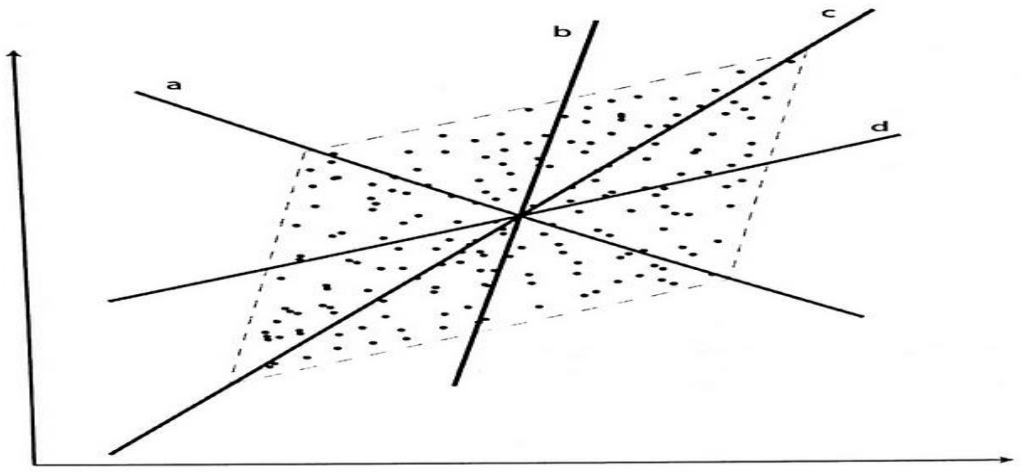
Introduction

Principal Component Analysis is a popular linear dimensionality reduction technique. Data in real world is very high dimensional so we use dimensionality reduction methods to reduce the data to 2 dimensions. The aim of the article is to compress the image using principal component analysis.

High dimensional data is sparse and appropriate statistical methods can not be applied on data. The image has dimension 200 x 200 pixels in size.

Principal Component Analysis



Principal Component Analysis is an unsupervised algorithm in which we don't have the labels. It finds orthogonal projections which are independent. The first principal component is in the direction of maximum variance in the data and so on. The principal components are linearly uncorrelated. eg: in the above scatter plot the first principal component is **c** which is in the direction of maximum variance which means spread of data and would give maximum information about the data. The second principal component is **a** which is orthogonal to **c** and would give less information about the data as compared to first principal component and so on. The data in high dimensional data would be projected onto these 2 principal components which would be in 2 dimension with minimum information loss.

Method

1. Load the image
2. Divide the image into 400 10 by 10 pixel blocks, so that the first block contains pixels in rows 1–10, columns 1–10, the second block contains pixels in rows 1–10, columns 11–20, and so on. Each block contains 100 pixel values and can be thought of as a 100-dimensional input vector, in total you have 400 such 100-dimensional input vectors.
3. Compute the first PCA component of this data set (400 items, 100 features) and project each input vectors onto that component — the result is one scalar value per input vector.
4. Project the scalar values back as a reconstruction of the original features, the result is one 100-dimensional vector per input vector. Draw the resulting picture: for each pixel block, instead of the original pixel values, draw the approximated values.
5. Do the same procedure for 2 components, 5, 10, 20, and 30 PCA components.

Results

From the above results we can see that we started off with 5 principal components and as we proceeded further the quality of the image improved. With 30 principal components we are able to get a reasonably good image. The amount of information which we are able to get from first principal component is 89% which is really nice

Conclusion

We are able to compress the image by taking only 30 principal components and we are getting really good results when compared with the original image. I have uploaded the code on my github account along with the image file. You can try increasing the principal components to see how it increases the quality of the image.

Image Compression using Principal Component Analysis (PCA)

Principal Component Analysis (PCA), is a dimensionality reduction method used to reduce the dimensionality of a dataset by transforming the data to a new basis where the dimensions are non-redundant (low covariance) and have high variance.

Introduction

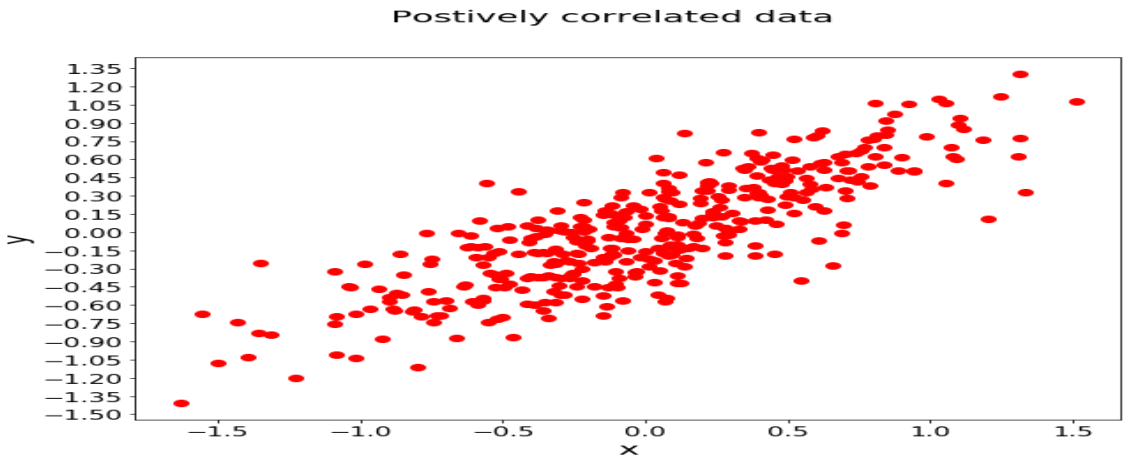
What is Dimensionality Reduction?

Dimensionality: The number of input variables or features for a dataset is referred to as its dimensionality.

Dimensionality reduction is the transformation of data from a high-dimensional space into a low-dimensional space so that the low-dimensional representation retains some meaningful properties of the original data, ideally close to its intrinsic dimension (number of variables needed in a minimal representation of the data).

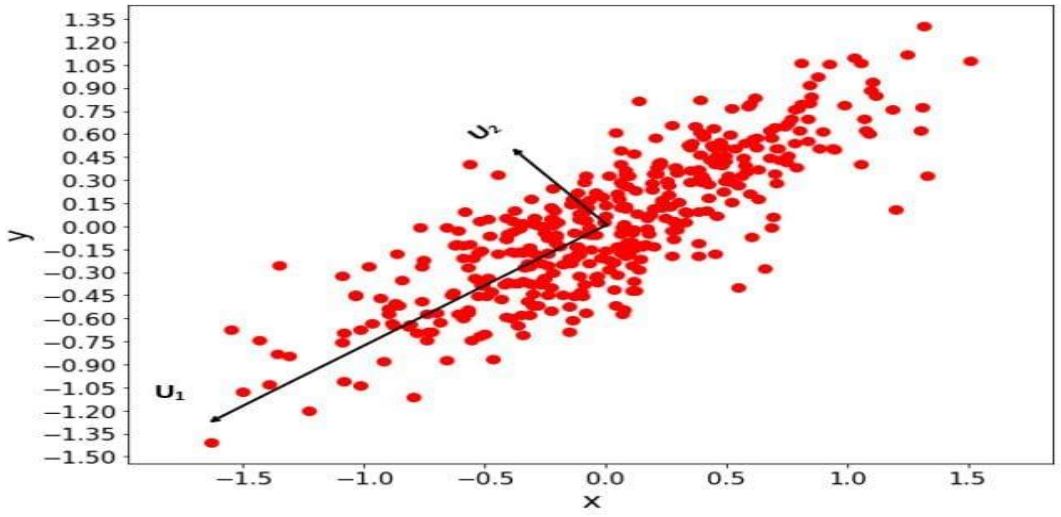
Dimensionality reduction refers to techniques that reduce the number of input variables in a dataset. More input features often make a predictive modeling task more challenging to model, generally referred to as the **curse of dimensionality**.

