Unpacking Complexity

A Look Back at Dimensionality Reduction

Minor In AI, IIT Ropar
30th May, 2025

Welcome, aspiring AI enthusiasts! Today, we're embarking on a journey to tackle one of the fundamental challenges in machine learning: dealing with vast amounts of data, sometimes described by an overwhelming number of characteristics or "dimensions." We'll learn how to simplify this complexity without losing the crucial insights.

Starting Our Journey: Where Are We Going?

Imagine you're planning a road trip. You open up a mapping application like Google Maps. Let's say your starting point is Hubballi and your destination is IIT Ropar.

Initially, you see the main route, perhaps the total distance, and estimated travel time by different modes like car or flight. But as you zoom in on the map, what else do you see? Suddenly, the map is filled with details! You see the names of smaller towns and cities along the way, highway numbers (like NH 548), toll points, restaurants, gas stations, shops, parks, water bodies, and even the names of local streets. The text might even change to the local language as you cross state borders!

All of this information is part of the map's data. The map contains *many* dimensions of information about the area it represents.

Now, think about your original goal: you just wanted to find a route from Hubballi to IIT Ropar. If you're planning a relaxed road trip and don't care about the fastest route, just the path itself and maybe places to stop when you're tired, a lot of the detailed information (like every single gas station or shop) might not be immediately relevant *to you right now*.

This brings us to a core idea: **Abstraction**. In computational thinking (which is more about *human-like thinking* applied to problems, not just thinking like a machine!), abstraction means focusing on the essential details while hiding or ignoring the irrelevant ones in a specific context. The map *has* all the details, but for your simple route-finding task, you only *need* a subset of that information.

This Google Maps scenario illustrates one way we deal with too much information: by selecting or focusing on the parts that are most relevant to our current task. This is a form of **Dimensionality Reduction**. We're reducing the number of 'dimensions' or types of information we are actively considering.

There's another way we might simplify information. Imagine you have to evaluate students for a 'Best Student' award. You might look at several criteria: their scores in tests, performance in projects, participation in discussions, completion of assignments, and oral test performance. Let's say each of these is graded on a scale of 1 to 100.

Instead of presenting a profile with five scores, you might decide to combine all these scores using a weighted formula (e.g., 20% from tests, 20% from projects, etc.) to produce a single, overall score for each student, say, out of 100. You then rank students based on this single score.

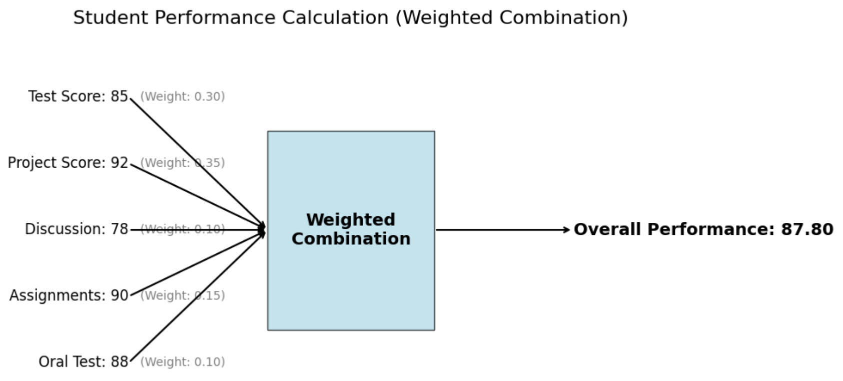

Figure 1: A graphic showing 5 inputs (Test Score, Project Score, Discussion, Assignments, Oral Test) going into a box labeled "Weighted Combination" and coming out as a single score (Overall Performance)

Here, you started with five dimensions (five different scores) and reduced it to just one dimension (the overall score). This is the second key way we perform dimensionality reduction: by *combining* multiple features into a smaller set of new features.

These two intuitions – selecting relevant features or combining features into a smaller representation – are at the heart of dimensionality reduction techniques in machine learning. They help us manage datasets that have a very large number of features.

