(IJRST) 2025, Vol. No. 15, Issue No. 4, Oct-Dec **e-ISSN: 2249-0604, p-ISSN: 2454-180X**

Numerical Linear Algebra: Applications in Machine Learning Using Python

Supraja S

RV PU College, Jayanagar

DOI:10.37648/ijrst.v15i04.006

[1] *Received: 30/09/2025; Accepted: 24/10/2025; Published: 29/10/2025*

Abstract

Modern machine learning is, at its core, large-scale numerical linear algebra (NLA): multiplying matrices, solving least-squares systems, computing eigenvectors/SVDs, and approximating kernels efficiently. As datasets and model sizes grow, classical "exact" methods often become too slow or too memory-heavy, making randomized and iterative NLA essential. This paper surveys where NLA appears in machine learning pipelines, which algorithms matter most (direct, iterative, and randomized), and how Python practitioners implement them using NumPy/SciPy and GPU-enabled frameworks. We also include a comparative analysis that connects algorithmic choices to accuracy, runtime, and scalability.

1. Introduction

Machine learning problems frequently involve linear models, embeddings, dimensionality reduction, kernel methods, and optimization. Each of these relies on NLA primitives: SVD/PCA for compression, QR/least squares for regression, eigen-decompositions for spectral methods, and linear solves in second-order optimization or implicit layers. What has changed in the 2015–2025 period is scale: matrices are often too large to factorize exactly, so practitioners increasingly use (a) iterative methods (Krylov subspace solvers), and (b) randomized numerical linear algebra (RandNLA), which uses randomness to produce fast, high-quality approximations. A modern survey of RandNLA emphasizes low-rank approximation, regression, and kernel matrix approximations as major "real-world" drivers.

ML component	Dominant NLA operation	Typical goal
Feature compression	PCA / truncated SVD	Reduce dimension, denoise
Linear & ridge regression	Least squares / normal equations	Fit weights efficiently
Kernel learning	PSD approximation (Nyström/sketching)	Avoid full (n \times n) kernels
Deep learning systems	Batched GEMM + decompositions	Speed and stability

2. NLA building blocks used in ML

2.1 Matrix factorizations and stability

Key decompositions:

- **SVD** (X \approx U\Sigma V^\top): best low-rank approximation (in spectral/Frobenius norms).
- **QR** (X = QR): stable least squares and orthogonalization.

[1] How to cite the article: Supraja S. (October, 2025); Numerical Linear Algebra: Applications in Machine Learning Using Python; *International Journal of Research in Science and Technology*; Vol 15, Issue 4; 87-92, DOI: http://doi.org/10.37648/ijrst.v15i04.006

- **Cholesky** (A = LL^\top) for SPD systems: fast, but memory-intensive for large dense problems.
- **Eigen-decomposition** for symmetric matrices: spectral clustering, PCA on covariance, graph Laplacians.

2.2 Iterative linear solvers

When matrices are large and sparse (common in graphs, recommender systems, PDE-inspired ML, or second-order approximations), iterative solvers are preferred. A representative line of work on randomized iterative methods unifies a family of "sketch-and-project" approaches and connects them to well-known methods like randomized Kaczmarz and coordinate descent.

2.3 Randomization, sketching, and approximate decompositions

RandNLA targets tasks like:

- **Approximate low-rank factorization** (randomized SVD/subspace iteration).
- **Sketched least squares** (compress rows/columns to solve faster).
- **Kernel matrix approximations** (Nyström, landmark sampling).
  A modern foundation paper summarizes these techniques and their practical tradeoffs. Recent work also studies robustness properties of randomized SVD variants (important when data is noisy).

Technique	What it approximates	Why ML uses it
Randomized SVD	Top singular vectors/values	PCA at scale, embeddings
Sketching	Linear regression / norms	Faster training on tall data
Nyström	Kernel PSD matrix	Scalable kernel methods
Randomized iterative solves	(Ax=b)	Sparse/structured systems

3. Applications in machine learning

3.1 Dimensionality reduction and representation learning (PCA/SVD)

PCA is widely used for preprocessing, compression, denoising, and as a building block for embeddings. The bottleneck is computing top singular vectors for large matrices. Randomized SVD and subspace iteration often deliver near-PCA quality with far lower runtime, especially when the target rank is small relative to input dimension. Foundations and practical guidance for these randomized matrix methods are covered in a modern Acta Numerica survey.
At the same time, sensitivity to noise matters in real datasets; recent analysis explores how randomized SVD behaves under noise and when it may degrade.

