifying fraudulent credit card transactions.
2. To Detect network intrusions or malware activity.
3. Spotting defective products in quality control.
4. Detecting rare diseases or abnormalities in health data.
5. Flag unusual stock market activity for insider trading detection.

---

3. Tsetlin Machine

The **Tsetlin Machine (TM)** algorithm first introduced by **Granmo in 2018**, built on the **Tsetlin Automaton (TA)**. Unlike traditional models, it utilizes **propositional logic** to detect intricate patterns, learning through a reward-and-penalty mechanism that refines its decision-making process.

One of the key strengths of TMs is their **low memory footprint** and **high learning speed**, making them highly efficient while delivering **competitive predictive performance** on benchmark datasets. Additionally, their simplicity enables seamless implementation on **low-power hardware**, making them ideal for energy-efficient AI applications.

Key Features —

- **Requires significantly less computation** than deep learning models.
- Easy to interpret because it generates **human-readable rules** instead of complex equations.
- It works best in constructing small scale AI systems.

You can find detailed information about this algorithm on their ***Github Repository*** and through this **research paper**.

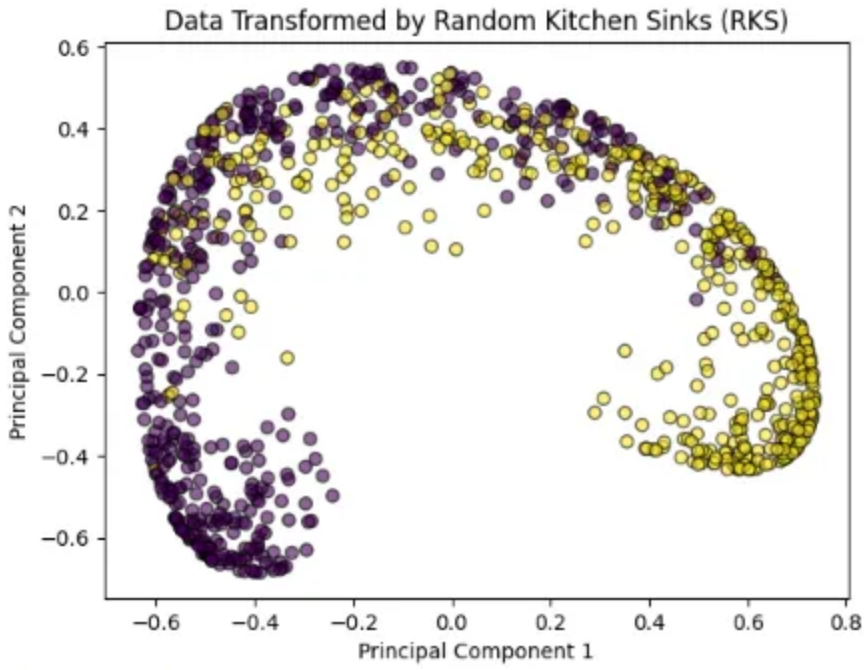
---

4. Random Kitchen Sinks (RKS)

Kernel methods like **Support Vector Machines (SVMs) and Gaussian Processes** are powerful, but they **struggle with large datasets** due to expensive kernel computations. **Random Kitchen Sinks (RKS)** is a clever trick that **approximates kernel functions efficiently**, making these methods scalable.

Instead of explicitly computing the **kernel function** (which can be computationally expensive), RKS projects data into a **higher-dimensional feature space** using **randomized Fourier features**. This allows models to approximate non-linear decision boundaries **without heavy computation**.

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_moons
from sklearn.kernel_approximation import RBFSampler

# Generate a non-linearly separable dataset
X, y = make_moons(n_samples=1000, noise=0.2, random_state=42)

# Apply Random Kitchen Sinks (RKS) for kernel approximation
rks = RBFSampler(gamma=1.0, n_components=500, random_state=42)
X_rks = rks.fit_transform(X)

# Visualizing the transformed feature space using PCA
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_rks)

plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y, edgecolors='k', alpha=0.6)
plt.title("Data Transformed by Random Kitchen Sinks (RKS)")
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.show()

print("Original Data Shape:", X.shape)
print("Transformed Data Shape (RKS Features):", X_rks.shape)
```




```
Original Data Shape: (1000, 2)
Transformed Data Shape (RKS Features): (1000, 500)
```

Where Can You Use This in Your Projects?

- Speeding up SVMs & Kernel Regression on large datasets.
- Efficiently approximating RBF (Radial Basis Function) kernels for scalable learning.
- Reducing memory and computational cost for non-linear models.

---

5. Bayesian Optimization

Bayesian Optimization is a **sequential, probabilistic approach** to optimizing expensive functions, like hyperparameter tuning in deep learning or machine learning models.

Instead of blindly testing different parameter values (like Grid Search or Random Search), Bayesian Optimization **models the objective function** using a probabilistic model (like a Gaussian Process) and selects the most promising values intelligently.

Where it's useful:

- **Hyperparameter tuning** — More efficient than grid search/random search.
- **A/B testing** — Finds the best variant without wasting resources.
- **Automated machine learning (AutoML)** — Powers tools like Google's AutoML.

