# Application of SVD for Image-to-Video Search and Understanding Its Limitations

Reynard Anderson Wijaya - 13524111[1,2]
*Program Studi Teknik Informatika*
*Sekolah Teknik Elektro dan Informatika*
*Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia*
[1]13524111@std.stei.itb.ac.id, [2]reynard15082006@gmail.com

***Abstract*—This paper presents an application of Singular Value Decomposition (SVD) for image-to-video search, where a query image is matched against frames extracted from a set of videos. Each frame is converted into a grayscale vector representation and reduced in dimensionality using SVD-based Principal Component Analysis (PCA). Similarity between the query and video frames is computed using cosine similarity in the reduced feature space. The method provides efficient data retrieval speed and preserves important structural information of the images. However, SVD-based search has several limitations, including sensitivity to illumination changes, dependence on grayscale intensity distributions, and the linearity assumption that restricts its ability to model complex visual variations. This study highlights both the practical strengths and fundamental drawbacks of SVD-based search applications in content-based video retrieval.**

***Keywords*—Content-based video retrieval, dimensionality reduction, image-to-video search, Principal Component Analysis (PCA), Singular Value Decomposition (SVD)**

## I. Introduction

The growth of digital video content has created a need for efficient methods to find visual information inside video databases. Common keyword-based search is insufficient for several practical applications, such as video surveillance, scene detection, and content verification, where users need to retrieve video frames based on visual similarity rather than text descriptions. This need motivates the development of content-based video retrieval systems capable of comparing a query image against thousands of video frames.

A major challenge in such systems is the high dimensionality of image data. For example, a single $64 \times 64$ grayscale frame contains 4096 pixel values, making direct comparison computationally expensive. Therefore, dimensionality reduction is necessary in order to compress visual information while keeping the important structural patterns of the image.

Singular Value Decomposition (SVD) provides a method for extracting the most informative components from image data. When applied through Principal Component Analysis (PCA), SVD generates a compact feature representation that allows fast similarity computation. However, SVD-based approaches also exhibit several limitations, including sensitivity to illumination changes, dependence on grayscale intensity distributions, and the linearity assumption that restricts its ability to model complex visual variations.

This paper presents the implementation of an SVD-based program for image-to-video search and analyzes its practical limitations. The proposed method consists of frame extraction, vectorization, SVD-based dimensionality reduction, and cosine similarity matching. Our aim is to provide a clear explanation of how SVD performs in image-to-video search and to identify the factors that affect its accuracy.

## II. Theoretical Basis

### A. Singular Value Decomposition (SVD)

Singular Value Decomposition (SVD) is a matrix factorization technique that decomposes matrix $A_{m \times n}$ with a rank of $k$ into three component matrices:

$$A = U\Sigma V^T$$

$$A = U\Sigma V^T = [\mathbf{u}_1 \ \mathbf{u}_2 \ \cdots \ \mathbf{u}_k | \mathbf{u}_{k+1} \ \cdots \ \mathbf{u}_m] \left[\begin{array}{ccc|c} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_k \\ \hline \multicolumn{3}{c|}{0_{(m-k)\times k}} & 0_{(m-k)\times(n-k)} \end{array}\right] \left[\begin{array}{c} \mathbf{v}_1^T \\ \mathbf{v}_2^T \\ \vdots \\ \mathbf{v}_k^T \\ \hline \mathbf{v}_{k+1}^T \\ \vdots \\ \mathbf{v}_n^T \end{array}\right]$$

Figure 1. Singular Value Decomposition (SVD) Formula
Source: [2]

where

- $U_{m \times m}$ is an orthogonal matrix whose columns are the left singular vectors of $A$.
- $V^T_{n \times n}$ is an orthogonal matrix whose columns are the right singular vectors of $A$.
- $\Sigma$ is a diagonal matrix containing the non-negative singular values of $A$ arranged in descending order ($\sigma_1 \geq \sigma_2 \geq \sigma_3 \geq \ ...$).

The singular values reflect the amount of variance obtained by each corresponding singular vector from the original matrix. In order to obtain an approximation of matrix $A$, we can perform SVD truncation by only keeping the top-$x$ largest singular values from $\Sigma$, and also keeping the first $x$ column from $U$ and $V$. The resulting

low-rank approximation is computed as:

$$A_x = U_x \Sigma_x V_x^T$$

This reduced matrix preserves the most significant information (variance) while discarding noise and redundancy. In applications such as image-to-video search, SVD truncation significantly reduces storage and computational cost while maintaining the essential structure of the data.