To explain the concept of dimensionality reduction, I will take an example, consider the following data where each point (vector) is represented using a linear combination of the x and y axes:



Motivation for Dimensionality Reduction

Now, what if we choose a different basis?



Here I have used u_1 and u_2 as a basis instead of x and y. We can observe that all the points have a minimal component in the direction of u_2 (almost noise).

It seems that the same data that was initially in R_2:(x,y) can now be represented in R_1:(u_1) by making an intelligent choice of basis.

Why not care about u_2.

Because the variance in the data in this direction is minimal (all data points have almost the same value in the $u_2$$\hat{}$2 direction), if we were to build a classifier on top of this data, then $u_2$$\hat{}$2 would not contribute to the classifier as the points are not distinguishable along this direction. In this way, we have reduced the dimensionality.

In general, we are interested in representing the data using fewer dimensions such that the data has higher variance along these dimensions.

Correlation

Data correlation is how one set of data may correspond to another set. If two columns are highly correlated (or have high covariance), one is redundant since it is linearly dependent on the other column.

We can normalize the correlation to get the correlation coefficient. The formula for the correlation coefficient is defined as:

Requirements for Dimensionality Reduction

In general, we are interested in representing the data using fewer dimensions such that,

- The data has **high variance** along these dimensions.
- The dimensions are linearly **independent** (uncorrelated).
- If we want to reduce dimensions by transforming the data into a new basis, the resulting basis should be **orthogonal**.

Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is a statistical procedure that uses an orthogonal transformation that converts a set of correlated variables to a set of uncorrelated variables.

To explain the concept of PCA mathematically, I will go over an example:

Orthogonal transformation

Assumptions

Let $p_1, p_2, p_3, \ldots, p_n$ be a set of n orthonormal vectors. Let p be a square matrix of order n such that $p_1, p_2, p_3, \ldots, p_n$ are the columns of matrix p.

Let $x_1, x_2, x_3, \ldots, x_n \in R_n$ be n data points and let X be a square matrix such that $x_1, x_2, x_3, \ldots, x_n$ are the rows of a matrix.

(Assumption: Data is zero-mean and unit variance), we will go over why we have this assumption at a later part.

Transformation

Suppose we want to represent x_i using the new basis P.

$x_i=\alpha_{i1}p_1+\alpha_{i2}p_2+.....+\alpha_{in}p_n$

As we have assumed P as an orthonormal basis, for an orthonormal basis, we can find α_i using, $\alpha_{ij}=x^T_i p_j$.

In general, the transformed data x'_i is given by $x'_i=x^T_i P$, thus the transformed matrix will be given as:

$X'=XP$

Covariance matrix (Σ)

If X is a matrix with zero mean, then $\Sigma=1/m*(X'^T X)$ is the covariance matrix. In other words, Σ_{ij} stores the covariance between columns i and j of X.

Explanation: Let C be the covariance matrix of X. Let μ_i and μ_j denote the means of i^{th} and j^{th} column of X, respectively.

Then by the definition of covariance, we can write:

C_{ij} = $1/m*1/$ $*$ $\Sigma_{mk=1}\Sigma$ $=1$ $(X_{ki}$ $-\mu_i$ $)(X_{kj}$ $-\mu_j$ $)$

C_{ij} = $1/m*1/$ $*$ $\Sigma_{mk=1}\Sigma$ $=1$ $X_{ki}X_{kj}$ $(\because \mu_i$ $=\mu_j$ $=0)$

C_{ij} = $1/m*1/$ $*$ $X^T_i X_j$ = $1/m*1/$ $*(X^T X)_{ij}$ $)$

So far we know that,

$X'=XP$ $'=$ $<$

Covariance matrix of transformed data can be written as:

$=1/m*=1/$ $*$ $X'^T X'$ $'$ $'$

$=1/m*(XP)^T XP=1/$ $*($ $)$

$=1/m*P^T X^T XP=1/$ $*$

$=1/m*P^T(X^T X)P=1/$ $*$ $($ $)$

$=P^T(1/m*(X^T X))P=$ $($ $1/$ $*($ $))$

$=P^T\Sigma P=\Sigma$

What do we want from the covariance matrix of transformed data?

Ideally we want,

- $(1/m*X'^TX')_{ij}=0$ $(1/\Sigma*\Sigma'\Sigma')\Sigma=0$ if i ≠≠ j (Covariance =≠ 0)
- $(1/m*X'^TX')_{ij}\neq 0$ $(1/\Sigma*\Sigma'\Sigma')\Sigma\neq 0$ if i =≠ j (Variance =≠ 0)

In other words, we want $P^T\Sigma P=D\Sigma=\Sigma$ (where $D\Sigma$ is a diagonal matrix).

Few points to ponder

- $X^TX\Sigma\Sigma\Sigma$ is symmetrical.
- It will have distinct non-negative eigenvalues, and thus, linearly independent eigenvectors.
- Eigenvectors of a symmetric matrix are orthogonal, which can be turned into an orthonormal basis.

Principal components

Now we know that $\Sigma\Sigma$ is a symmetric matrix, and the eigenvectors of $\Sigma\Sigma$ can be used as a suitable orthonormal basis. From the Diagonalization of matrix principle, we can say that to make $P^T\Sigma P\Sigma\Sigma\Sigma$ a diagonal matrix, $P\Sigma$ will be the matrix of eigenvectors of the matrix $\Sigma\Sigma$.

Now we can perform orthonormal transformation:

$x_i=\Sigma_{nj=1}(\alpha_{ij}p_j)\Sigma\Sigma=\Sigma\Sigma=1\Sigma(\Sigma\Sigma\Sigma\Sigma\Sigma)$

The $n\Sigma$ orthogonal eigenvectors $p_j\Sigma\Sigma$ are the **Principal Components**.

Dimensionality reduction

As discussed already, we want to retain uncorrelated dimensions that have a maximum variance. Therefore for dimensionality reduction, we will sort the eigenvectors according to eigenvalues in descending order and keep top $k\Sigma$ eigenvectors to represent $x_i\Sigma\Sigma$:

$x'_i=\Sigma_{kj=1}\alpha_{ij}p_j\Sigma\Sigma'=\Sigma\Sigma=1\Sigma\Sigma\Sigma\Sigma\Sigma\Sigma$

Where $x'_i\Sigma\Sigma'$ is a reconstructed vector with $k\Sigma$ dimensions, earlier we were using $n\Sigma$ dimensions to represent it. Hence the dimensionality is being reduced.