Principal Component Analysis (PCA): Finding the Main Directions

One of the most widely used techniques for dimensionality reduction is **Principal Component Analysis (PCA)**. PCA falls under the second type of dimensionality reduction we discussed – it transforms your existing features into a *new, smaller set of features* called Principal Components.

Without diving deep into the complex mathematics involving covariance matrices, eigenvalues, and eigenvectors (which are fascinating but handled by libraries for us), the core idea of PCA is to find the directions (or components) in your data that capture the most **variance**. Variance tells us how spread out the data is in a particular direction. Directions with high variance usually contain more information about the data's structure.

Let's look at a simple synthetic 2D dataset. Imagine points scattered on a graph. The code below generates such data:

```python
# Code from lecture - Part 01 (Synthetic Data Generation and Plotting)
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

# Create synthetic 2D dataset
np.random.seed(0)
x = np.random.normal(0, 1, 100)
y = 2 * x + np.random.normal(0, 1, 100)
X = np.vstack((x, y)).T

# Plot original data (Conceptual - plot not shown here to save space)
# plt.scatter(X[:,0], X[:,1], alpha=0.5)
# plt.title("Original 2D Data")
# plt.xlabel("x")
# plt.ylabel("y")
# plt.axis('equal')
# plt.grid(True)
# plt.show()
```

Listing 1: Synthetic Data Generation and Plotting (Code for Original Data)

If you were to plot this original data, you'd see that the data points aren't just randomly scattered; they have a clear trend, almost like a line. They are much more spread out along this diagonal direction than perpendicular to it.

When we apply PCA to this 2D data and ask it to find two principal components, it identifies these directions of maximum variance. The first principal component (PC1) will align with the direction where the data is most spread out, and the second (PC2) will be perpendicular to the first, capturing the remaining variance.

```
# Code from lecture - Part 01 (PCA on Synthetic Data)
# (Assuming X is already defined from the previous code block)
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

print("Explained Variance Ratio:", pca.explained_variance_ratio_)
plt.scatter(X_pca[:,0], X_pca[:,1], alpha=0.5)
plt.title("Data after PCA")
plt.xlabel("PC1")
plt.ylabel("PC2")
plt.grid(True)
plt.axis('equal')
plt.show()
```

Listing 2: PCA on Synthetic Data and Plotting the Result

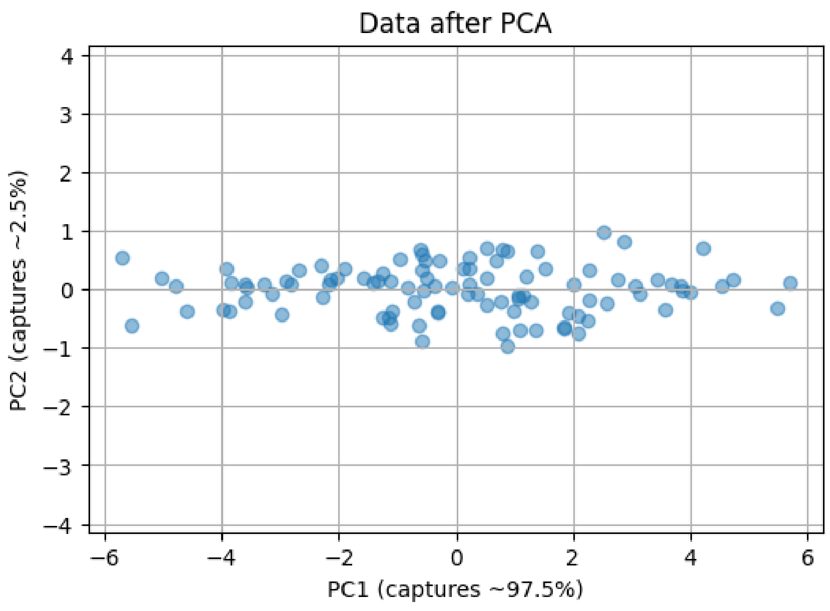

Figure 2: Scatter plot of the synthetic data after PCA, points now clustered tightly around the PC1 axis (the x-axis in the new plot), with PC2 spread much less

Look at the output printed by the code: `Explained Variance Ratio: [0.9748... 0.0251...]`.

