Nilesh Barla | 10 min | 8th August, 2023

ML Model Development

Data forms the foundation of any machine learning algorithm, without it, Data Science can not happen. Sometimes, it can contain a huge number of features, some of which are not even required. Such redundant information makes modeling complicated. Furthermore, interpreting and understanding the data by visualization gets difficult because of the high dimensionality. This is where dimensionality reduction comes into play.

In this article you will learn:

1. What is dimensionality reduction?
2. What is the curse of dimensionality?
3. Tools and libraries used for dimensionality reduction
4. Algorithms used for dimensionality reduction
5. Applications
6. Advantages and disadvantages

What is dimensionality reduction?

Dimensionality reduction is the task of reducing the number of features in a dataset. In machine learning tasks like regression or classification, there are often too many variables to work with. These variables are also called **features**. The higher the number of features, the more difficult it is to model them, this is known as the **curse of**

The process of dimensionality reduction essentially transforms data from high-dimensional feature space to a low-dimensional feature space. Simultaneously, it is also important that meaningful properties present in the data are not lost during the transformation.

Dimensionality reduction is commonly used in data visualization to understand and interpret the data, and in machine learning or deep learning techniques to simplify the task at hand.

Curse of dimensionality

It is well known that ML/DL algorithms need a large amount of data to learn invariance, patterns, and representations. If this data comprises a large number of features, this can lead to the curse of dimensionality. The curse of dimensionality, first introduced by Bellman, describes that in order to estimate an arbitrary function with a certain accuracy the number of features or dimensionality required for estimation grows exponentially. This is especially true with big data which yields more **sparsity**.

Tools and library

The most popular library for dimensionality reduction is **scikit-learn** (sklearn). The library consists of three main modules for dimensionality reduction algorithms:

1. Decomposition algorithms
   - Principal Component Analysis
   - Kernel Principal Component Analysis
   - Non-Negative Matrix Factorization
   - Singular Value Decomposition
2. Manifold learning algorithms
   - t-Distributed Stochastic Neighbor Embedding
   - Spectral Embedding
   - Locally Linear Embedding
3. Discriminant Analysis
   - Linear Discriminant Analysis

1. Decomposition algorithms
    - Principal Component Analysis
    - Kernel Principal Component Analysis
    - Non-Negative Matrix Factorization
    - Singular Value Decomposition
2. Manifold learning algorithms
    - t-Distributed Stochastic Neighbor Embedding
    - Spectral Embedding

Decomposition algorithms

Decomposition algorithm in scikit-learn involves dimensionality reduction algorithms. We can call various techniques using the following command:

```
from sklearn.decomposition import PCA, KernelPCA, NMF
```

Principal Component Analysis (PCA)

Principal Component Analysis, or PCA, is a dimensionality-reduction method to find lower-dimensional space by preserving the **variance** as measured in the high dimensional input space. It is an unsupervised method for dimensionality reduction.

PCA transformations are linear transformations. It involves the process of finding the principal components, which is the decomposition of the feature matrix into eigenvectors. This means that PCA will not be effective when the distribution of the dataset is non-linear.

Let's understand PCA with python code.


```
from sklearn.decomposition import PCA, KernelPCA, NMF
```

Principal Component Analysis (PCA)

Principal Component Analysis, or PCA, is a dimensionality-reduction method to find lower-dimensional space by preserving the **variance** as measured in the high dimensional input space. It is an unsupervised method for dimensionality reduction.

PCA transformations are linear transformations. It involves the process of finding the principal components, which is the decomposition of the feature matrix into eigenvectors. This means that PCA will not be effective when the distribution of the dataset is non-linear.

Let's understand PCA with python code.

structure and the property of the data that would be principal components. The **eigenvalues** on the other hand help us to determine the principal components. The highest eigenvalues and their corresponding eigenvectors make the most important principal components.

4. **Final output**: It is the dot product of the standardized matrix and the eigenvector. Note that the number of columns or features will be changed.

Reducing the number of variables of data not only reduces complexity but also decreases the accuracy of the machine learning model. However, with a smaller number of features it is easy to explore, visualize and analyze, it also makes machine learning algorithms computationally less expensive. In simple words, the idea of PCA is to reduce the number of variables of a data set, while preserving as much information as possible.

Let's also take a look at the modules and functions sklearn provides for PCA.