Use case: Image compression

(For illustration, I have used the Olivetti dataset, available at: https://www.kaggle.com/imrandude/olivetti)

Consider we are given a large number of images of human faces (for this dataset, we have 400400 images), each image is 64×6464×64(409640964096 **dimensions).**

Now, we would like to represent and store the images using much fewer dimensions(say 100100 dimensions).

```
# Importing necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from numpy import linalg as LA

dirname = '/content/gdrive/My Drive/Kaggle'
fileName = 'olivetti_faces.npy'
faces = np.load(os.path.join(dirname,fileName))
```

Let us see a sample image from the dataset.

```
plt.imshow(faces[1], cmap='gray')
plt.axis('off')
```

Let's see what the average of all images looks like:

```
avgFace = np.average(faces, axis=0)
plt.imshow(avgFace, cmap='gray')
plt.axis('off')
```

First, we will make all our images zero centered, subtracting the average image from each image in the matrix for zero centering.

```
X = faces
X = X.reshape((X.shape[0], X.shape[1]**2)) #flattening the image
X = X - np.average(X, axis=0) #making it zero centered

#printing a sample image to show the effect of zero centering
plt.imshow(X[0].reshape(64,64), cmap='gray')
plt.axis('off')
```

Now, after making the images zero-centered, we will calculate the covariance matrix.

```
cov_mat = np.cov(X, rowvar = False)

#now calculate eigen values and eigen vectors for cov_mat
cov_mat = np.cov(X, rowvar = False)

#sort the eigenvalues in descending order
sorted_index = np.argsort(eigen_values)[::-1]

sorted_eigenvalue = eigen_values[sorted_index]
#similarly sort the eigenvectors
sorted_eigenvectors = eigen_vectors[:,sorted_index]
```

Initially, the image had $4096$4096 dimensions. Let's reduce the dimension from $4096$4096 to $100$100.

```
n_components = 100
eigenvector_subset = sorted_eigenvectors[:,0:n_components]
print(eigenvector_subset.shape)
```

```
(4096, 100)
```

These $100$100 dimensions are the **Principal Components**. Now, as we can see, the shape of eigenvector_subset is $(4096,100)$(4096,100). If represented as a $64×64$64×64 image after performing transpose on this matrix, we can get $100$100 such images.

These $100$100 images will be called **Eigenfaces**, and one can represent any image as a linear combination of these $100$100 images.

Let's print first $16$16 eigenfaces.

```
fig = plt.figure(figsize=[25,25])
for i in range(16):
  if(i%4==0):
  fig.add_subplot(4,4,i+1)
  plt.imshow(eigenvector_subsetT[i].reshape(64,64) , cmap= 'gray')
  plt.axis('off')
plt.show()
```

How is the image compressed?

Initially, the image had dimensions of $4096 4096$. Let's reduce the dimension from $4096 4096$ to $100 100$.

```
x_reduced = np.dot(eigenvector_subset.transpose(),X.transpose()).transpose()
print(x_reduced.shape)
```

```
(400, 100)
```

Earlier, to store 400400 images, we required a $400 \times 4096 400 \times 4096$ matrix. We need to store $400 \times 100 400 \times 100$ matrix for x_reduced and $4096 \times 100 4096 \times 100$ matrix for storing principal components.

As the image is gray-scale, let's suppose it requires 22 bits to store each pixel of an image. Therefore, after compressing the image, we will be able to save:

$=((400 \times 4096) \times 2)] - ((400 \times 100) + (4096 \times 100)) \times 2 = ((400 \times 4096) \times 2)] - ((400 \times 100) + (4096 \times 100)) \times 2$

$= 2377600 = 2377600$ bits

$= 297200 = 297200$ bytes

$\approx 290 \approx 290$ KB

For our example, the images were gray-scale and had a low resolution; that is why we could save only 290290 KB. On the other hand, suppose images have very high resolution with more than one channel, then one can use this method to save lots of space.

Conclusion

As we have seen in this tutorial, using the concept of PCA, we have compressed the images and stored the eigenfaces. And to retain the image, we reconstruct it using the stored eigenfaces.

But we have to note that there is extra calculation overhead for reconstructing the image (matrix multiplication), and also, there will be some reconstruction error. The greater the number of eigenfaces lesser is the reconstruction error.

Support Vector Machines (SVMs) are a type of supervised machine learning algorithm that can be used for classification and regression tasks. In this article, we will focus on using SVMs for image classification.
When a computer processes an image, it perceives it as a two-dimensional array of pixels. The size of the array corresponds to the resolution of the image, for example, if the image is 200 pixels wide and 200 pixels tall, the array will have the dimensions 200 x 200 x 3. The first two dimensions represent the width and height of the image, respectively, while the third dimension represents the RGB color channels. The values in the array can range from 0 to 255, which indicates the intensity of the pixel at each point.

In order to classify an image using an SVM, we first need to extract features from the image. These features can be the color values of the pixels, edge detection, or even the textures present in the image. Once the features are extracted, we can use them as input for the SVM algorithm.

The SVM algorithm works by finding the hyperplane that separates the different classes in the feature space. The key idea behind SVMs is to find the hyperplane that maximizes the margin, which is the distance between the closest points of the different classes. The points that are closest to the hyperplane are called support vectors.

One of the main advantages of using SVMs for image classification is that they can effectively handle high-dimensional data, such as images. Additionally, SVMs are less prone to overfitting than other algorithms such as neural networks.

In machine learning where the model is trained by input data and expected output data.
To create such a model, it is necessary to go through the following phases:

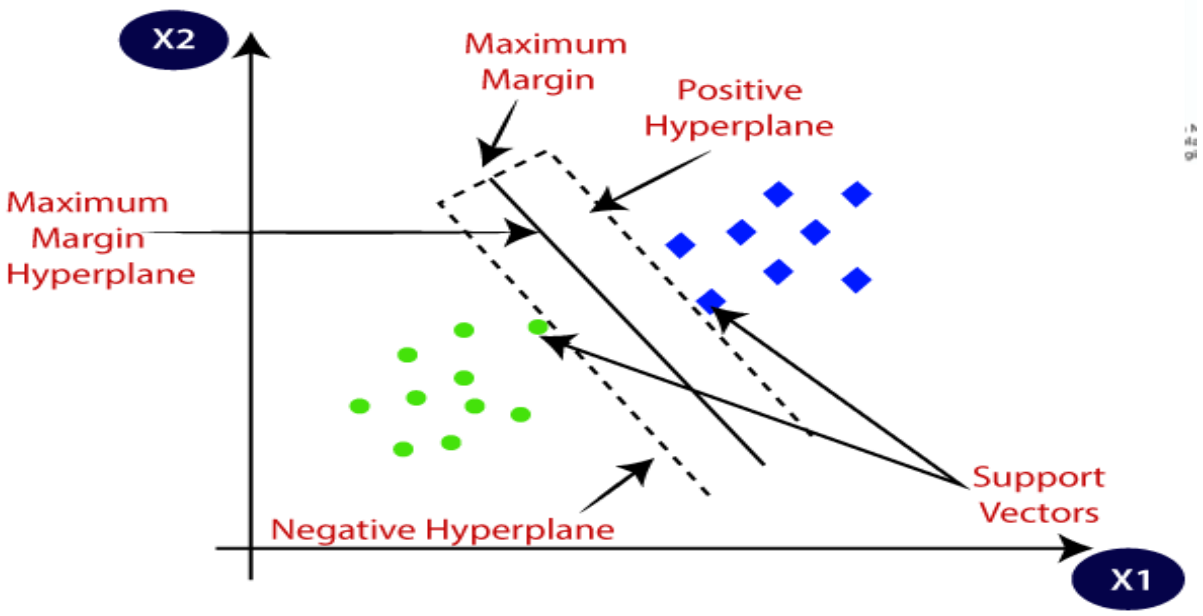
1. Import required libraries
2. Load the image and convert it to a dataframe.
3. separate input features and targets.
4. Split train and test value.
5. Build and train the model
6. Model evaluation.
7. Prediction