This is a crucial insight! It tells us how much of the *total* variance (spread) in the original data is captured by each principal component.

PC1 captures about 97.5% of the variance, and PC2 captures about 2.5%. This confirms visually what we saw – almost all the interesting variation in the data lies along that main diagonal direction (which is now our PC1). The variation in the perpendicular direction (PC2) is very small.

This **explained variance ratio** is key for deciding how many components to keep. If the first component already captures 97.5% of the data's "information" (as measured by variance), do we really need the second component that only adds 2.5%? Probably not, especially if we wanted to reduce the dimensions significantly. In this case, even though we started with 2 dimensions, we could effectively represent most of the data's variance using just 1 principal component (PC1).

This is the beauty of PCA: it helps us understand which dimensions (the new principal components) are most important in representing the data's variability, allowing us to discard less important ones.

PCA in Action: What are the Trade-offs?

Now, let's see how applying PCA impacts real machine learning tasks. We'll use a few different datasets and models.

Case Study 1: Classifying Handwritten Digits

We'll use the Digits dataset, which contains small (8x8 pixel) images of handwritten digits from 0 to 9. Each image is represented by 64 features (one for each pixel). We want to train a model to classify these images. Since we have 10 distinct classes (0-9), this is a classification problem, and Logistic Regression is a suitable model.

Let's compare training a Logistic Regression model directly on the original 64 features versus training it on a reduced set of features obtained using PCA. We'll reduce the dimensions from 64 to 20 using `n_components=20`.

```python
# Code from lecture - Part 2 (Digits Dataset - Logistic Regression with/without PCA
    )
import numpy as np
from sklearn.datasets import load_digits
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Load dataset
digits = load_digits()
X = digits.data
y = digits.target

# Standardize features (important for PCA!)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# ----------------- Without PCA -----------------
# Split data BEFORE PCA for fair comparison
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, random_state=42) #
     Added random_state for consistency
clf = LogisticRegression(max_iter=1000)
clf.fit(X_train, y_train)
print("Accuracy without PCA:", accuracy_score(y_test, clf.predict(X_test)))

```

```python
# ----------------- With PCA -----------------
pca = PCA(n_components=20, random_state=42) # Added random_state
X_pca = pca.fit_transform(X_scaled) # Apply PCA on the WHOLE scaled dataset first
# Now split the PCA-transformed data
X_train_pca, X_test_pca, y_train_pca, y_test_pca = train_test_split(X_pca, y,
    random_state=42)
clf_pca = LogisticRegression(max_iter=1000) # Use a new classifier instance
clf_pca.fit(X_train_pca, y_train_pca)
print("Accuracy with PCA:", accuracy_score(y_test_pca, clf_pca.predict(X_test_pca))
    )
```

Listing 3: Digits Dataset - Logistic Regression with/without PCA

(Note: Code adjusted slightly from lecture for clarity and consistency in splitting after PCA and using separate classifier instances)

When we run this, we typically see an accuracy around 97-98% without PCA, and slightly lower, around 93-94%, with PCA (reducing to 20 components).

Reflection 1: Using PCA **might reduce your model's accuracy**. This happens because by reducing the number of dimensions, you *are* losing some information, even if it's the less important variance. The key question becomes: Is this loss in accuracy acceptable for your problem? If a 4-5% drop is too much, then PCA might not be suitable in this case, or you might need to keep more components.

Case Study 2: Predicting California Housing Prices

Next, let's look at the California Housing dataset, which has 8 features describing housing districts (like median income, house age, number of rooms, latitude, longitude) and a target variable: the median house value. Since we're trying to predict a continuous value, this is a regression problem, and Linear Regression is a basic model for this.