We can start by loading the most dataset:

```
from sklearn.datasets import load_digits
digits = load_digits()
digits.data.shape
```

```
digits.data.shape
```

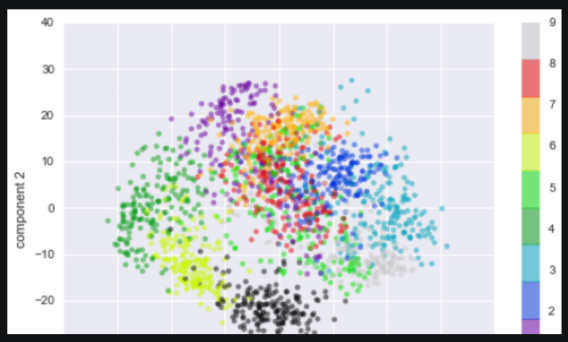
machine learning model. However, with a smaller number of features it is easy to explore, visualize and analyze, it also makes machine learning algorithms computationally less expensive. In simple words, the idea of PCA is to reduce the number of variables of a data set, while preserving as much information as possible.

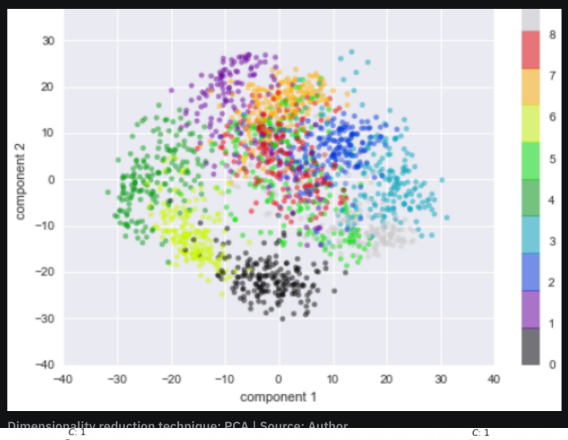
Let's also take a look at the modules and functions sklearn provides for PCA.

We can start by loading the most dataset:

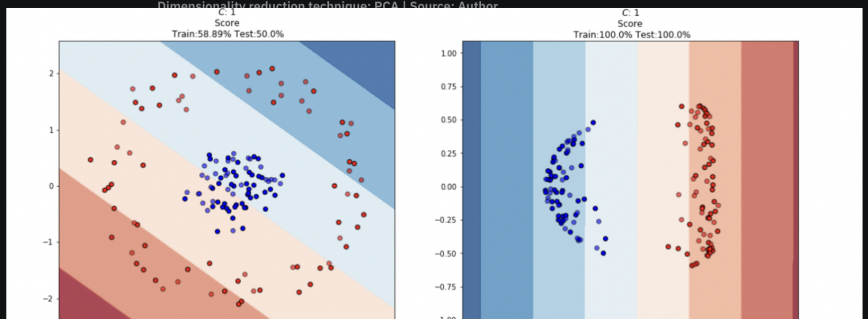
```
from sklearn.datasets import load_digits
digits = load_digits()
digits.data.shape
```

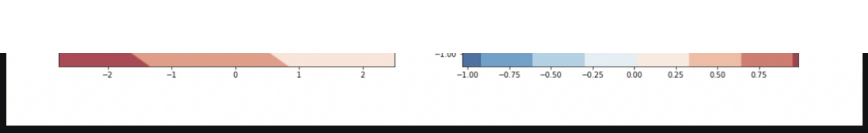
```
plt.scatter(projected[:, 0], projected[:, 1],
            c=digits.target, edgecolor='none', alpha=0.5,
            cmap=plt.cm.get_cmap('spectral', 10))
plt.xlabel('component 1')
plt.ylabel('component 2')
plt.colorbar();
```





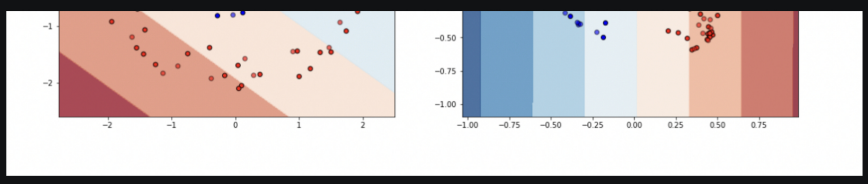
Dimensionality reduction technique: PCA | Source: Author





Dimensionality reduction technique: KPCA | Source: Author

Kernel PCA uses a kernel function ϕ that calculates the dot product of the data for non-linear mapping. In other words, the function ϕ maps the original d-dimensional features into a larger, k-dimensional feature space by



Dimensionality reduction technique: KPCA | Source: Author

Kernel PCA uses a kernel function ϕ that calculates the dot product of the data for non-linear mapping. In other words, the function ϕ maps the original d-dimensional features into a larger, k-dimensional feature space by creating non-linear combinations of the original features.