Support Vector Machine Algorithm

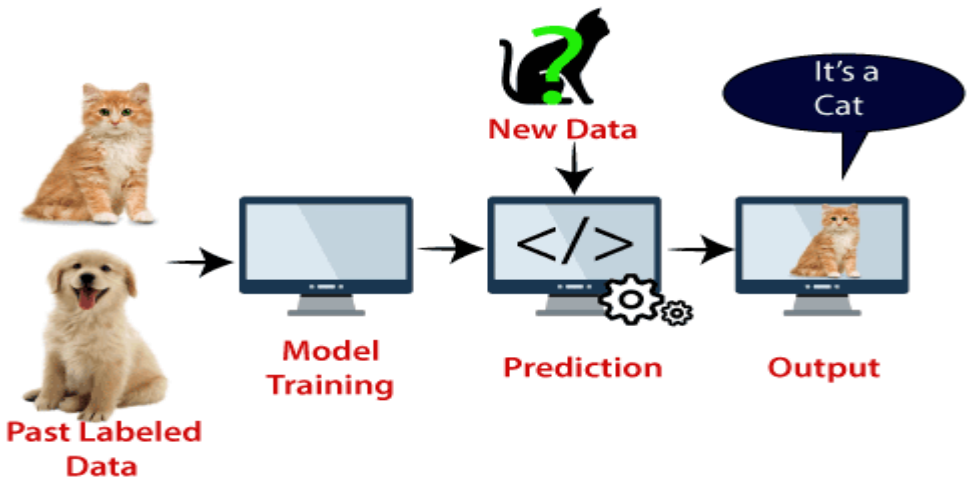
Support Vector Machine or SVM is one of the most popular Supervised Learning algorithms, which is used for Classification as well as Regression problems. However, primarily, it is used for Classification problems in Machine Learning.

The goal of the SVM algorithm is to create the best line or decision boundary that can segregate n-dimensional space into classes so that we can easily put the new data point in the correct category in the future. This best decision boundary is called a hyperplane.

SVM chooses the extreme points/vectors that help in creating the hyperplane. These extreme cases are called as support vectors, and hence algorithm is termed as Support Vector Machine. Consider the below diagram in which there are two different categories that are classified using a decision boundary or hyperplane:



Example: SVM can be understood with the example that we have used in the KNN classifier. Suppose we see a strange cat that also has some features of dogs, so if we want a model that can accurately identify whether it is a cat or dog, so such a model can be created by using the SVM algorithm. We will first train our model with lots of images of cats and dogs so that it can learn about different features of cats and dogs, and then we test it with this strange creature. So as support vector creates a decision boundary between these two data (cat and dog) and choose extreme cases (support vectors), it will see the extreme case of cat and dog. On the basis of the support vectors, it will classify it as a cat. Consider the below diagram:



SVM algorithm can be used for **Face detection, image classification, text categorization,** etc.

Types of SVM

SVM can be of two types:

- **Linear SVM:** Linear SVM is used for linearly separable data, which means if a dataset can be classified into two classes by using a single straight line, then such data is termed as linearly separable data, and classifier is used called as Linear SVM classifier.
- **Non-linear SVM:** Non-Linear SVM is used for non-linearly separated data, which means if a dataset cannot be classified by using a straight line, then such data is termed as non-linear data and classifier used is called as Non-linear SVM classifier.

Hyperplane and Support Vectors in the SVM algorithm:

Hyperplane: There can be multiple lines/decision boundaries to segregate the classes in n-dimensional space, but we need to find out the best decision boundary that helps to classify the data points. This best boundary is known as the hyperplane of SVM.

The dimensions of the hyperplane depend on the features present in the dataset, which means if there are 2 features (as shown in image), then hyperplane will be a straight line. And if there are 3 features, then hyperplane will be a 2-dimension plane.

We always create a hyperplane that has a maximum margin, which means the maximum distance between the data points.

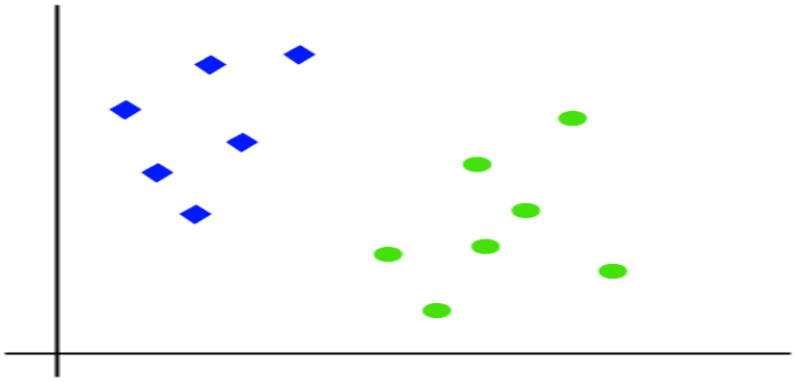
Support Vectors:

The data points or vectors that are the closest to the hyperplane and which affect the position of the hyperplane are termed as Support Vector. Since these vectors support the hyperplane, hence called a Support vector.

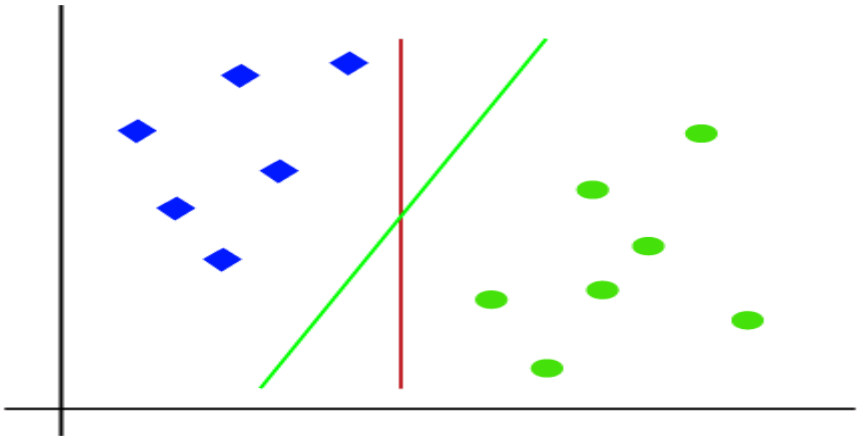
How does SVM works?