Here, we'll compare Linear Regression on the original 8 features versus using PCA to reduce the dimensions to 5. We'll also measure the time it takes to train the model, as reducing dimensions can sometimes speed things up. The error metric for regression is typically Mean Squared Error (MSE), where lower is better.

```python
# Code from lecture - Part 2 (California Housing Dataset - Linear Regression with/
    without PCA)
import numpy as np
import pandas as pd
import time
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.metrics import mean_squared_error

# Load dataset
data = fetch_california_housing()
X = pd.DataFrame(data.data, columns=data.feature_names)
y = pd.Series(data.target)

# Standardize features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# ----------------- Without PCA -----------------
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, random_state=42)

start_time_no_pca = time.time()
```

```python
model_no_pca = LinearRegression()
model_no_pca.fit(X_train, y_train)
end_time_no_pca = time.time()

y_pred_no_pca = model_no_pca.predict(X_test)
mse_no_pca = mean_squared_error(y_test, y_pred_no_pca)
training_time_no_pca = end_time_no_pca - start_time_no_pca

print("Without PCA:")
print("MSE:", mse_no_pca)
print("Training Time:", training_time_no_pca)

# ----------------- With PCA -----------------
pca = PCA(n_components=5, random_state=42)
X_pca = pca.fit_transform(X_scaled)
X_train_pca, X_test_pca, y_train_pca, y_test_pca = train_test_split(X_pca, y,
    random_state=42)

start_time_pca = time.time()
model_pca = LinearRegression()
model_pca.fit(X_train_pca, y_train_pca)
end_time_pca = time.time()

y_pred_pca = model_pca.predict(X_test_pca)
mse_pca = mean_squared_error(y_test_pca, y_pred_pca)
training_time_pca = end_time_pca - start_time_pca

print("With PCA:")
print("MSE:", mse_pca)
print("Training Time:", training_time_pca)
```

Listing 4: California Housing Dataset - Linear Regression with/without PCA

Running this code shows that the training time **decreases** with PCA (e.g., from ˜0.07s to ˜0.03s), but the MSE **increases** (meaning the error is higher).

Reflection 2: PCA can **reduce training time**, especially on larger datasets with many features, because the model has fewer dimensions to work with. However, similar to accuracy, this often comes at the cost of increased error (in regression) or decreased accuracy (in classification). You need to weigh the benefit of faster training against the penalty in performance.

Case Study 3: Classifying Breast Cancer Tumors

For our third PCA example, let's look at the Breast Cancer Wisconsin dataset. This contains features computed from images of cell nuclei to determine if a tumor is malignant (cancerous) or benign (not cancerous). This is a binary classification problem (two classes), perfect for Logistic Regression.

We'll again compare performance with and without PCA, this time reducing the features to 10.

```python
# Code from lecture - Part 2 (Breast Cancer Dataset - Logistic Regression with/
    without PCA)
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
import time

# Load data
data = load_breast_cancer()
X, y = data.data, data.target
```

```python

# Train/test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)

# Standardize features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# ---- Without PCA ----
start_time = time.time()
lr = LogisticRegression(max_iter = 1000, random_state=42) # Added random_state
lr.fit(X_train_scaled, y_train)
preds = lr.predict(X_test_scaled)
end_time = time.time()
accuracy_no_pca = accuracy_score(y_test, preds)
time_no_pca = end_time - start_time

# ---- With PCA ----
pca = PCA(n_components=10, random_state=42) # Added random_state
X_train_pca = pca.fit_transform(X_train_scaled)
X_test_pca = pca.transform(X_test_scaled)

start_time = time.time()
lr_pca = LogisticRegression(max_iter = 1000, random_state=42) # Added random_state
lr_pca.fit(X_train_pca, y_train)
preds_pca = lr_pca.predict(X_test_pca)
end_time = time.time()
accuracy_pca = accuracy_score(y_test, preds_pca)
time_pca = end_time - start_time

# ---- Results ----
print(f"Without PCA - Accuracy: {accuracy_no_pca:.4f}, Time: {time_no_pca:.4f}
    seconds")
print(f"With PCA    - Accuracy: {accuracy_pca:.4f}, Time: {time_pca:.4f} seconds")
```

Listing 5: Breast Cancer Dataset - Logistic Regression with/without PCA

(Note: Code adjusted slightly from lecture for `random_state` *consistency)*

Interestingly, in this case, when you run the code, you might find that the accuracy **increases** with PCA (e.g., from ˜96% to ˜98%) *and* the training time **decreases**.