Let assume a dataset x that contains two features x1 and x2:

$$\mathbf{x} = \begin{bmatrix} x_1 & x_2 \end{bmatrix}^T \qquad \mathbf{x} \in I\!R^d$$

After applying the kernel trick we get:

$$\mathbf{x}' = \begin{bmatrix} x_1 & x_2 & x_1 x_2 & x_1^2 & x_1 x_2^3 & \dots \end{bmatrix}^T \qquad \mathbf{x} \in I\!R^k (k >> d)$$

To get a more intuitive understanding of Kernel PCA let's define a feature space that cannot be linearly separated.

```
from sklearn.datasets import make_circles
from sklearn.decomposition import KernelPCA
np.random.seed(0)
X, y = make_circles(n_samples=400, factor=.3, noise=.05)
```

```
X, y = make_circles(n_samples=400, factor=.3, noise=.05)
```

Now, let's plot and see our dataset.

```
plt.figure(figsize=(15,10))
plt.subplot(1, 2, 1, aspect='equal')
plt.title("Original space")
reds = y == 0
blues = y == 1

plt.scatter(X[reds, 0], X[reds, 1], c="red",
            s=20, edgecolor='k')
plt.scatter(X[blues, 0], X[blues, 1], c="blue",
            s=20, edgecolor='k')
```

Dimensionality reduction technique: KPCA | Source: Author

As you can see in this dataset the two classes cannot be separated linearly. Now, let's define kernel PCA and see how it separates this feature space.

```
kpca = KernelPCA(kernel="rbf", fit_inverse_transform=True, gamma=10, )
X_kpca = kpca.fit_transform(X)
plt.subplot(1, 2, 2, aspect='equal')
plt.scatter(X_kpca[reds, 0], X_kpca[reds, 1], c="red",
            s=20, edgecolor='k')
plt.scatter(X_kpca[blues, 0], X_kpca[blues, 1], c="blue",
            s=20, edgecolor='k')
plt.title("Projection by KPCA")
plt.xlabel(r"1st principal component in space induced by $phi$")
plt.ylabel("2nd component")
```

```
kpca = KernelPCA(kernel="rbf", fit_inverse_transform=True, gamma=10, )
X_kpca = kpca.fit_transform(X)
plt.subplot(1, 2, 2, aspect='equal')
plt.scatter(X_kpca[reds, 0], X_kpca[reds, 1], c="red",
            s=20, edgecolor='k')
plt.scatter(X_kpca[blues, 0], X_kpca[blues, 1], c="blue",
            s=20, edgecolor='k')
plt.title("Projection by KPCA")
plt.xlabel(r"1st principal component in space induced by $phi$")
plt.ylabel("2nd component")
```

After applying KPCA, it is able to linearly separate the two classes in the dataset.

Singular Value Decomposition (SVD)

The singular value decomposition or SVD is a factorization method of a real or complex matrix. It is efficient when working with a sparse dataset; a dataset having a lot of zero entries. This type of dataset is usually found in the Recommender Systems, rating, and reviews dataset, et cetera.

The idea of SVD is that every matrix of shape nXp factorizes into $A = USV^T$, where U is the orthogonal matrix, S is a diagonal matrix and V^T is also an orthogonal matrix.

$$\mathbf{A}_{nxp} = \mathbf{U}_{nxn}\ \mathbf{S}_{nxp}\ \mathbf{V}^{\mathbf{T}}_{pxp}$$

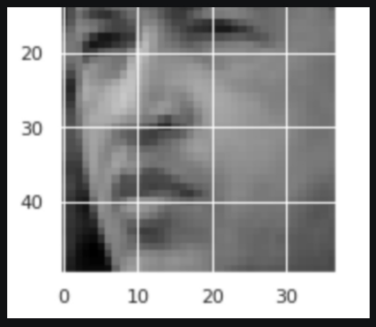
The advantage of SVD is that the orthogonal matrices capture the structure of the original matrix A which means that their properties do not change when multiplied by other numbers. This can help us approximate A.

Now let's understand SVD using code. To get a better understanding of the algorithm we will use a face dataset that scikit-learn provides

when working with a sparse dataset; a dataset having a lot of zero entries. This type of dataset is usually found in the Recommender Systems, rating, and reviews dataset, et cetera.