Linear SVM:

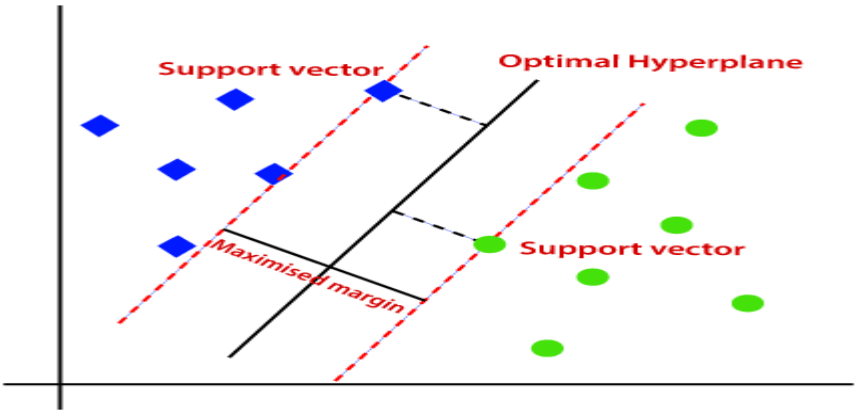
The working of the SVM algorithm can be understood by using an example. Suppose we have a dataset that has two tags (green and blue), and the dataset has two features x1 and x2. We want a classifier that can classify the pair(x1, x2) of coordinates in either green or blue. Consider the below image:



So as it is 2-d space so by just using a straight line, we can easily separate these two classes. But there can be multiple lines that can separate these classes. Consider the below image:

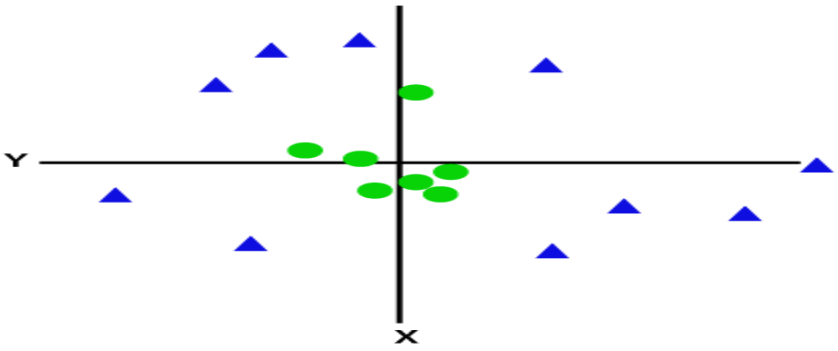


Hence, the SVM algorithm helps to find the best line or decision boundary; this best boundary or region is called as a **hyperplane**. SVM algorithm finds the closest point of the lines from both the classes. These points are called support vectors. The distance between the vectors and the hyperplane is called as **margin**. And the goal of SVM is to maximize this margin. The **hyperplane** with maximum margin is called the **optimal hyperplane**.



Non-Linear SVM:

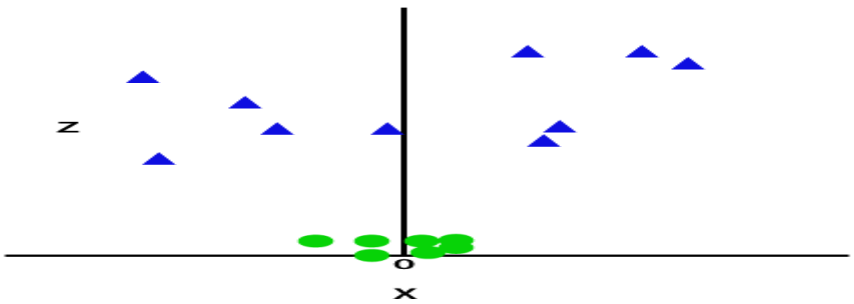
If data is linearly arranged, then we can separate it by using a straight line, but for non-linear data, we cannot draw a single straight line. Consider the below image:



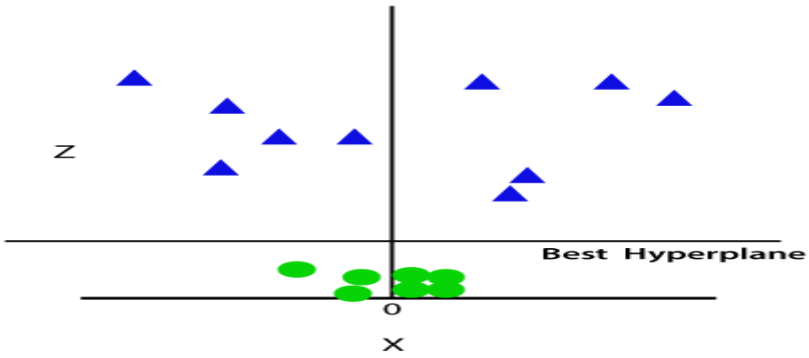
So to separate these data points, we need to add one more dimension. For linear data, we have used two dimensions x and y, so for non-linear data, we will add a third dimension z. It can be calculated as:

$$z=x^2 +y^2$$

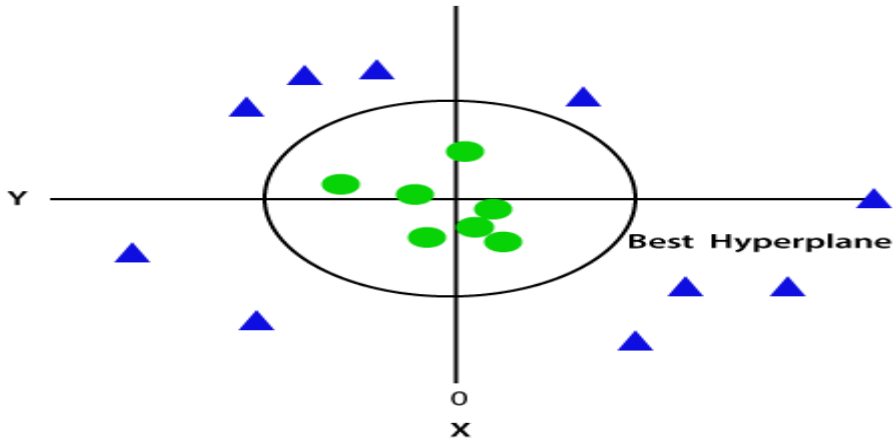
By adding the third dimension, the sample space will become as below image:



So now, SVM will divide the datasets into classes in the following way. Consider the below image:



Since we are in 3-d Space, hence it is looking like a plane parallel to the x-axis. If we convert it in 2d space with z=1, then it will become as:



Principal Component Analysis

Principal Component Analysis is an unsupervised learning algorithm that is used for the dimensionality reduction in machine learning. It is a statistical process that converts the observations of correlated features into a set of linearly uncorrelated features with the help of orthogonal transformation. These new transformed features are called the **Principal Components**. It is one of the popular tools that is used for exploratory data analysis and predictive modeling. It is a technique to draw strong patterns from the given dataset by reducing the variances.