### *B. Principal Component Analysis (PCA)*

Principal Component Analysis (PCA) is a dimensionality reduction technique that transforms high dimensional data into a lower dimensional subspace while keeping the original variance as much as possible. PCA identifies the directions (principal components) where the data shows the greatest variability and projects the original data into these directions.

Given matrix $A_{m \times n}$, we first compute the mean of all column vectors:

$$\mu = \frac{1}{N} \sum_{i=1}^{N} x_i ; \; N = number\ of\ columns$$

The data is then centered by subtracting the mean from each column vector:

$$x_{i,c} = x_i - \mu ; \; i = 1, 2, \ldots, N.$$

All centered vectors $x_{i,c}$ form the centered data matrix $X_c$. From this matrix, the covariance matrix can be computed as:

$$C = \frac{1}{N} X_c X_c^T$$

In image-to-video search applications, however, $X_c$ is typically very large, making the computation of the full covariance matrix computationally expensive. To address this, SVD truncation is used to obtain a reduced representation. By keeping only the top $k$ singular vectors, we construct a projection matrix:

$$Z = X_c V_k^T$$

where $V_k$ contains the top $k$ right singular vectors. The matrix $Z$ serves as the reduced-dimension representation of the data. In this paper, $Z$ is subsequently used to compute cosine similarity between an image query and video frames (as explained in the following section).

### *C. Cosine Similarity*

Cosine similarity is a type of similarity measure used to determine how similar two vectors are regardless of their length. Given two vectors $a$ and $b$, cosine similarity is defined as:

$$sim(a, b) = \frac{a \cdot b}{||a||\,||b||}$$

where

- $a \cdot b$ is the dot product of the vectors.
- $||a||$ and $||b||$ are their Euclidean norms.

Cosine similarity ranges from -1 to 1 (opposite to identical). Because it is based on the direction of the vectors rather than length, it is effective for comparing high dimensional data such as image feature vectors. After dimensionality reduction using PCA or SVD, cosine similarity provides a computationally efficient way to measure how closely the query image matches each video frame.

In content-based video retrieval, frames with the highest cosine similarity scores are considered the most visually similar to the query.

## III. Implementation

In this implementation, several libraries are used to facilitate the functionality of the image-to-video search program. These libraries support video processing, numerical computation, dimensionality reduction, and similarity measurement. The main libraries used are as follows:

1. OpenCV (cv2): Used for video handling and image processing, including frame extractions from videos, image resizing, and grayscale conversion.
2. NumPy: Provides efficient array operations and numerical computation, mainly for vector representation of image frames and matrix manipulation.
3. Scikit-learn: Used for implementing Principal Component Analysis (PCA), which also uses Singular Value Decomposition (SVD) to reduce the dimensionality of image vectors.
4. Pickle: Used to serialize and store the trained PCA model, metadata, and extracted features for later reuse without repeating the preprocessing stage.
5. Operating System (os) module: Handles directory traversal and file management for loading video data, saving model files, and accessing query images.

The implementation consists of two main stages: model building and query searching. In the model building stage, frames are extracted from videos and transformed into low-dimensional feature vectors using SVD-based PCA. In the query searching stage, a query image is projected into the same feature space and compared against stored video frame features using cosine similarity.

### *A. Model Building Stage*

Model building is the preprocessing stage where video data are transformed into compact numerical representations in a form of matrix that can later be used for similarity-based search. This stage consists of three main steps: frame extraction and vectorization, dimensionality reduction using SVD-based PCA, and model storage.

```
def __init__(self, video_dir, frame_size=(64,64), step=2, out_dim=64):
    self.video_dir = video_dir
    self.frame_size = frame_size
    self.step = step
    self.out_dim = out_dim
    self.features = []
    self.meta = []
    self.video_names = []
```

Figure 2. ModelBuilder class structure
Source: author's source code

### *A.1 Frame Extraction and Vectorization*

In this step, each video from the dataset is processed to extract each representative frame at a fixed time interval. All video files with extensions such as .mp4, .mkv, .avi, and .mov are first detected from the specified video directory.