```
import numpy as np
from bayes_opt import BayesianOptimization
```

```python
# Define an objective function (e.g., optimizing x^2 * sin(x))
def objective_function(x):
    return -(x**2 * np.sin(x))  # We aim to maximize, so we negate it

# Define parameter bounds
param_bounds = {'x': (-5, 5)}

# Initialize Bayesian Optimizer
optimizer = BayesianOptimization(
    f=objective_function,  # Function to optimize
    pbounds=param_bounds,  # Parameter space
    random_state=42
)

# Run Optimization
optimizer.maximize(init_points=5, n_iter=20)

# Best Parameters Found
print("Best parameters:", optimizer.max)
```

iter	target	x
1	1.496	-1.255
2	19.89	4.507
3	-3.941	2.32
4	-0.8119	0.9866
5	-3.477	-3.44
6	23.97	4.997
7	23.97	4.999
8	23.97	4.999
9	23.97	4.998
10	23.97	5.0
11	23.97	4.999
12	23.97	4.997
13	23.97	4.999
14	23.97	5.0
15	23.97	5.0
16	23.97	4.998
17	23.97	4.999
18	23.97	4.999
19	23.97	4.999
20	23.97	5.0
21	23.97	4.999
22	23.96	4.997
23	23.97	5.0
24	23.97	4.998
25	23.97	5.0