PCA generally tries to find the lower-dimensional surface to project the high-dimensional data.

PCA works by considering the variance of each attribute because the high attribute shows the good split between the classes, and hence it reduces the dimensionality. Some real-world applications of PCA are ***image processing, movie recommendation system, optimizing the power allocation in various communication channels.*** It is a feature extraction technique, so it contains the important variables and drops the least important variable.

The PCA algorithm is based on some mathematical concepts such as:

- Variance and Covariance

- Eigenvalues and Eigen factors

Some common terms used in PCA algorithm:

- **Dimensionality:** It is the number of features or variables present in the given dataset. More easily, it is the number of columns present in the dataset.
- **Correlation:** It signifies that how strongly two variables are related to each other. Such as if one changes, the other variable also gets changed. The correlation value ranges from -1 to +1. Here, -1 occurs if variables are inversely proportional to each other, and +1 indicates that variables are directly proportional to each other.
- **Orthogonal:** It defines that variables are not correlated to each other, and hence the correlation between the pair of variables is zero.
- **Eigenvectors:** If there is a square matrix M, and a non-zero vector v is given. Then v will be eigenvector if Av is the scalar multiple of v.
- **Covariance Matrix:** A matrix containing the covariance between the pair of variables is called the Covariance Matrix.

Principal Components in PCA

As described above, the transformed new features or the output of PCA are the Principal Components. The number of these PCs are either equal to or less than the original features present in the dataset. Some properties of these principal components are given below:

- The principal component must be the linear combination of the original features.
- These components are orthogonal, i.e., the correlation between a pair of variables is zero.
- The importance of each component decreases when going to 1 to n, it means the 1 PC has the most importance, and n PC will have the least importance.

Steps for PCA algorithm

1. **Getting the dataset**
   Firstly, we need to take the input dataset and divide it into two subparts X and Y, where X is the training set, and Y is the validation set.
2. **Representing data into a structure**
   Now we will represent our dataset into a structure. Such as we will represent the two-dimensional matrix of independent variable X. Here each row corresponds to the data items, and the column corresponds to the Features. The number of columns is the dimensions of the dataset.
3. **Standardizing the data**
   In this step, we will standardize our dataset. Such as in a particular column, the features with high variance are more important compared to the features with lower variance. If the importance of features is independent of the variance of the feature, then we will divide each data item in a column with the standard deviation of the column. Here we will name the matrix as Z.
4. **Calculating the Covariance of Z**
   To calculate the covariance of Z, we will take the matrix Z, and will transpose it. After transpose, we will multiply it by Z. The output matrix will be the Covariance matrix of Z.
5. **Calculating the Eigen Values and Eigen Vectors**
   Now we need to calculate the eigenvalues and eigenvectors for the resultant covariance matrix Z. Eigenvectors or the covariance matrix are the directions of the

axes with high information. And the coefficients of these eigenvectors are defined as the eigenvalues.

6. **Sorting the Eigen Vectors**
   In this step, we will take all the eigenvalues and will sort them in decreasing order, which means from largest to smallest. And simultaneously sort the eigenvectors accordingly in matrix P of eigenvalues. The resultant matrix will be named as P*.
7. **Calculating the new features Or Principal Components**
   Here we will calculate the new features. To do this, we will multiply the P* matrix to the Z. In the resultant matrix Z*, each observation is the linear combination of original features. Each column of the Z* matrix is independent of each other.
8. **Remove less or unimportant features from the new dataset.**
   The new feature set has occurred, so we will decide here what to keep and what to remove. It means, we will only keep the relevant or important features in the new dataset, and unimportant features will be removed out.

Applications of Principal Component Analysis

- PCA is mainly used as the dimensionality reduction technique in various AI applications such **as computer vision, image compression, etc.**
- It can also be used for finding hidden patterns if data has high dimensions. Some fields where PCA is used are Finance, data mining, Psychology, etc.

Confusion Matrix in Machine Learning

The confusion matrix is a matrix used to determine the performance of the classification models for a given set of test data. It can only be determined if the true values for test data are known. The matrix itself can be easily understood, but the related terminologies may be confusing. Since it shows the errors in the model performance in the form of a matrix, hence also known as an **error matrix**. Some features of Confusion matrix are given below:

- For the 2 prediction classes of classifiers, the matrix is of 2*2 table, for 3 classes, it is 3*3 table, and so on.
- The matrix is divided into two dimensions, that are **predicted values** and **actual values** along with the total number of predictions.
- Predicted values are those values, which are predicted by the model, and actual values are the true values for the given observations.
- It looks like the below table:

n = total predictions	Actual: No	Actual: Yes
Predicted: No	True Negative	False Positive
Predicted: Yes	False Negative	True Positive

The above table has the following cases:

- **True Negative:** Model has given prediction No, and the real or actual value was also No.
- **True Positive:** The model has predicted yes, and the actual value was also true.
- **False Negative:** The model has predicted no, but the actual value was Yes, it is also called as **Type-II error**.
- **False Positive:** The model has predicted Yes, but the actual value was No. It is also called a **Type-I error.**

Need for Confusion Matrix in Machine learning

- It evaluates the performance of the classification models, when they make predictions on test data, and tells how good our classification model is.
- It not only tells the error made by the classifiers but also the type of errors such as it is either type-I or type-II error.
- With the help of the confusion matrix, we can calculate the different parameters for the model, such as accuracy, precision, etc.

Example: We can understand the confusion matrix using an example.

Suppose we are trying to create a model that can predict the result for the disease that is either a person has that disease or not. So, the confusion matrix for this is given as:

n = 100	Actual: No	Actual: Yes	
Predicted: No	TN: 65	FP: 3	68
Predicted: Yes	FN: 8	TP: 24	32
	73	27	

From the above example, we can conclude that:

- The table is given for the two-class classifier, which has two predictions "Yes" and "NO." Here, Yes defines that patient has the disease, and No defines that patient does not has that disease.
- The classifier has made a total of **100 predictions**. Out of 100 predictions, **89 are true predictions**, and **11 are incorrect predictions**.
- The model has given prediction "yes" for 32 times, and "No" for 68 times. Whereas the actual "Yes" was 27, and actual "No" was 73 times.

Calculations using Confusion Matrix:

We can perform various calculations for the model, such as the model's accuracy, using this matrix. These calculations are given below:

- **Classification Accuracy:** It is one of the important parameters to determine the accuracy of the classification problems. It defines how often the model predicts the correct output. It can be calculated as the ratio of the number of correct predictions made by the classifier to all number of predictions made by the classifiers. The formula is given below:

$$\text{Accuracy}=\frac{TP+TN}{TP+FP+FN+TN}$$

- **Misclassification rate:** It is also termed as Error rate, and it defines how often the model gives the wrong predictions. The value of error rate can be calculated as the number of incorrect predictions to all number of the predictions made by the classifier. The formula is given below:

$$\text{Error rate}=\frac{FP+FN}{TP+FP+FN+TN}$$

- **Precision:** It can be defined as the number of correct outputs provided by the model or out of all positive classes that have predicted correctly by the model, how many of them were actually true. It can be calculated using the below formula:

$$\text{Precision} = \frac{TP}{TP+FP}$$

- **Recall:** It is defined as the out of total positive classes, how our model predicted correctly. The recall must be as high as possible.