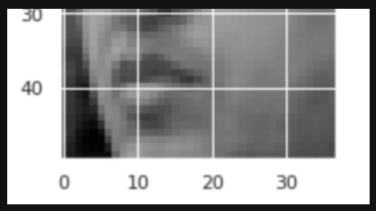
The idea of SVD is that every matrix of shape nXp factorizes into $A = USV^T$, where U is the orthogonal matrix, S is a diagonal matrix and V^T is also an orthogonal matrix.

$$\mathbf{A}_{nxp} = \mathbf{U}_{nxn}\ \mathbf{S}_{nxp}\ \mathbf{V}^{\mathbf{T}}_{pxp}$$



Create a function for easy visualization of images.

```
def draw_img(img_vector, h=img_height, w=img_width):
    plt.imshow( img_vector.reshape((h,w)), cmap=plt.cm.gray)
    plt.xticks(())
    plt.yticks(())
```



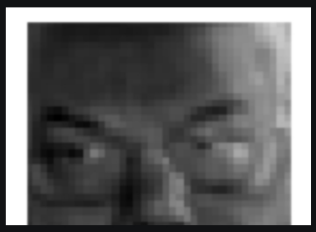
Create a function for easy visualization of images.

Before applying SVD it is better to standardize the data.

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler(with_std=False)
Xstd = scaler.fit_transform(X)
```

After standardizing this is how the image looks.



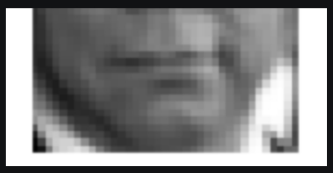
```
scaler = StandardScaler(with_std=False)
Xstd = scaler.fit_transform(X)
```

After standardizing this is how the image looks.

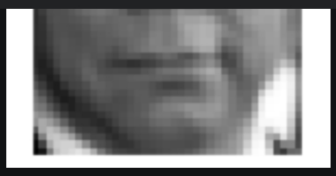
Now, we can apply the SVD function from NumPy and decompose the matrix into three matrices.

```
from numpy.linalg import svd

U, S, VT = svd(Xstd)
```

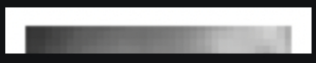


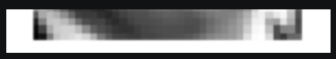
Now, we can apply the SVD function from NumPy and decompose the matrix into three matrices.



Now, let's perform dimensionality reduction. To do that we just need to reduce the number of features from the orthogonal matrices.

```
Xhat_500 = US[:, 0:500] @ VT[0:500, :]
# inverse transform Xhat to reverse standardization
Xhat_500_orig = scaler.inverse_transform(Xhat_500)
# draw recovered image
draw_img(Xhat_500_orig[49])
```

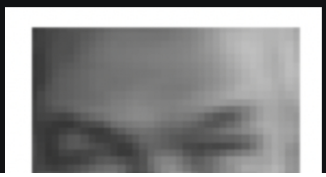


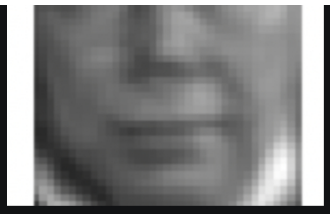


Now, let's perform dimensionality reduction. To do that we just need to reduce the number of features from the orthogonal matrices.

```
Xhat_500 = US[:, 0:500] @ VT[0:500, :]
# inverse transform Xhat to reverse standardization
Xhat_500_orig = scaler.inverse_transform(Xhat_500)
# draw recovered image
draw_img(Xhat_500_orig[49])
```

```
# inverse transform Xhat to reverse standardization
Xhat_100_orig = scaler.inverse_transform(Xhat_100)
# draw recovered image
draw_img(Xhat_100_orig[49])
```





```
    plt.subplot(1, len(dim_vec), i + 1)
    draw_img(dim_reduce(US, VT, d)[49])
```

Dimensionality reduction technique: SVD | Source: Author

As you can see the first image contains the least number of features yet it can still construct the abstract version of the image and as we increase the features, we eventually obtain the original image. This proves that SVD can retain the basic structure of the data.

Dimensionality reduction technique: SVD | Source: Author

As you can see the first image contains the least number of features yet it can still construct the abstract version of the image and as we increase the features, we eventually obtain the original image. This proves that SVD can retain the basic structure of the data.

Non-negative Matrix Factorization (NMF)

NMF is an unsupervised machine learning algorithm. When a non-negative input matrix X of dimension mXn is given to the algorithm, it is decomposed into the product of two non-negative matrices W and H. W is of the

NMF is good for parts-based representation of the dataset i.e. NMF provides an **efficient**, **distributed representation,** and can aid in the discovery of the structure of interest within the data.

Let's understand NMF with code. We will use the same data that we used in SVD.

First, we will fit the model to the data.