```
Best parameters: {'target': np.float64(23.972908020332486), 'params': {'x': np.float64(4.999204230296606)}}
```

6. Hopfield Networks

A **Hopfield network** is an type of recurrent neural network (RNN) that specializes in **pattern recognition** and **error correction** by storing binary patterns in its memory. When given a new input, it identifies and retrieves the closest stored pattern, even if the input is incomplete or noisy. This ability, known as **auto-association**, enables the network to reconstruct full patterns from partial or corrupted inputs. For example, if trained on images, it can recognize and restore them even when sections are missing or distorted.

You can find the Python implementation of Hopefield Networks by going through this **Google Colab Notebook or** Give a try with your own patterns through this **App.**

Where it's useful:

- **Memory recall systems** — It is useful for restoring corrupt images or fills in missing data.
- **Error correction** — Used in telecommunications for correcting transmission errors.
- **Neuroscience simulations** — Models human memory processes.

---

7. Self-Organizing Maps (SOMs)

A **Self-Organizing Map (SOM)** is a type of **neural network** that uses **unsupervised learning** to organize and visualize high-dimensional data in a **low-dimensional (usually 2D) grid**. Unlike traditional neural networks that rely on error correction (like backpropagation), SOMs use **competitive learning** — where neurons compete to represent input patterns.

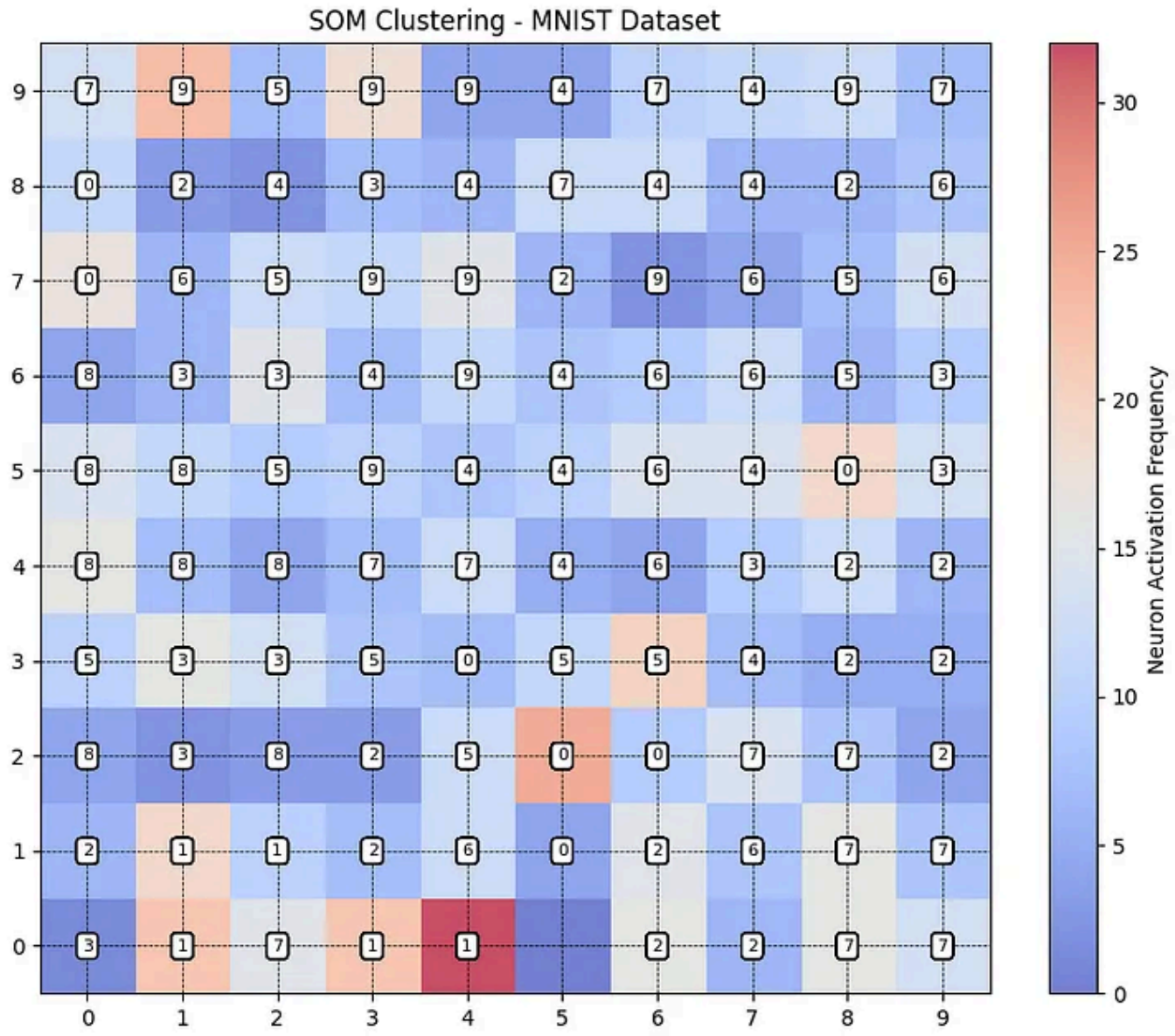
A key feature of SOMs is their **neighborhood function**, which helps maintain the original structure and relationships within the data. This makes them especially useful for **clustering, pattern recognition, and data exploration**.

Where it's useful:

- **Market segmentation** — Identifies different customer groups.
- **Medical diagnosis** — Clusters patient symptoms for disease detection.
- **Anomaly detection** — Detects fraud or defects in manufacturing.

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
from minisom import MiniSom
from tensorflow.keras.datasets import mnist

# Choose dataset: 'mnist', 'wine', 'customers'
DATASET = "mnist"  # Change to 'wine' or 'customers' for different datasets

# Load dataset
if DATASET == "mnist":
    (X_train, y_train), _ = mnist.load_data()
    X_train = X_train.reshape(X_train.shape[0], -1) / 255.0  # Flatten and norma
    X_train, y_train = X_train[:1000], y_train[:1000]  # Use subset for faster t

elif DATASET == "wine":
    df = sns.load_dataset("wine_quality")
    X_train = df.drop(columns=["quality"]).values
    X_train = X_train / np.linalg.norm(X_train, axis=1, keepdims=True)  # Normal
    y_train = df["quality"].values

elif DATASET == "customers":
    url = "https://raw.githubusercontent.com/MachineLearningWithPython/datasets/
    df = pd.read_csv(url)
    X_train = df[["Annual Income (k$)", "Spending Score (1-100)"]].values
    X_train = X_train / np.linalg.norm(X_train, axis=1, keepdims=True)
    y_train = None  # No predefined labels

# Initialize and train SOM
som_size = (10, 10)
```

```
som = MiniSom(som_size[0], som_size[1], X_train.shape[1], sigma=1.0, learning_ra
som.random_weights_init(X_train)
som.train_random(X_train, 1000)

# Create activity heatmap
activation_map = np.zeros(som_size)
for x in X_train:
    winner = som.winner(x)
    activation_map[winner] += 1

plt.figure(figsize=(10, 8))
plt.imshow(activation_map.T, cmap="coolwarm", origin="lower", alpha=0.7)
plt.colorbar(label="Neuron Activation Frequency")

# Overlay Data Points
for i, x in enumerate(X_train):
    winner = som.winner(x)
    label = str(y_train[i]) if y_train is not None else "•"
    plt.text(winner[0], winner[1], label, color="black", fontsize=8, ha="center"
             bbox=dict(facecolor="white", edgecolor="black", boxstyle="round,pad

plt.title(f"SOM Clustering - {DATASET.upper()} Dataset")
plt.xticks(range(som_size[0]))
plt.yticks(range(som_size[1]))
plt.grid(color="black", linestyle="--", linewidth=0.5)
plt.show()
```



8. Field-Aware Factorization Machines (FFMs)

Field-Aware Factorization Machines (FFMs) are an extension of **Factorization Machines (FMs)**, specifically designed for **high-dimensional, sparse data** — commonly found in recommendation systems and online advertising (CTR prediction).