Each video is opened and its frame rate (frames per second) is obtained. Frames are then sampled every fixed time interval (one frame every two seconds in this implementation) to reduce redundancy and computational cost. For each selected frame, the following preprocessing steps are applied:

1. The frame is resized to a fixed resolution of $64 \times 64$ pixels to ensure dimensionality across all frames.
2. The frame is converted into grayscale to simplify the representation and reduce sensitivity to color variations.
3. The grayscale image is flattened into a one-dimensional vector with a length of 4096.
4. A single video contains $4096 \times N$ matrix representation with N being the number of frames.

```
def extract_frames(self):
    video_files = [os.path.join(self.video_dir, f)
                   for f in os.listdir(self.video_dir)
                   if f.lower().endswith(('.mp4','.mkv','.avi','.mov'))]
    print("Found videos:", video_files)
```

Figure 3. extract_frames function (obtain video data)
Source: author's source code

```
for vid_i, vf in enumerate(video_files):
    print("Processing video:", vf)
    self.video_names.append(os.path.basename(vf))
    cap = cv2.VideoCapture(vf)
    fps = cap.get(cv2.CAP_PROP_FPS)
    interval = int(fps * self.step)

    frame_idx = 0
    total = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))

    while frame_idx < total:
        print(f" Extracting the {frame_idx+1}/{total} frame", end='\r')
        cap.set(cv2.CAP_PROP_POS_FRAMES, frame_idx)
        ret, frame = cap.read()
        if not ret:
            break

        frame = cv2.resize(frame, self.frame_size)
        frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

        vec = frame.flatten()
        self.features.append(vec)
        self.meta.append((vid_i, frame_idx / fps))

        frame_idx += interval

    print(f"Extracted a total of {len(self.meta)} frames until video {vf}")
    cap.release()
```

Figure 4. extract_frames function (transform the features in each frame into one-dimensional vector of length 4096)
Source: author's source code

```
self.features = np.array(self.features).astype("float32")
print("Raw features:", self.features.shape)
```

Figure 5. extract_frames function (change feature list into numpy array)
Source: author's source code

Each resulting vector represents a video frame in pixel space and is stored as a feature vector. Additionally, metadata consisting of the video index and the corresponding timestamp (in seconds) is recorded in self.meta for later retrieval.

### *A.2 Dimensionality Reduction Using SVD-Based PCA*

The extracted frame vectors are high-dimensional and computationally expensive to compare directly. Therefore, dimensionality reduction is applied using Principal Component Analysis (PCA), which internally relies on Singular Value Decomposition (SVD).

PCA is fitted on the entire set of extracted frame vectors to identify the directions of maximum variance in the data. The original one-dimensional vector with a length of 4096 are then projected into a lower-dimensional subspace with dimension $k = 64$. This process preserves the most significant structural information while discarding noise and redundancy.

```
def reduce(self):
    pca = PCA(n_components=self.out_dim)
    reduced = pca.fit_transform(self.features)

    self.reduced = reduced.astype("float32")
    self.pca = pca

    print("Reduced:", self.reduced.shape)
```

Figure 6. reduce function (reduce each vectors from 4096-dimension into 64-dimension)
Source: author's source code

The output of this stage is a reduced feature matrix where each row corresponds to a frame representation in the PCA subspace. The trained PCA model is also retained, as it is required to encode query images into the same feature space during the search stage.

### *A.3 Model Storage and Metadata Preservation*

After dimensionality reduction, the resulting data are stored for efficient reuse without repeating the preprocessing stage. Four main components are saved:

1. The reduced feature vectors of all extracted video frames, stored in features.npy. This NumPy binary file contains the feature matrix of size $N \times k$ after PCA transformation, where $N$ is the number of frames and $k$ is the reduced dimensionality.
2. The trained PCA model is stored in pca.pkl. This pickle file keeps the PCA parameters, including the mean vector and principal components.
3. Metadata, stored in meta.pkl, which maps each feature vector to its corresponding video index

and timestamp.

4. Video filenames, stored in video_names.pkl, which connects video indices with their actual file names.

```
def save(self, outdir):
    os.makedirs(outdir, exist_ok=True)
    np.save(os.path.join(outdir,"features.npy"), self.reduced)
    with open(os.path.join(outdir,"pca.pkl"), 'wb') as f:
        pickle.dump(self.pca, f)
    with open(os.path.join(outdir,"meta.pkl"), 'wb') as f:
        pickle.dump(self.meta, f)
    with open(os.path.join(outdir, "video_names.pkl"), "wb") as f:
        pickle.dump(self.video_names, f)

    print("Model saved to", outdir)
```

Figure 7. save function (keep components in features.npy, pca.pkl, meta.pkl, and video_names.pkl)
Source: author's source code

By storing these components, the program enables fast query processing and accurate retrieval of the original video source and temporal/timestamp location for each matched frame.