Python sketch (randomized truncated SVD idea)

```
import numpy as np

# X: (n_samples, n_features)
# Goal: approximate top-k components with randomized range finding
def randomized_pca(X, k, oversample=10, n_iter=2, seed=0):
rng = np.random.default_rng(seed)
   n, d = X.shape
   Omega = rng.standard_normal((d, k + oversample))
   Y = X @ Omega
```

```
    for _ in range(n_iter):
        Y = X @ (X.T @ Y)
    Q, _ = np.linalg.qr(Y, mode="reduced")
    B = Q.T @ X
Uhat, S, Vt = np.linalg.svd(B, full_matrices=False)
    U = Q @ Uhat[:, :k]
    return U, S[:k], Vt[:k, :]
```

Approach	**Accuracy expectation**	**Typical use case**
Exact SVD	Best possible	Small/medium dense matrices
Randomized SVD	Near-exact for low-rank-ish data	Large PCA/embeddings
Power/Lanczos variants	Good for top spectrum	Sparse or structured matrices

3.2 Linear models: least squares and regularization

Linear regression and ridge regression reduce to least squares problems. For dense data, direct solvers (QR/Cholesky) are common; for huge "tall" matrices, sketched least squares can reduce cost dramatically by compressing the system before solving. RandNLA's role in regression is highlighted in modern surveys that cover regression and streaming/one-pass algorithms.

Solver style	**Strength**	**Weakness**
QR-based least squares	Very stable	Expensive for very large data
Normal equations + Cholesky	Fast if well-conditioned	Can be numerically risky
Sketched least squares	Scales well	Approximation error to manage

3.3 Kernel methods at scale (Nyström and landmarking)

Kernel methods (SVMs, kernel ridge regression, Gaussian processes) require working with an (n \times n) PSD kernel matrix, which is infeasible for large (n). The Nyström method approximates the kernel using a smaller set of landmark points. A line of work analyzes sampling strategies (including ridge leverage score ideas) for Nyström-style approximations.
Overall, modern RandNLA treatments explicitly call out kernel matrix approximation (Nyström/CUR/streaming) as central ML applications.

Kernel scaling method	**Memory cost**	**Notes**
Full kernel	(O(n^2))	Accurate, rarely feasible
Nyström	(O(nm))	Choose (m \ll n), sampling matters
Random features	(O(nd))	Good for shift-invariant kernels, feature dimension drives quality

3.4 Optimization and "linear solves inside learning"

Second-order-ish methods, implicit differentiation, and some meta-learning setups require solving linear systems involving Jacobians/Hessians (or approximations). Large systems are typically solved iteratively. Randomized iterative methods unify and extend classical solvers, providing flexible sampling-based updates with convergence guarantees.

Where the solve appears	Example	Why iterative helps
Implicit layers	((I - J)^{-1}v)	Avoid explicit inverse
Second-order updates	(H \Delta = g)	Hessian too large to form
Graph methods	Laplacian systems	Sparse, scalable solves

4. Python implementation ecosystem

Python ML succeeds because high-level APIs sit on optimized linear algebra backends (BLAS/LAPACK on CPU; CUDA/cuBLAS/cuSOLVER on GPU). A 2020 survey of machine learning in Python highlights how core ML workflows depend on efficient array/tensor linear algebra libraries.

On the deep learning side, PyTorch exposes many decompositions and solvers through torch.linalg, which makes it practical to prototype NLA-heavy ML methods with GPU acceleration.

Example: least squares on GPU with PyTorch

```
import torch

A = torch.randn(20000, 200, device="cuda")
b = torch.randn(20000, 1, device="cuda")

# Solve min_x ||Ax - b||_2 using torch.linalg.lstsq
x = torch.linalg.lstsq(A, b).solution
```

Tooling and performance can vary across frameworks; benchmarking work discusses “linear algebra awareness” and related performance considerations in TensorFlow/PyTorch ecosystems (archived as an arXiv DOI).