In **standard Factorization Machines (FMs)**, each feature has a **single latent vector** for interaction with all other features. In **FFMs**, each feature has **multiple latent vectors, one per field** (group of features). This **field-awareness** allows FFMs to model interactions between different groups of features better

The FFMs model is given by:

$$y = w_0 + \sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} \langle v_{i,f_j}, v_{j,f_i} \rangle x_i x_j$$

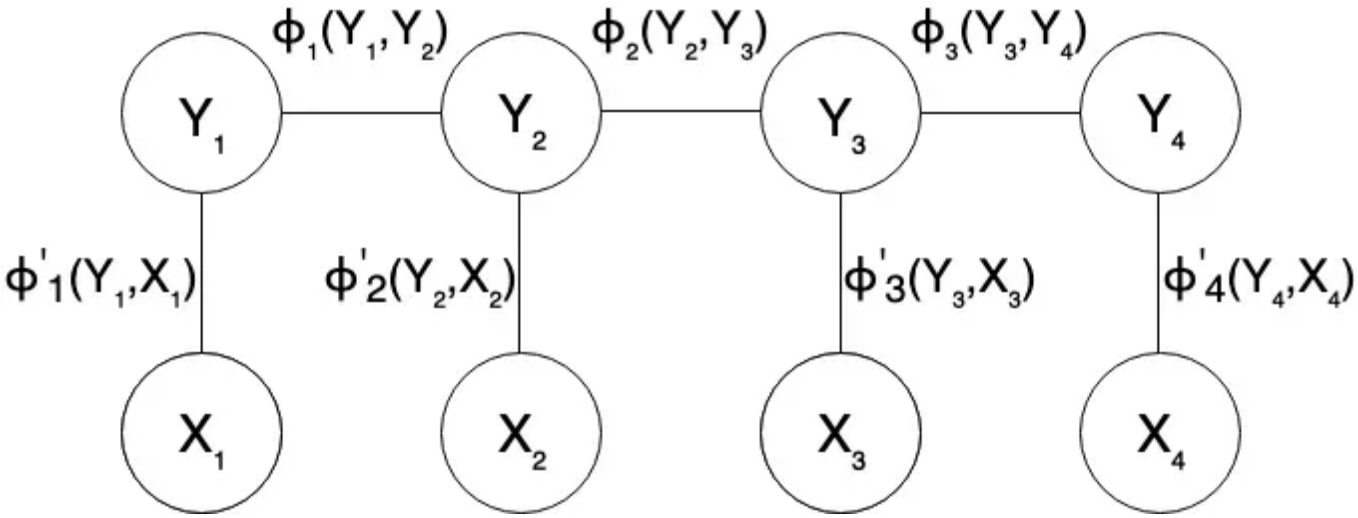
Where it's useful:

- **Recommendation systems** — Used by Netflix, YouTube, and Amazon.
- **Advertising** — Predicts which ads a user is likely to click on.
- **E-commerce** — Improves product suggestions based on user behavior.

---

9. Conditional Random Fields (CRFs)

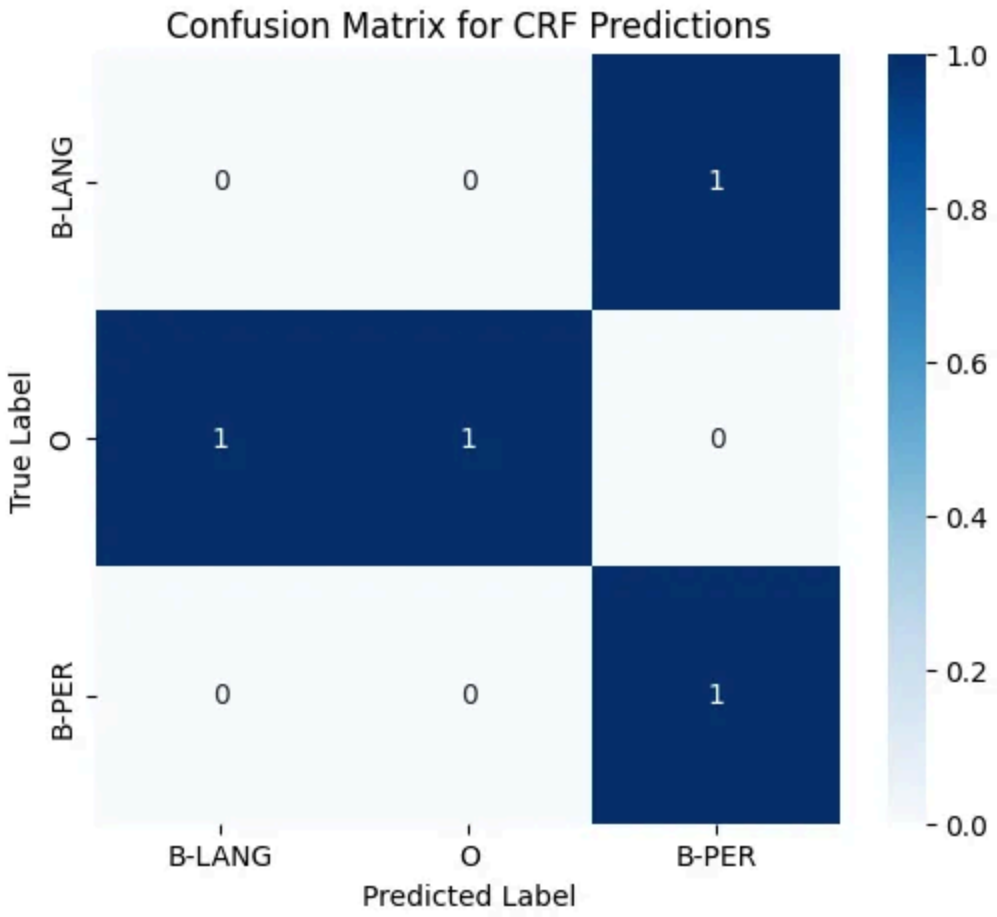
Conditional Random Fields (CRFs) are **probabilistic models used for structured prediction**. Unlike traditional classifiers that make independent predictions, CRFs take context into account, making them **useful for sequential data**.