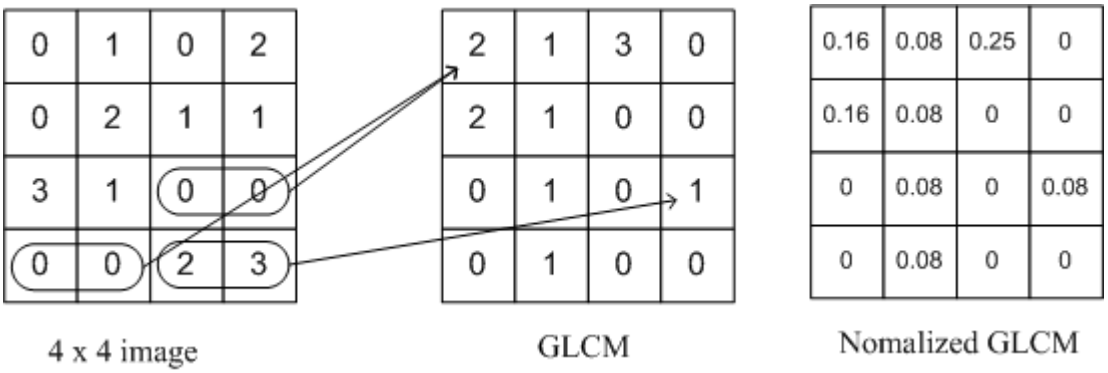
$$\text{Recall} = \frac{TP}{TP+FN}$$

- **F-measure:** If two models have low precision and high recall or vice versa, it is difficult to compare these models. So, for this purpose, we can use F-score. This score helps us to evaluate the recall and precision at the same time. The F-score is maximum if the recall is equal to the precision. It can be calculated using the below formula:

$$\text{F-measure} = \frac{2*Recall*Precision}{Recall+Precision}$$

Other important terms used in Confusion Matrix:

- **Null Error rate:** It defines how often our model would be incorrect if it always predicted the majority class. As per the accuracy paradox, it is said that "*the best classifier has a higher error rate than the null error rate.*"
- **ROC Curve:** The ROC is a graph displaying a classifier's performance for all possible thresholds. The graph is plotted between the true positive rate (on the Y-axis) and the false Positive rate (on the x-axis).
- *Gray Level Co-occurrence Matrix (GLCM)* is used for texture analysis. We consider two pixels at a time, called the reference and the neighbour pixel. We define a particular spatial relationship between the reference and neighbour pixel before calculating the GLCM. For eg, we may define the neighbour to be 1 pixel to the right of the current pixel, or it can be 3 pixels above, or 2 pixels diagonally (one of NE, NW, SE, SW) from the reference.
- Once a spatial relationship is defined, we create a GLCM of size (Range of Intensities x Range of Intensities) all initialised to `0`. For eg, a `8` bit single channel Image will have a `256x256` GLCM. We then traverse through the image and for every pair of intensities we find for the defined spatial relationship, we increment that cell of the matrix.



- Each entry of the GLCM[i,j] holds the count of the number of times that pair of intensities appears in the image with the defined spatial relationship.
- The matrix may be made symmetrical by adding it to its transpose and normalised to that each cell expresses the probability of that pair of intensities occurring in the image.
- Once the GLCM is calculated, we can find texture properties from the matrix to represent the textures in the image.

GLCM Properties

- The properties can be calculated over the entire matrix or by considering a window which is moved along the matrix.
- **Mean**
- **Variance**
- **Correlation**
- **Contrast**
- **IDM (Inverse Difference Moment)**
- **ASM (Angular Second Moment)**
- **Entropy**
- **Max Probability**
- **Energy**
- **Dissimilarity**

- **Morphological operations** can be used to refine or modify the shapes of objects in images
  Many morphological operations can be applied to **binary images** to improve an image segmentation
- **Grayscale morphological operations** can also be used as processing steps before binarization, or to help identify regional maxima and minima

Erosion & dilation

Our first two morphological operations, **erosion** and **dilation**, are actually identical to minimum and maximum filtering respectively, described in the previous chapter. The names erosion and dilation are used more often when speaking of binary images, but the operations are the same irrespective of the kind of image.

Structuring elements

The neighborhood used to calculate the result for each pixel is defined by a **structuring element**. This is similar to a filter kernel, except that it only has values 0 and 1 (for ignoring or including the neighborhood pixel, respectively).

Here, we assume the background value in our binary image is 0 (black) and foreground is 1 (white).
Erosion will make objects in the binary image smaller, because a pixel will be set to the background value if *any* other pixels in the neighborhood are background. This can split single objects into multiple pieces.

Conversely, **dilation** makes objects bigger, since the presence of a single foreground pixel anywhere in the neighborhood will result in a foreground output. This can also cause objects to merge.

Show code cell content

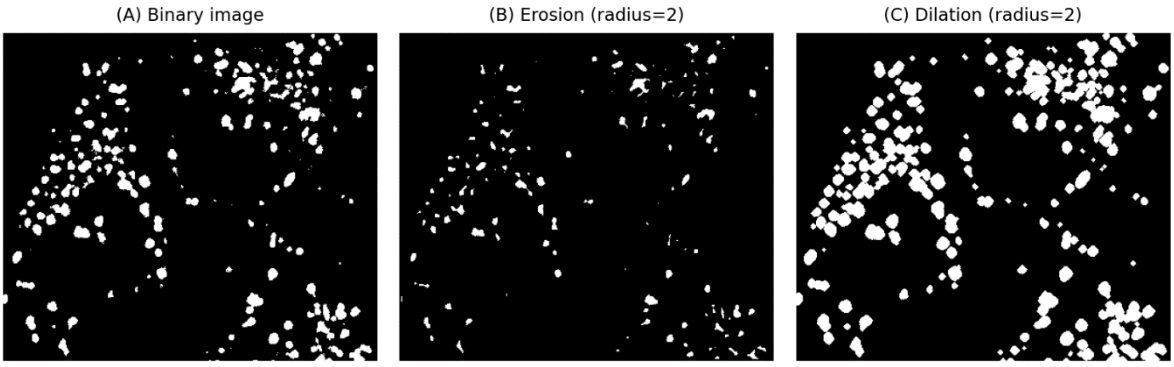


Fig. 103 The effects of erosion and dilation on a binary image of small structures.

Opening & closing

The fact that erosion and dilation alone affect sizes can be a problem: we may like their abilities to merge, separate or remove objects, but prefer that they had less impact upon areas and volumes. Combining both operations helps achieve this.