```
from sklearn.decomposition import NMF
model = NMF(n_components=200, init='nndsvd', random_state=0)
W = model.fit_transform(X)
V = model.components_
```

NMF takes a bit of time to decompose the data. Once the data is decomposed we can then visualize the factorized components.

```
num_faces = 20
plt.figure(figsize=(1.8 * 5, 2.4 * 4))
```


```
from sklearn.decomposition import NMF
model = NMF(n_components=200, init='nndsvd', random_state=0)
W = model.fit_transform(X)
V = model.components_
```

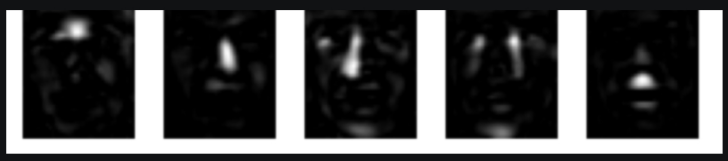
NMF takes a bit of time to decompose the data. Once the data is decomposed we can then visualize the factorized components.

Dimensionality reduction technique: NMF | Source: Author

From the image above we can see that NMF is very efficient to capture the underlying structure of the data. It is also worth mentioning that NMF captures only the linear attributes.

Advantages of NMF:

1. Data compression and visualization



Dimensionality reduction technique: NMF | Source: Author

From the image above we can see that NMF is very efficient to capture the underlying structure of the data. It is also worth mentioning that NMF captures only the linear attributes.

Advantages of NMF:

1. Data compression and visualization

2. Robustness to noise

```
from sklearn.manifold import TSNE, LocallyLinearEmbedding, SpectralEmbedding
```

t-Distributed Stochastic Neighbor Embedding (t-SNE)

t-Distributed Stochastic Neighbor Embedding or t-SNE is a dimensionality reduction technique well suited for data visualization. Unlike PCA which simply maximizes the variance, t-SNE minimizes the divergence between

data visualization. Unlike PCA which simply maximizes the variance, t-SNE minimizes the divergence between two distributions. Essentially, it recreates the distribution of a high-dimensional space in a low-dimensional space rather than maximizing variance or even using a kernel trick.

We can get a high-level understanding of t-SNE in three simple steps:

1. It first creates a probability distribution for the high-dimensional samples.
2. Then, it defines a similar distribution for the points in the low-dimensional embedding.
3. Finally, it tries to minimize the KL-divergence between the two distributions.

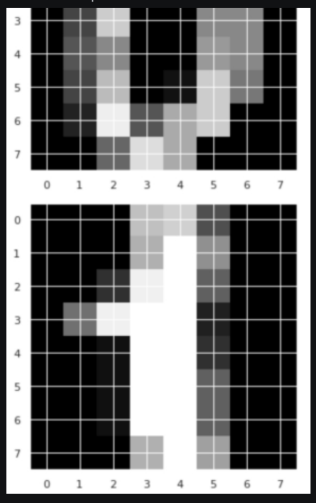
Now let's understand it with code. For t-SNE, we will use the MNIST dataset again. Firstly, we import TSNE and then the data as well.

t-Distributed Stochastic Neighbor Embedding (t-SNE)

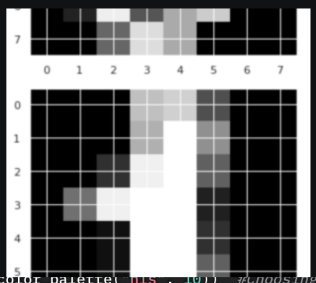
t-Distributed Stochastic Neighbor Embedding or t-SNE is a dimensionality reduction technique well suited for data visualization. Unlike PCA which simply maximizes the variance, t-SNE minimizes the divergence between two distributions. Essentially, it recreates the distribution of a high-dimensional space in a low-dimensional space rather than maximizing variance or even using a kernel trick.

We can get a high-level understanding of t-SNE in three simple steps:

1. It first creates a probability distribution for the high-dimensional samples.
2. Then, it defines a similar distribution for the points in the low-dimensional embedding.



Dimensionality reduction technique: t-SNE |