Conditional Random Field structure [source]

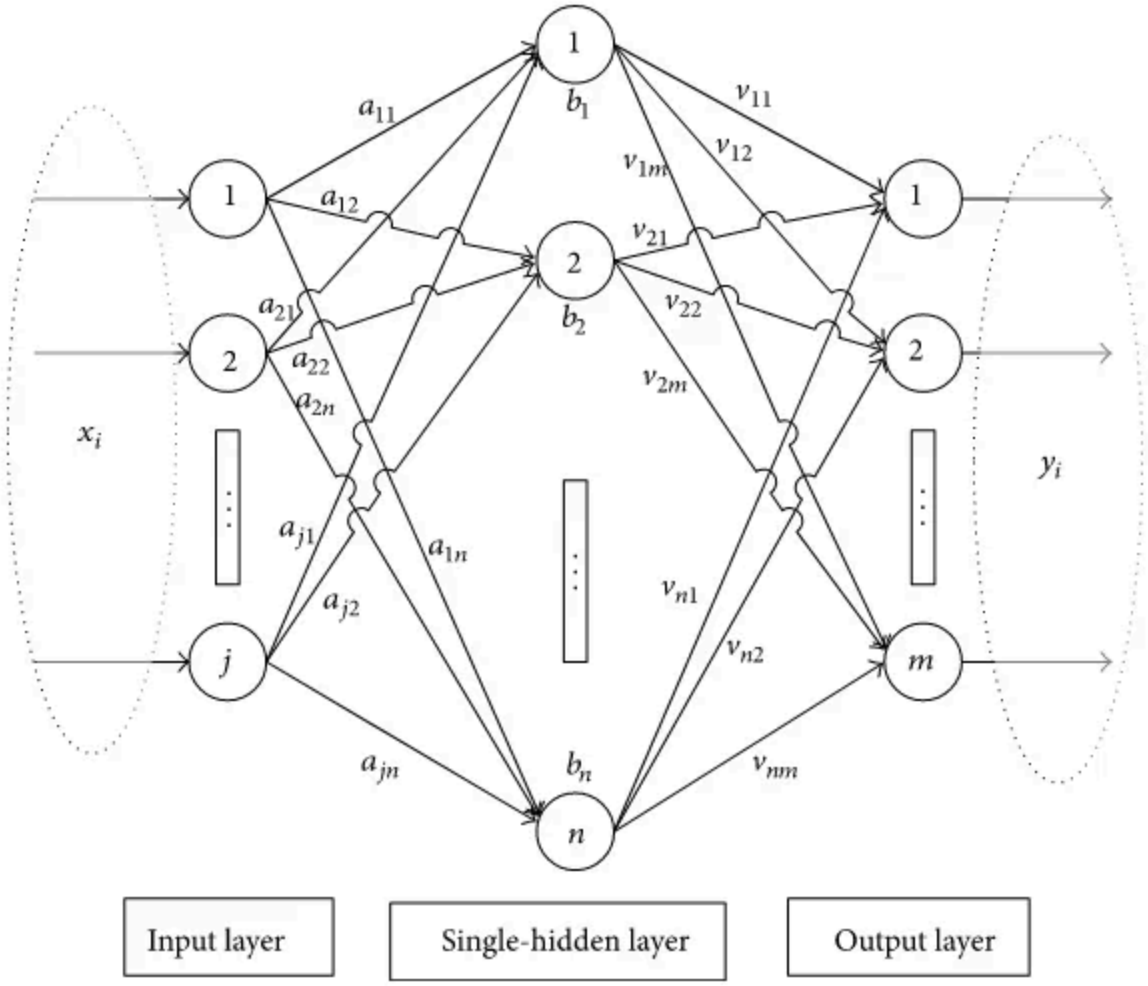
```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
from sklearn_crfsuite import CRF, metrics
from sklearn.model_selection import train_test_split

# Sample dataset (Simplified NER-like format)
X = [[{'word': 'John'}, {'word': 'loves'}, {'word': 'Python'}],
     [{'word': 'Alice'}, {'word': 'codes'}, {'word': 'in'}, {'word': 'Java'}]]

y = [['B-PER', 'O', 'B-LANG'],  # Named Entity Recognition (NER) Labels
     ['B-PER', 'O', 'O', 'B-LANG']]

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5, random_

# Define CRF model
crf = CRF(algorithm='lbfgs', max_iterations=50)
crf.fit(X_train, y_train)

# Make predictions
y_pred = crf.predict(X_test)

# Flatten lists for confusion matrix
y_test_flat = [label for seq in y_test for label in seq]
y_pred_flat = [label for seq in y_pred for label in seq]

# Get unique labels
labels = list(set(y_test_flat + y_pred_flat))
```

```
# Compute confusion matrix
conf_matrix = confusion_matrix(y_test_flat, y_pred_flat, labels=labels)

# Plot heatmap
plt.figure(figsize=(6, 5))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues", xticklabels=labels,
plt.title("Confusion Matrix for CRF Predictions")
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.show()
```



10. Extreme Learning Machines (ELMs)

Extreme Learning Machines (ELMs) are a type of **feedforward neural network** that train extremely fast by **randomly initializing hidden layer weights** and learning only the output weights. Unlike traditional neural networks, ELMs **don't use backpropagation**, making them significantly faster for training.