## *B. Query Processing and Search Stage*

The query processing stage aims to determine the origin of a given query image by comparing it with preprocessed video frame representations stored in the model. This stage consists of model loading, query encoding, and similarity-based retrieval.

```
class Search:
    def __init__(self, model_dir, frame_size=(64,64)):
        self.model_dir = model_dir
        self.frame_size = frame_size
        self.features = np.load(os.path.join(model_dir, "features.npy"))
        with open(os.path.join(model_dir, "pca.pkl"), "rb") as f:
            self.pca = pickle.load(f)
        with open(os.path.join(model_dir, "meta.pkl"), "rb") as f:
            self.meta = pickle.load(f)
        with open(os.path.join(model_dir, "video_names.pkl"), "rb") as f:
            self.video_names = pickle.load(f)

        print("Loaded features:", self.features.shape)
        print("Loaded meta:", len(self.meta))
```

Figure 8. Search class structure
Source: author's source code

### *B.1 Model Loading and Initialization*

At initialization, the program loads all necessary model components generated during the preprocessing stage. By loading these components once at initialization, the program avoids repeated preprocessing and enables efficient query evaluation.

```
def load_query(self, path):
    img = cv2.imread(path)
    img = cv2.resize(img, self.frame_size)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    vec = img.flatten().astype("float32")
    return vec
```

Figure 9. load_query function
Source: author's source code

### *B.2 Query Image Preprocessing and Encoding*

The input query is provided in the form of a single image file. To ensure compatibility with the stored frame representations, the query image undergoes the same preprocessing pipeline used during model construction.

```
def encode_query(self, vec):
    z = self.pca.transform([vec]).astype("float32")
    return z
```

Figure 10. encode_query function (using load_query results)
Source: author's source code

After preprocessing, the query vector is projected into the PCA subspace using the trained PCA model. This step ensures that both the query image and video frames are in the same feature space, allowing direct similarity comparison.

### *B.3 Similarity Measurement and Retrieval*

Once the query has been encoded, similarity scores between the query vector and all stored video frame vectors are calculated using cosine similarity. This metric is suitable for high-dimensional feature spaces, as it focuses on the direction of vectors rather than their length.

The frames are then sorted in descending order based on their similarity scores. The top-*k* most similar frames are selected, and for each result, the program reports:

1. The corresponding video filename.
2. The timestamp within the video.
3. The similarity score.

```
def search(self, query_vec, topk=5):
    sims = cosine_similarity(query_vec, self.features)[0]
    idx = np.argsort(sims)[::-1][:topk]

    results = []
    for i in idx:
        vid_id, timestamp = self.meta[i]
        score = sims[i]
        filename = self.video_names[vid_id]
        results.append((filename, timestamp, score))

    return results
```

Figure 11. search function
Source: author's source code

This process allows the program to identify both which video and at what time the query image most likely appears.

## *C. Program Integration and Execution*

This section describes how the preprocessing stage (model building) and the query processing stage are integrated into a complete image-to-video search program. The main program coordinates model construction, query execution, and result presentation through a unified workflow.

### *C.1 Model Availability Check and Construction*

Before performing any search operation, the program verifies whether the preprocessed model already exists. The model is considered complete if the following files are present:

1. PCA-reduced feature matrix (features.npy)
2. PCA transformation model (pca.pkl)
3. Metadata mapping frame indices to timestamps

(meta.pkl)

4. Video filename mapping (video_names.pkl)

```python
def build_model_if_needed():
    needed_files = [
        "features.npy",
        "pca.pkl",
        "meta.pkl",
        "video_names.pkl"
    ]

    if all(os.path.exists(os.path.join(MODEL_DIR, f)) for f in needed_files):
        print("Model found. Skipping preprocessing.")
        return

    print("Model not found. Building model...\n")

    builder = ModelBuilder(video_dir=VIDEO_DIR, frame_size=(64,64), step=2, out_dim=64)
    builder.extract_frames()
    builder.reduce()
    builder.save(MODEL_DIR)

    print("\nModel built successfully.\n")
```

Figure 12. build_model_if_needed function
Source: author's source code

If all required files are found, the preprocessing stage will be skipped to avoid wasteful computation. Otherwise, the program will initiate the model-building process, which includes frame extraction, dimensionality reduction using PCA (SVD-based), and model serialization. This design significantly improves efficiency when multiple queries are executed.