```
palette = np.array(sb.color_palette("hls", 10))  #Choosing color palette

# Create a scatter plot.
f = plt.figure(figsize=(8, 8))
ax = plt.subplot(aspect='equal')
sc = ax.scatter(x[:,0], x[:,1], lw=0, s=40,c=palette[colors.astype(np.int)])
# Add the labels for each digit.
txts = []
for i in range(10):
    # Position of each label.
    xtext, ytext = np.median(x[colors == i, :], axis=0)
    txt = ax.text(xtext, ytext, str(i), fontsize=24)
    txt.set_path_effects([pe.Stroke(linewidth=5, foreground="w"),
                          pe.Normal()])
```

```
                                    pe.Normal()])
        txts.append(txt)
    return f, ax, txts
```

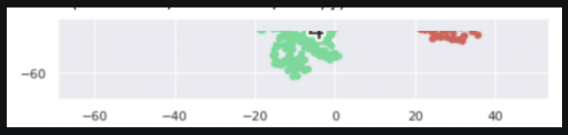
Now we perform data visualization on the transformed dataset.

```
plot(digits_final,Y)
```

```
        xtext, ytext = np.median(x[colors == i, :], axis=0)
        txt = ax.text(xtext, ytext, str(i), fontsize=24)
        txt.set_path_effects([pe.Stroke(linewidth=5, foreground="w"),
                              pe.Normal()])
        txts.append(txt)
    return f, ax, txts
```

Now we perform data visualization on the transformed dataset.