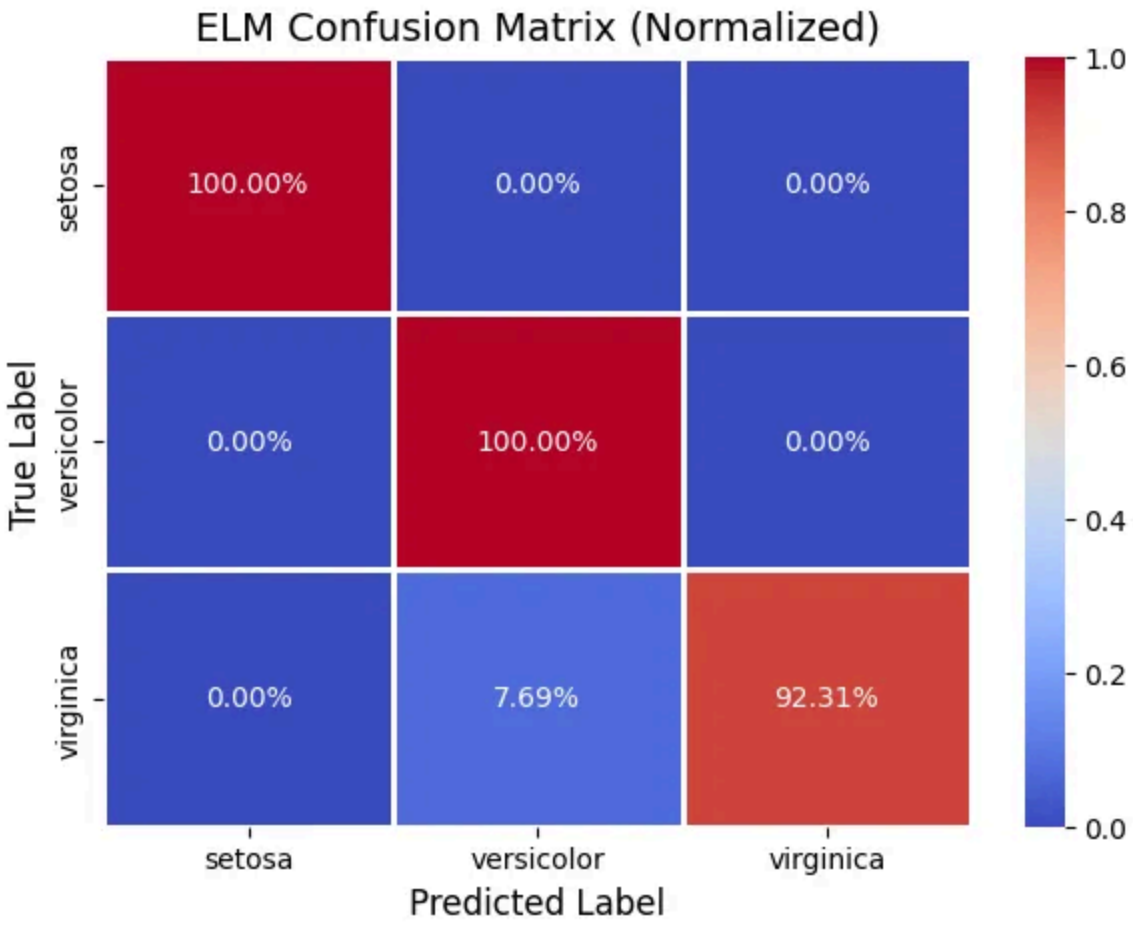
Single hidden Layer Feedforward Neural network, Source: Shifei Ding under CC BY 3.0 [source]

When to Use ELMs?

- When you need a **fast training speed** (compared to deep learning).
- For **classification and regression tasks** with large datasets.
- When a **shallow model (single hidden layer)** is sufficient.
- When you don't need fine-tuning of hidden layer weights.