### *C.2 Query Execution Pipeline*

For each query image, the program executes a standardized processing pipeline:

1. The query image is loaded and preprocessed using the same resizing and grayscale conversion applied during model construction.
2. The query vector is projected into the PCA feature space using the stored PCA model.
3. Cosine similarity is computed between the query vector and all stored video frame vectors.
4. The top-*k* most similar frames are selected and returned as search results.

```python
def format_time(seconds: float):
    m = int(seconds // 60)
    s = int(seconds % 60)
    return f"{m:02d}:{s:02d}"
```

Figure 13. format_time function
Source: author's source code

```python
def run_query(query_path):
    searcher = Search(MODEL_DIR)

    vec = searcher.load_query(query_path)
    encoded = searcher.encode_query(vec)
    results = searcher.search(encoded, topk=5)

    print("\n=== QUERY RESULT ===")
    print("Query image:", query_path)
    print("----------------------")

    for filename, ts, score in results:
        print(f" {filename:<25} | time {format_time(ts)} | sim={score:.4f}")
    print("----------------------\n")
```

Figure 14. run_query function
Source: author's source code

This pipeline ensures that both query images and video frames are represented consistently, enabling meaningful similarity comparisons.

### *C.3 Result Interpretation and Output Formatting*

The retrieved results are displayed in the command line interface with the following information:

1. Video filename.
2. Estimated timestamp in minute–second format.
3. Similarity score.

```python
if __name__ == "__main__":
    build_model_if_needed()
    run_query(QUERY1_PATH)
    run_query(QUERY2_PATH)
```

Figure 15. main.py (the executed functions)
Source: author's source code

The timestamp conversion from seconds to a readable format allows users to quickly locate the corresponding scene within the video. Multiple queries can be executed sequentially without rebuilding the model, proving the model's reusability.

## IV. Testing

The program is tested using 12 videos from One Punch Man Season 1, a Japanese animated series about Saitama, a hero capable of defeating any opponent with a single punch. All test videos originate from the same series in order to create a more challenging retrieval scenario, as many scenes share similar visual characteristics.

Two query images are selected for evaluation, as shown in Figure 16 and Figure 17.

Figure 16. Query image 1 (query1.png)
Source: author's local query

Figure 17. Query image 2 (query2.png)
Source: author's local query

When executing main.py, the program first checks whether a preprocessed model already exists. If not, the program performs frame extraction and model construction. The following outputs are shown in Figure 18 and Figure 19.

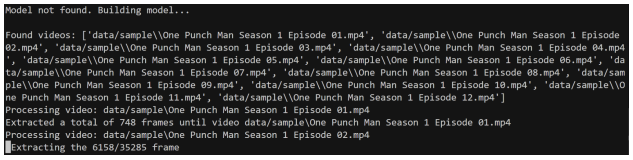


Figure 18. Program output (the start of model building)
Source: author's source code

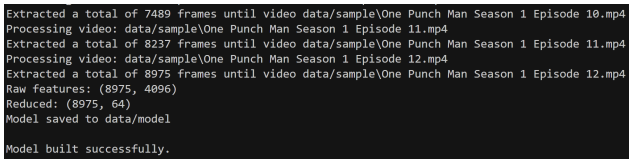


Figure 19. Program output (the end of model building)
Source: author's source code

After the model is built, the query images are processed. The retrieval results for the first and second queries are shown in Figure 20 and Figure 21.

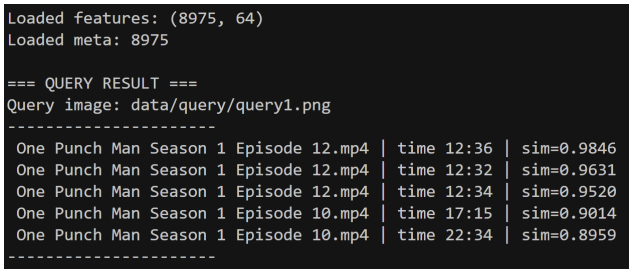


Figure 20. Retrieval result for query image 1
Source: author's source code


Figure 21. Retrieval result for query image 2
Source: author's source code

For the first query image, the top results are retrieved from One Punch Man Season 1 Episode 12 at timestamps around 12:32-12:36, with similarity scores above 0.95. This means that the system successfully found both the correct video and the approximate time of the scene.

For the second query image, the highest similarity scores are obtained from One Punch Man Season 1 Episode 05 around 16:24-16:26, with similarity scores above 0.98. Additional results from other episodes means that there are other visually similar scenes, but with lower similarity scores.