Reflection 3: While PCA often introduces a trade-off, there are cases, particularly with noisy datasets or highly correlated features, where reducing dimensions can act like a **denoising** step. By focusing on the principal components that capture the most significant variance, PCA might effectively filter out noise or redundant information in less important components, leading to improved accuracy and reduced training time simultaneously.

So, the decision to use PCA isn't always straightforward. It depends on the dataset, the model, and your priorities (accuracy vs. speed). You need to experiment and evaluate the results.

TSNE: Seeing Your Data in 2D

PCA is great for reducing dimensions for model training, but what if your goal isn't training, but simply to *understand* and *visualize* your high-dimensional data? Imagine you have data with many features, and you want to see if data points of different classes naturally cluster together in a 2D plot. You can't plot 10, 20, or 64 dimensions easily.

This is where another dimensionality reduction technique called **t-Distributed Stochastic Neighbor Embedding (t-SNE)** comes in. Unlike PCA, t-SNE is **primarily for visualization**. Its goal is to map high-dimensional data points to a low-dimensional space (usually 2D or 3D) in a way that preserves the *local relationships* between points. Points that were close together in the high-dimensional space are likely to be close together in the t-SNE plot.

Let's use the famous Iris dataset. It has 4 features (sepal length/width, petal length/width) and three classes of iris flowers (setosa, versicolor, virginica).

A standard way to visualize a dataset with a few features is using a pairwise scatter plot matrix, showing scatter plots for every combination of two features. The code to generate such a plot is:

```python
# Code from lecture - IRIS Data Set Visualization (Pairplot)
from sklearn.datasets import load_iris
import seaborn as sns
import pandas as pd
import matplotlib.pyplot as plt

# Load dataset
iris = load_iris()
X = iris.data
y = iris.target
feature_names = iris.feature_names
target_names = iris.target_names

# Create DataFrame including labels
df = pd.DataFrame(X, columns=feature_names)
df['Species'] = [target_names[i] for i in y]

# Plot pairwise scatterplots (Conceptual - plot not shown here)
# sns.pairplot(df, hue='Species', palette='Set2', markers=["o", "s", "D"])
# plt.suptitle("Pairwise Scatterplots of Iris Dataset Features", y=1.02)
# plt.show()
```

Listing 6: IRIS Data Set Visualization (Pairplot - Code Only)

This plot, if generated, would be informative but can be a matrix of many plots (e.g., 16 plots for 4 features). If you had 10 features, you'd have 100 plots. It's hard to get an overall picture of how the data is structured across all dimensions simultaneously.

Now, let's use t-SNE to reduce the 4 dimensions of the Iris dataset to 2 dimensions and plot the result:

```python
# Code from lecture - t-SNE on Iris Dataset
from sklearn.datasets import load_iris
from sklearn.manifold import TSNE
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

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

# Standardize the features (good practice for many algorithms, including t-SNE)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Apply t-SNE
tsne = TSNE(n_components=2, random_state=42, perplexity = 30, max_iter = 1000)
```

```python
X_embedded = tsne.fit_transform(X_scaled)

# Create DataFrame for plotting
df_tsne = pd.DataFrame()
df_tsne["Component 1"] = X_embedded[:, 0]
df_tsne["Component 2"] = X_embedded[:, 1]
df_tsne["Label"] = [target_names[i] for i in y]

# Plot the 2D t-SNE result
plt.figure(figsize=(8, 6))
sns.scatterplot(data=df_tsne, x="Component 1", y="Component 2", hue="Label",
    palette="Set2", s=100)
plt.title("t-SNE on Iris Dataset")
plt.show()
```