Layer	Common Python choice	What it provides
CPU dense NLA	NumPy/SciPy	LAPACK/BLAS-based factorizations
Sparse NLA	SciPy sparse	Iterative solvers, sparse formats
GPU tensors	PyTorchtorch.linalg	Batched decompositions, GPU solves
Scaling patterns	Sketching/randomization	Reduced memory/time on big data

5. Comparative analysis and practical guidelines

5.1 Accuracy–speed–memory tradeoffs

The right method depends on matrix structure:

- **Dense + medium size:** exact factorizations (SVD/QR) are fine.
- **Large + approximately low-rank:** randomized SVD/subspace iteration is often the best win.
- **Sparse + huge:** iterative solvers (CG/GMRES-style) with good preconditioning are critical; randomized iterative variants provide additional flexibility.
- **Kernel methods:** Nyström or related sampling methods are often required to fit memory.

Task	Exact method	Scalable alternative	Typical tradeoff
PCA / embeddings	Full SVD	Randomized SVD	Huge speedup, tiny accuracy loss
Least squares	QR	Sketched regression	Faster, approximation error
Linear system solve	Cholesky	Iterative / randomized iterative	Less memory, needs conditioning care
Kernel learning	Full kernel	Nyström	Smaller memory, sampling-sensitive

5.2 When to worry about conditioning and noise

Two common failure modes in ML NLA:

1. **Ill-conditioned problems** (collinearity, badly scaled features): direct normal equations can amplify error; prefer QR/SVD or regularization.
2. **Noise-dominated spectra**: randomized low-rank methods can pick up noise directions if the eigen/singular value gap is small; modern analyses study this sensitivity explicitly.

Symptom	Likely cause	Fix
Unstable coefficients	Ill-conditioning	Standardize + ridge, use QR/SVD
Slow iterative convergence	Poor conditioning	Preconditioner, better scaling
PCA components look noisy	Weak spectral gap	Increase iterations/oversampling, validate

6. Conclusion

Numerical linear algebra is not just "under the hood" of machine learning; it is the mechanism through which most ML models are trained, compressed, and scaled. In Python, practical NLA-enabled ML comes from (1) optimized CPU/GPU backends exposed through NumPy/SciPy and torch.linalg, and (2) algorithmic strategies like sketching, randomized low-rank approximation, and iterative solvers that make previously impossible problems feasible. For practitioners, the best results come from matching the algorithm to matrix structure (dense vs sparse, low-rank vs full-rank, well-conditioned vs ill-conditioned) and validating approximation quality on downstream metrics, not just reconstruction error.

References

Calandriello, D., Lazaric, A., & Valko, M. (2017). Analysis of Nyström method with sequential ridge leverage score sampling. In *Proceedings of the 34th International Conference on Machine Learning* (Vol. 70, pp. 1832-1840). PMLR.

Gower, R. M., & Richtárik, P. (2017). Randomized iterative methods for linear systems. *SIAM Journal on Matrix Analysis and Applications, 38*(2), 511–544. https://doi.org/10.1137/15M1025487

Kressner, D., Steinlechner, M., & Vandereycken, B. (2016). Preconditioned low-rank Riemannian optimization for linear systems with tensor product structure. *SIAM Journal on Scientific Computing, 38*(4), A2018–A2044. https://doi.org/10.1137/15M1032909

Martinsson, P.-G., & Tropp, J. A. (2020). Randomized numerical linear algebra: Foundations and algorithms. *Acta Numerica, 29*, 403–572. https://doi.org/10.1017/S0962492920000021

Raschka, S., Patterson, J., & Nolet, C. (2020). Machine learning in Python: Main developments and technology trends in data science, machine learning, and artificial intelligence. *Information, 11*(4), 193. https://doi.org/10.3390/info11040193

Sankaran, A., Alashti, N. A., & Psarras, C. (2022). Benchmarking the linear algebra awareness of TensorFlow and PyTorch. *arXiv*. https://doi.org/10.48550/arXiv.2202.09888

Wang, Z., & Mahoney, M. W. (2025). On the noise sensitivity of the randomized SVD. *IEEE Transactions on Information Theory, 71*(5), 3642–3657. https://doi.org/10.1109/TIT.2024.3450412