Overall, the testing results show that the image-to-video search program based on SVD is able to retrieve relevant video frames efficiently and accurately based on visual similarity.

## V. Conclusion

This study presents an image-to-video search system through a program that combines frame extraction, vectorization, SVD-based dimensionality reduction, and cosine similarity matching. Video frames are represented as grayscale pixel vectors and projected into a lower-dimensional feature space, enabling efficient comparison between query images and video frames.

Based on the testing results using videos from One Punch Man Season 1, the program is able to correctly retrieve the source video and approximate timestamp of the query images with high similarity scores. The use of PCA significantly reduces feature dimensionality while preserving important visual information, which allows cosine similarity to be applied effectively for matching.

However, the program still has several limitations. First, since raw grayscale pixel values are used as features, the retrieval performance may be reduced when the query image and video frames are different in scale or visual resolution. Second, the preprocessing stage requires extracting and encoding frames from all videos, which causes the model-building process to become increasingly time-consuming as the video duration or dataset size grows. Third, to reduce computational cost, each frame is represented as a flattened $64 \times 64$ grayscale vector (4096 dimensions) and frames are sampled at a fixed interval of one frame every two seconds. While this design improves efficiency, it may reduce retrieval accuracy by discarding some visual details and missing short transitional scenes.

For future work, the system can be improved by including more feature representations, such as deep learning-based embeddings, as well as applying temporal consistency to reduce false matches. Despite these limitations, the testing results show that the proposed approach is effective as a baseline image-to-video search system and demonstrates the practical application of SVD, PCA, and cosine similarity in visual search tasks.

## VI. Appendix

The source code for the image-to-video search program is available at https://github.com/RND815/SVD-based-Image-to-Video-Search.git.

The repository does not include video samples or query images, therefore, users are required to provide their own video files in the sample folder and query images (named query1.png and query2.png) in the query folder.

Further explanation and demonstration of the program can be accessed through the following video: https://youtu.be/RUu061iePYw?si=9K1Ben7_WwvDwI5E.

## VII. Acknowledgment

The author would like to express his sincere gratitude to Ir. Rila Mandala, M.Eng., Ph.D., as the lecturer of IF2123 Geometric and Linear Algebra for his valuable

guidance and dedication to spread his knowledge, which greatly contributes to the completion of this paper. The author would also like to thank all his friends and family members for their support throughout the process of researching and writing this paper.

## References

[1] R. Munir, “Tugas Besar 2: EigenPustaka: Sistem Perpustakaan Digital dengan Rekomendasi Semantik dan Pencarian Visual Berbasis SVD,” IF2123 Aljabar Geometri - Semester I Tahun 2025/2026. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2025-2026/Tubes2-IF2123-Algeo-2025.pdf. [Accessed: Dec. 10, 2025]

[2] R. Munir, “Singular Value Decomposition (SVD) (Bagian 1),” IF2123 Aljabar Geometri - Semester I Tahun 2025/2026. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2025-2026/Algeo-21-Singular-value-decomposition-Bagian1-2025.pdf. [Accessed: Dec. 12, 2025]

[3] R. Munir, “Singular Value Decomposition (SVD) (Bagian 2),” IF2123 Aljabar Geometri - Semester I Tahun 2025/2026. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2025-2026/Algeo-22-Singular-value-decomposition-Bagian2-2025.pdf. [Accessed: Dec. 12, 2025]

[4] T. Krantz and A. Jonker, “What Is Cosine Similarity?,” IBM Think. [Online]. Available: https://www.ibm.com/think/topics/cosine-similarity. [Accessed: Dec. 20, 2025].

[5] NumPy Developers, “numpy.linalg.svd,” NumPy reference. [Online]. Available: https://numpy.org/doc/2.3/reference/generated/numpy.linalg.svd.html. [Accessed: Dec. 20, 2025].

[6] J. Hui, “Machine Learning - Singular Value Decomposition (SVD) & Principal Component Analysis (PCA),” Medium. [Online]. Available: https://jonathan-hui.medium.com/machine-learning-singular-value-decomposition-svd-principal-component-analysis-pca-1d45e885e491. [Accessed: Dec. 20, 2025].

## Pernyataan

I hereby declare that the paper I have written is entirely my own work, and not a reproduction, adaptation, or translation of another individual's work. Furthermore, I declare that this paper is free from any form of plagiarism

Bandung, 24 December 2025

Reynard Anderson Wijaya (13524111)