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import confusion_matrix, accuracy_score
import hpelm  # High-Performance ELM

# Load dataset
iris = datasets.load_iris()
X, y = iris.data, iris.target

# Normalize features
scaler = StandardScaler()
X = scaler.fit_transform(X)

# Split dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_

# Convert labels to one-hot encoding (for ELM)
y_train_onehot = np.eye(len(set(y)))[y_train]
y_test_onehot = np.eye(len(set(y)))[y_test]

# Define and Train ELM
elm = hpelm.ELM(X_train.shape[1], y_train_onehot.shape[1])
elm.add_neurons(50, "sigm")  # 50 hidden neurons with sigmoid activation
elm.train(X_train, y_train_onehot, "c")  # 'c' for classification

# Make predictions
y_pred = elm.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)

# Evaluate model
accuracy = accuracy_score(y_test, y_pred_classes)
print(f"Accuracy: {accuracy:.2f}")

# Compute confusion matrix
cm = confusion_matrix(y_test, y_pred_classes)
cm_percentage = cm.astype("float") / cm.sum(axis=1)[:, np.newaxis]  # Normalize

# Create the figure
plt.figure(figsize=(7, 5))
sns.heatmap(cm_percentage, annot=True, fmt=".2%", cmap="coolwarm", linewidths=2,
            xticklabels=iris.target_names, yticklabels=iris.target_names)