Listing 7: t-SNE on Iris Dataset

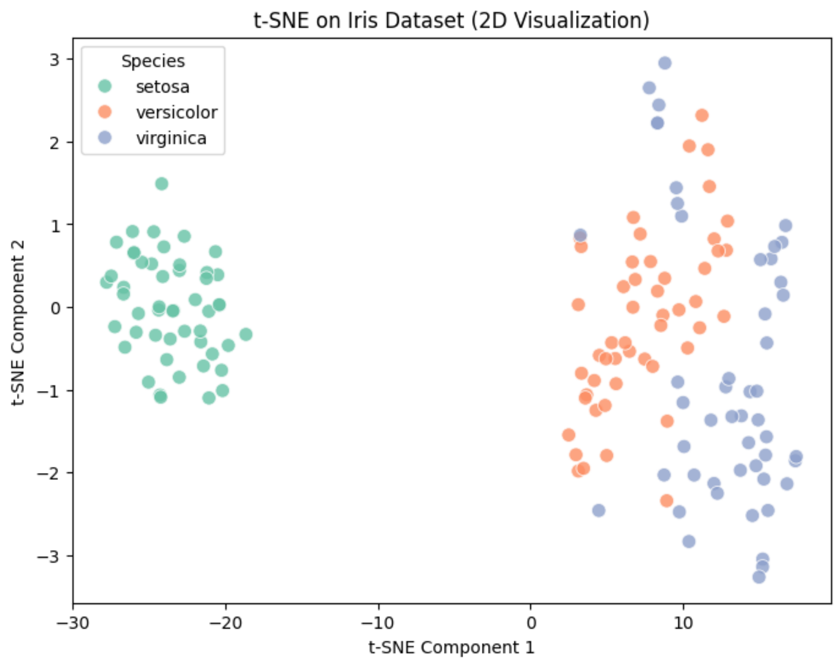

Figure 3: 2D Scatter plot of the Iris dataset after t-SNE, showing three distinct clusters of points colored by species

Look at this plot! The three species, which were somewhat mixed in the pairwise plots (if you were to view them), are now clearly separated into distinct clusters in this single 2D visualization. t-SNE has done a great job of showing the underlying structure of the data in a way we can easily understand visually.

Reflection 4: Use **t-SNE specifically for visualizing high-dimensional data** to explore clusters or structure. **Do not use it to reduce dimensions for training a model** when your goal is better accuracy or faster training time, as PCA is generally more suitable for that purpose and its components are interpretable in terms of variance.

The `perplexity` parameter in t-SNE is roughly like telling the algorithm how many ”neighbors” each point should consider when arranging itself in the low-dimensional space. It influences how t-SNE balances focusing on local structure versus global structure. A common range for perplexity is between 5 and 50.

Wrapping Up: Your Dimensionality Reduction Toolkit

You've now taken your first steps into the world of dimensionality reduction! We started with simple intuitions inspired by everyday tasks like using maps and evaluating students. We then explored **PCA**, a powerful technique to transform data into a smaller set of principal components based on variance, and saw how it can impact model accuracy and training time. Finally, we introduced **t-SNE**, a specialized tool perfect for visualizing complex datasets in a simple 2D or 3D space.

Remember these key takeaways:

- Dimensionality reduction helps handle datasets with many features by selecting or combining them.
- **PCA** is great for reducing dimensions for machine learning model training, potentially improving speed but sometimes affecting accuracy. You need to evaluate the trade-offs.
- **t-SNE** is primarily for visualizing high-dimensional data to understand its structure and clustering.
- The math behind these techniques (like eigenvalues for PCA or probability distributions for t-SNE) is complex, but libraries like scikit-learn handle it for you. Your job is to understand *what* they do and *when* to use them effectively.

As you continue your AI journey, you'll encounter many datasets where dimensionality reduction is essential. Experiment with these tools, observe their impact, and build your intuition on when and how to apply them.

Happy exploring, and see you in the next chapter!