```
plot(digits_final,Y)
```



Dimensionality reduction technique: t-SNE | Source: Author

As it can be seen, t-SNE clusters the data beautifully. Compared to PCA, t-SNE performs well on nonlinear data. The drawback with t-SNE is that when the data is big it consumes a lot of time. So it is better to perform PCA followed by t-SNE.

Locally Linear Embedding (LLE)

Locally Linear Embedding or LLE is a non-linear and unsupervised machine learning method for dimensionality reduction. LLE takes advantage of the local structure or topology of the data and preserves it on a lower-dimensional feature space.

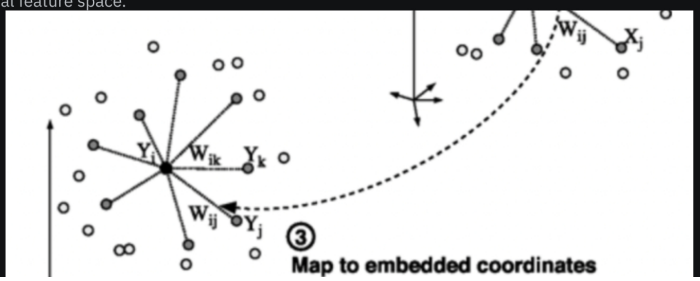
LLE optimizes faster but fails on noisy data.

Let's break the whole process into three simple steps:

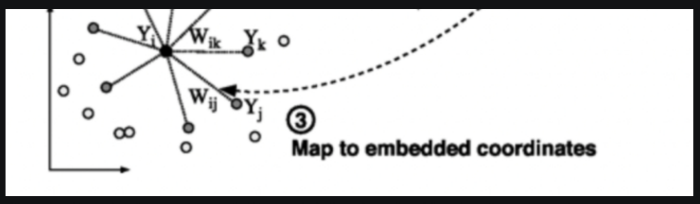
As it can be seen, t-SNE clusters the data beautifully. Compared to PCA, t-SNE performs well on nonlinear data. The drawback with t-SNE is that when the data is big it consumes a lot of time. So it is better to perform PCA followed by t-SNE.

Locally Linear Embedding (LLE)

Locally Linear Embedding or LLE is a non-linear and unsupervised machine learning method for dimensionality reduction. LLE takes advantage of the local structure or topology of the data and preserves it on a lower-dimensional feature space.



Dimensionality reduction technique: LLE | Source: S. T. Roweis and L. K. Saul, Nonlinear dimensionality reduction by locally linear embedding



Dimensionality reduction technique: LLE | Source: S. T. Roweis and L. K. Saul, Nonlinear dimensionality reduction by locally linear embedding

We can again break the whole process into three simple steps:

1. **Preprocessing**: Construct a Laplacian matrix representation of the data or graph.
2. **Decomposition**: Compute eigenvalues and eigenvectors of the constructed matrix and then map each point to a lower-dimensional representation. Spectral embedding makes use of the second smallest eigenvalue and its corresponding eigenvector.
3. **Clustering**: Assign points to two or more clusters, based on the representation. Clustering is usually done using k-means clustering.

Applications: Spectral Embedding finds its application in image segmentation.

Discriminant Analysis

Discriminant Analysis is another module that scikit-learn provides. It can be called using the following command:

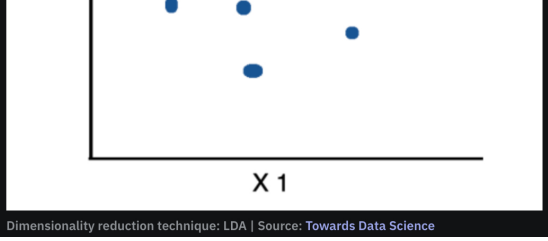
```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
```

and its corresponding eigenvector.
3. **Clustering**: Assign points to two or more clusters, based on the representation. Clustering is usually done using k-means clustering.

Applications: Spectral Embedding finds its application in image segmentation.

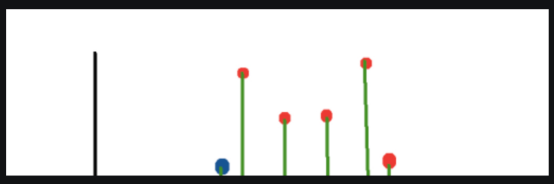
Discriminant Analysis

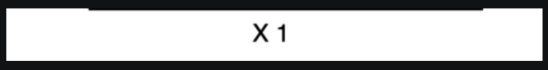
Discriminant Analysis is another module that scikit-learn provides. It can be called using the following command:



Dimensionality reduction technique: LDA | Source: Towards Data Science

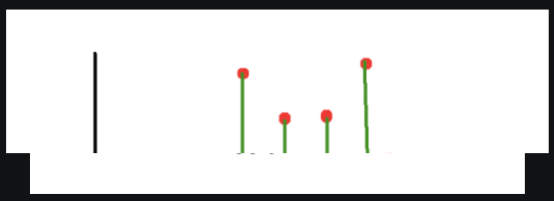
One way to solve this problem is to project all the data points in the x-axis.





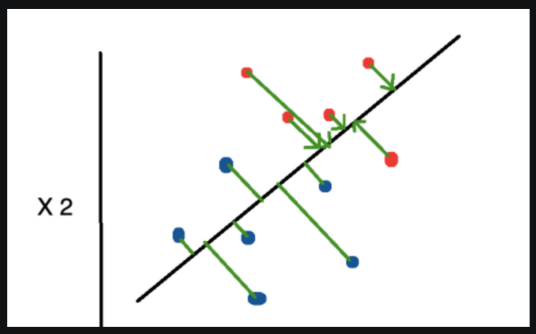
Dimensionality reduction technique: LDA | Source: Towards Data Science

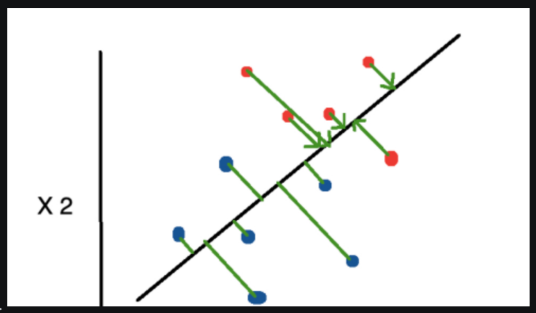
One way to solve this problem is to project all the data points in the x-axis.



Dimensionality reduction technique: LDA | Source: Towards Data Science

A better approach will be to compute the distance between all the points in the data and fit a new linear line that passes through them. This new line can now be used to project all the points.





the following 5 steps:

1. Compute the d-dimensional mean vectors for the different classes from the dataset.
2. Compute the scatter matrices (in-between-class and within-class scatter matrices). the Scatter matrix is used to make estimates of the covariance matrix. This is done when the covariance matrix is difficult to calculate or joint variability of two random variables is difficult to calculate.
3. Compute the eigenvectors (e1, e2, e3, ..ed) and corresponding eigenvalues (λ1,λ2, ...λd) for the scatter

matrices.

4. Sort the eigenvectors by decreasing eigenvalues and choose k eigenvectors with the largest eigenvalues to form a d×k dimensional matrix W (where every column represents an eigenvector).
5. Use this d×k eigenvector matrix to transform the samples onto the new subspace. This can be summarized by the matrix multiplication: Y=X×W (where X is an n×d dimensional matrix representing the n samples, and y are the transformed n×k-dimensional samples in the new subspace).

To know about LDA you can check out this article.

Applications of dimentionality reduction

3. Compute the eigenvectors (e1, e2, e3...ed) and corresponding eigenvalues (λ1,λ2,...,λd) for the scatter matrices.
4. Sort the eigenvectors by decreasing eigenvalues and choose k eigenvectors with the largest eigenvalues to form a d×k dimensional matrix W (where every column represents an eigenvector).
5. Use this d×k eigenvector matrix to transform the samples onto the new subspace. This can be summarized by the matrix multiplication: Y=X×W (where X is an n×d dimensional matrix representing the n samples, and y are the transformed n×k-dimensional samples in the new subspace).

To know about LDA you can check out this article.

Applications of dimentionality reduction

- It also helps remove redundant features and noise.
- It tackles the curse of dimensionality

Disadvantages of dimensionality reduction:

- It may lead to some amount of data loss.
- Accuracy is compromised.

Final thoughts

In this article, we learned about dimensionality reduction and also about the curse of dimensionality. We touched on the different algorithms that are used in dimensionality reduction with mathematical details and through code as well.

It is worth mentioning these algorithms are supposed to be used based on the task at hand. For instance, if the nature of your data is linear then use decomposition methods otherwise use manifold learning techniques.

It is considered to be a good practice to first visualize the data and then decide which method to use. Also, do

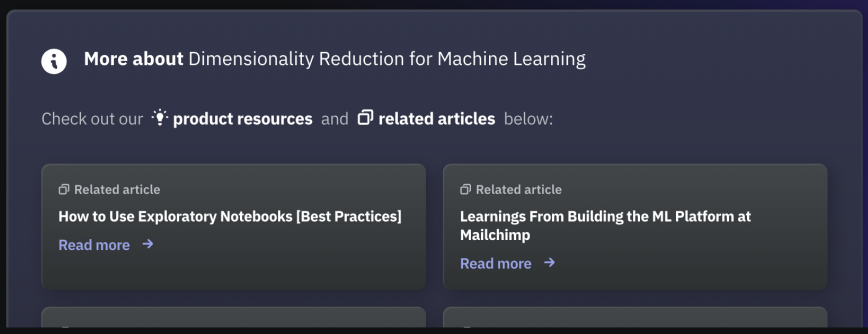
- Accuracy is compromised.

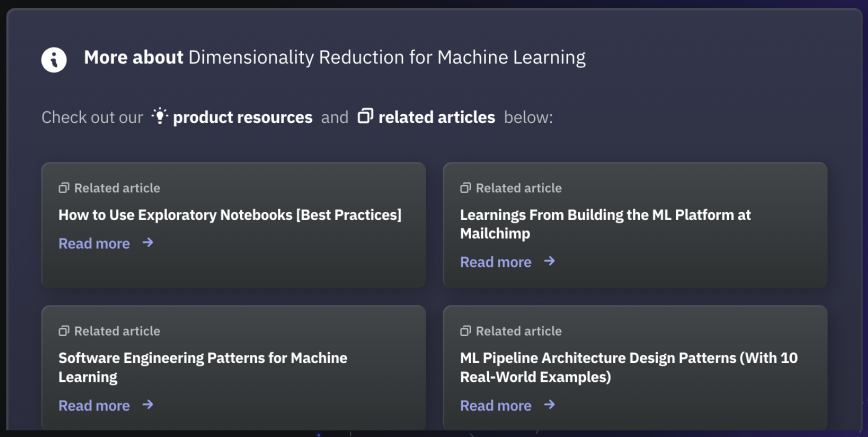
Final thoughts

In this article, we learned about dimensionality reduction and also about the curse of dimensionality. We touched on the different algorithms that are used in dimensionality reduction with mathematical details and through code as well.

It is worth mentioning these algorithms are supposed to be used based on the task at hand. For instance, if the nature of your data is linear then use decomposition methods otherwise use manifold learning techniques.

11. Feature Selection and Extraction





Take an interactive product tour →

Your e-mail

Get Newsletter

Deployment options

Service status

SOLUTIONS

ML Platform Engineer

Data Scientist

ML Engineer

ML Team Lead

Enterprise

Integrations

MLflow

Neptune vs TensorBoard

Other Comparisons

ML Experiment Tracking Tools

ML Metadata Stores

ML Model Registries

MLOps Newsletter

Security portal and SOC 2

The Best MLOps Tools • MLOps at a Reasonable Scale • ML Metadata Store • MLOps: What, Why, and How • Experiment Tracking in Machine Learning

Terms of Service Privacy Policy SLA