# Improve readability
plt.xlabel("Predicted Label", fontsize=12)
```

```
    plt.ylabel("True Label", fontsize=12)
    plt.title("ELM Confusion Matrix (Normalized)", fontsize=14)
    plt.show()
```

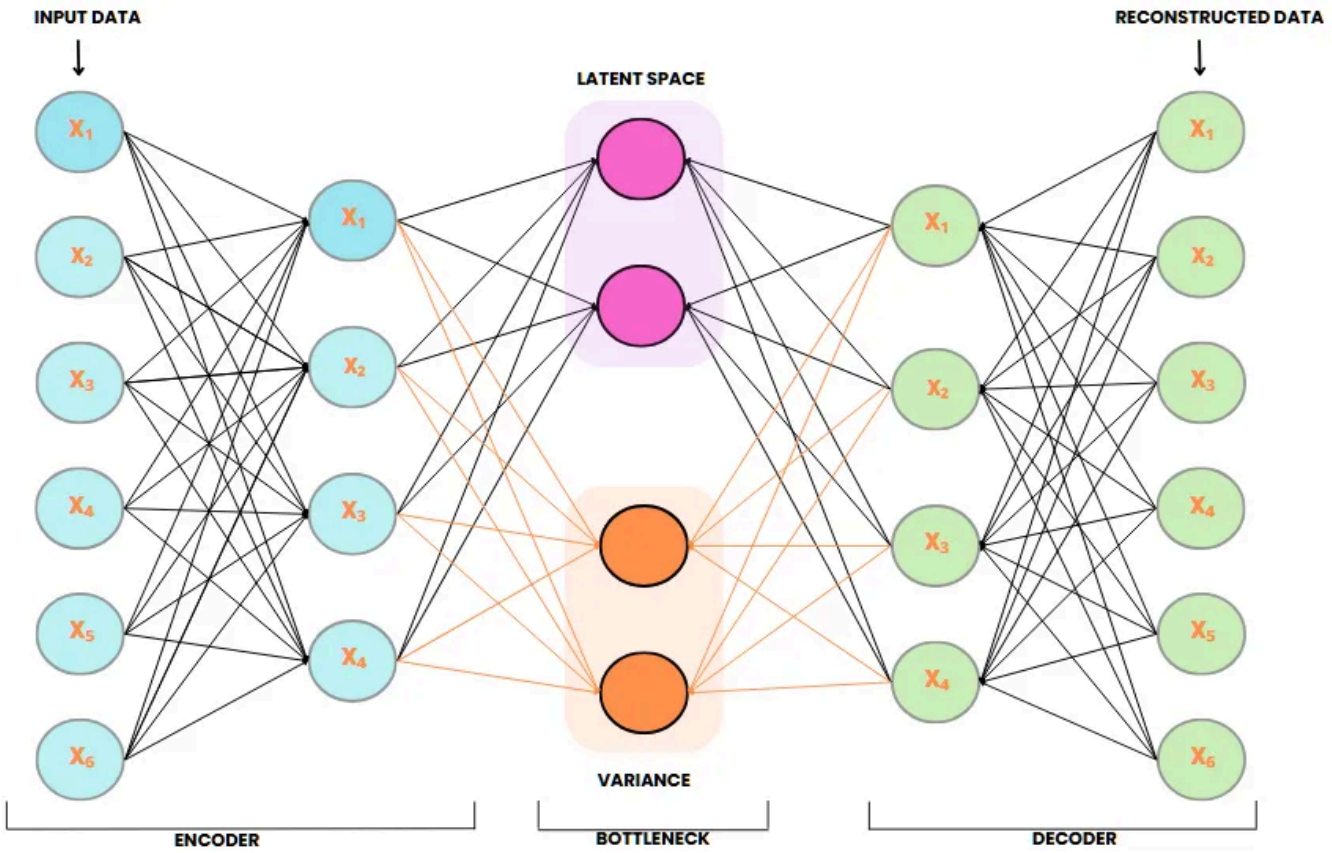


Bonus: Variational Autoencoder (VAE)

A Variational Autoencoder (VAE) is a generative deep learning model designed to learn latent representations of input data and generate new data samples that resemble the training data. Unlike standard Autoencoders, VAEs introduce stochasticity by learning a probabilistic latent space where

the encoder outputs mean (μ) and variance (σ) instead of fixed representations.

During training, random latent vectors are sampled from these distributions to generate diverse outputs via the decoder. This makes VAEs highly effective in tasks like image generation, data augmentation, anomaly detection, and latent space exploration.



Variational Autoencoder (VAE) Architecture

Thanks For Reading Till Here, If You Like My Content and Want To Support Me The Best Way is —

1. Leave a Clap👋and your thoughts 💬 below.
2. Follow Me On *Medium*.
3. Connect With Me On *LinkedIn*.
4. Attach yourself to *My Email List* to never miss reading another article of mine

Don’t Forgot To Smash 50 Claps 👋If u Liked This Article….It won’t take much time 😄

Machine Learning | Data Science | Artificial Intelligence | Education | Python

Published in The Pythoneers

12.8K Followers · Last published 17 hours ago

Following

Your home for innovative tech stories about Python and its limitless possibilities. Discover, learn, and get inspired.

Written by Abhay Parashar

30K Followers · 80 Following

Follow

Guarding Systems by Day 👨‍💻, Crafting Stories by Night ✍🏽, Weaving Code by Starlight 🤖

Responses (3)

Alex Mylnikov

What are your thoughts?

Sara Nóbrega she/her
17 hours ago

Some of these are wildly underrated. Love how you showed real code too - bookmarking this for the next weird dataset I meet.

6 Reply

Sai Manohar
9 hours ago

Would be helpful if the pros and cons are mentioned.

3 Reply

Madan kumar yelburgi
7 hours ago

Good to know some of the new algos mentioned. Thanks for the brief article.

1 Reply

More from Abhay Parashar and The Pythoneers

In The Pythoneers by Abhay Parashar

18 Insanely Useful Python Automation Scripts I Use Everyday

Scripts That Increased My Productivity and Performance Even More

Dec 18, 2024 4K 39

In The Pythoneers by Abhay Parashar

How to Explain Each Core Machine Learning Model in an Interview

From Regression to Clustering to CNNs: A Brief Guide to 25+ Machine Learning Models

Mar 12 1.6K 16

In The Pythoneers by Aashish Kumar

Best Practices for Structuring a Python Project Like a Pro! 🚀

Follow these best practices to structure your Python projects like a pro – clean, scalable,...

Mar 14 509 10

In The Pythoneers by Abhay Parashar

14 Powerful Prompt Engineering Techniques You Need to Try

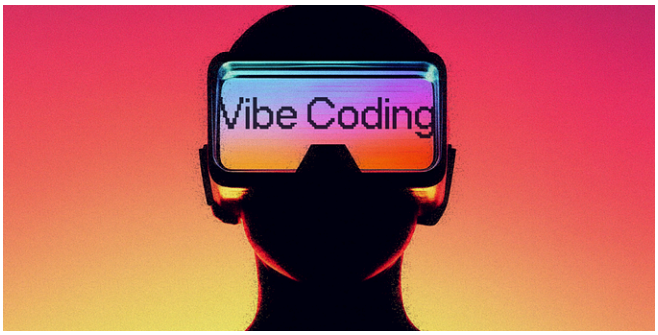
Because with the right prompt, there's no limit to what AI can do

Feb 21 469 4

See all from Abhay Parashar

See all from The Pythoneers

Recommended from Medium



In Data Science Collective by Yu-Cheng Tsai

Building an LLM Agent with N8N and Open-WebUI

A Hands-On Perspective from a Machine Learning Developer

3d ago 138

In UX Collective by Pete Sena

Cracking the code of vibe coding

Not all vibes are good.

4d ago 1K 21

In Towards AI by Thomas Reid

Run OpenAI's New Agents For Free

Using Ollama and a local LLM

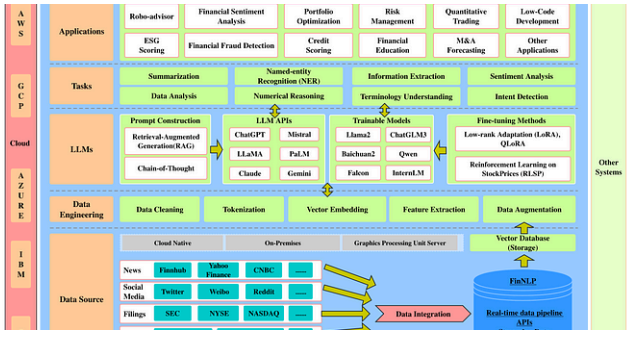
6d ago 148 1

In OpenSourceScribes by C. L. Beard

9 Necessary Open Source Tools

Blazing-fast JSON parsing to federated AI

5d ago 142 1



In AI monks.io by JIN

FinGPT: The Future of Financial Analysis—Revolutionizing Marke...

Discover how FinGPT is disrupting traditional financial tools like Bloomberg Terminal,...

Amos Gyamfi

The 3 Best Python Frameworks To Build UIs for AI Apps

Build user-friendly AI chat interfaces crucial for seamless interactions, demo presentatio...