Opening consists of an erosion followed by a dilation. It therefore first shrinks objects, and then expands whatever remains to *approximately* its original size.

Such a process is not as pointless as it may first sound. If erosion causes very small objects to completely disappear, clearly the dilation cannot make them reappear: they are gone for good. Barely-connected objects separated by erosion are also not reconnected by the dilation step.

Closing is the opposite of opening, i.e. a dilation followed by an erosion, and similarly changes the shapes of objects. The dilation can cause almost-connected objects to merge, and these often then remain merged after the erosion step. If you wish to count objects, but they are wrongly subdivided in the segmentation, closing may help make the counts more accurate.

Show code cell content

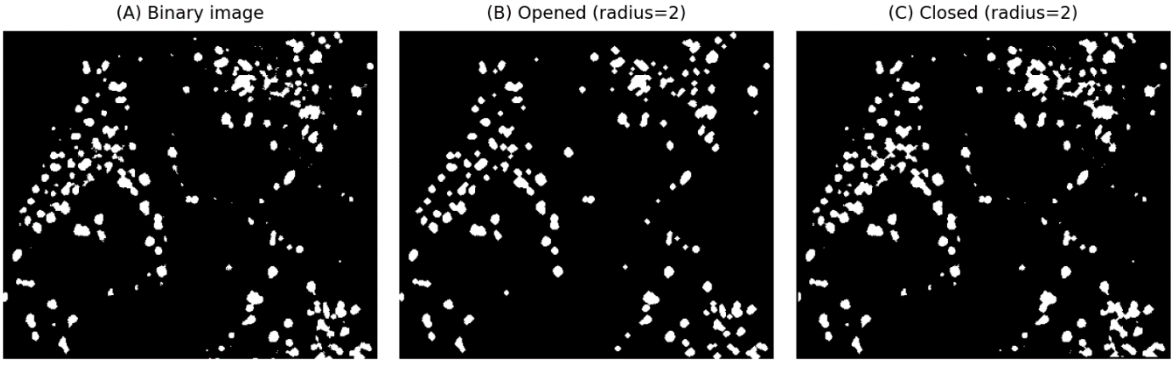


Fig. 104 The effects of opening and closing on a binary image of small structures. Unlike when using erosion or dilation alone, the sizes of objects are largely preserved although the contours are modified. Opening has the effect of completely removing the smallest or thinnest objects.

Boundaries & outlines

We can make use of the operations above to identify outlines in a binary image. To do this, we first need a clear definition of what we mean by ‘outline’.

The **inner boundary** may be defined as *the foreground pixels that are adjacent to background pixels*. We can determine the inner boundary by

- Duplicating the binary image
- Eroding with a 3×3 structuring element
- Subtracting the eroded image from the original

The **outer boundary** may be defined as *the background pixels that are adjacent to foreground pixels*. We can determine the outer boundary by

- Duplicating the binary image
- Dilating with a 3×3 structuring element
- Subtracting the original image from the dilated image

Thicker boundaries

There's no reason to limit outlines to being 1 pixel thick. Choosing a larger structuring element makes it possible create thicker outlines. We might also subtract an eroded image from a dilated image to identify a thicker boundary that contains both inner and outer pixels.

One application of creating thick boundaries in microscopy images of cells is to generate a binary image of the nuclei, and then a second binary image representing a ring around the nucleus. This makes it possible to make measurements that are likely to be within the cytoplasm, just outside the nucleus, without the task of identifying the full area of the cell – which is often difficult if the cell or membrane are not clearly visible.

Finding local minima & maxima

Erosion and dilation can be used to find pixels that are **local maxima** or **local minima** very easily, with the caveat that the results are inexact and often unusable. Nevertheless, the trick works 'well enough' sufficiently often to be worth knowing.

Here, we focus on maxima; the process for detecting local minima is identical, except that either the image should be inverted or erosion used instead of dilation.

A local maximum can be defined as a pixel with a value greater than all its neighbors, or a connected group of pixels with the same higher value than the surrounding pixels. An easy way to detect these pixels is to dilate the image with 3×3 maximum filter, and check for pixel values that are unchanged (i.e. where the pixel was already a maximum within its neighborhood).

This is inexact because it does not *only* identify maxima; it also detections some 'plateaus' where pixels have identical values to their neighbors. In practice, this is not always a problem because noise can make plateaus virtually non-existent for many real-world images (at least ones that haven't been clipped).

A bigger problem is that the approach often identifies far too many maxima to be useful (Fig. 106).

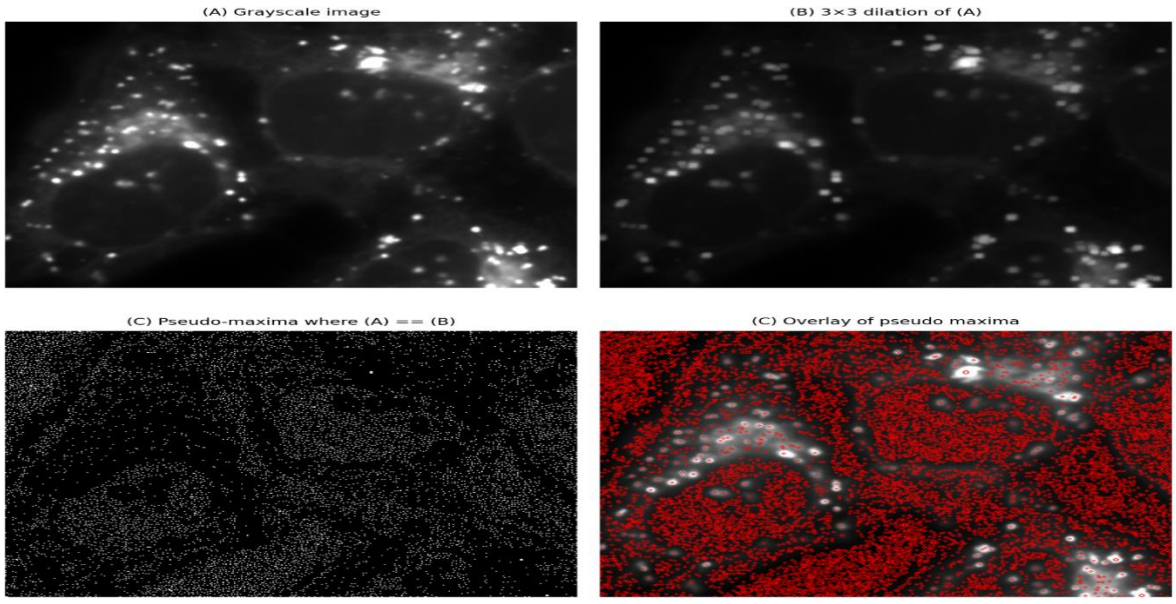


Fig. 106 Identifying local maxima with the help of a 3×3 dilation tends to find too many maxima to be useful.

We can reduce these by either increasing the size of the maximum filter (therefore requiring pixels to be maximal across a larger region), or by pre-smoothing the image (usually with a Gaussian filter). However, tuning the parameters becomes difficult.

We will see an alternative approach that is often more intuitive in H-Maxima & H-Minima.

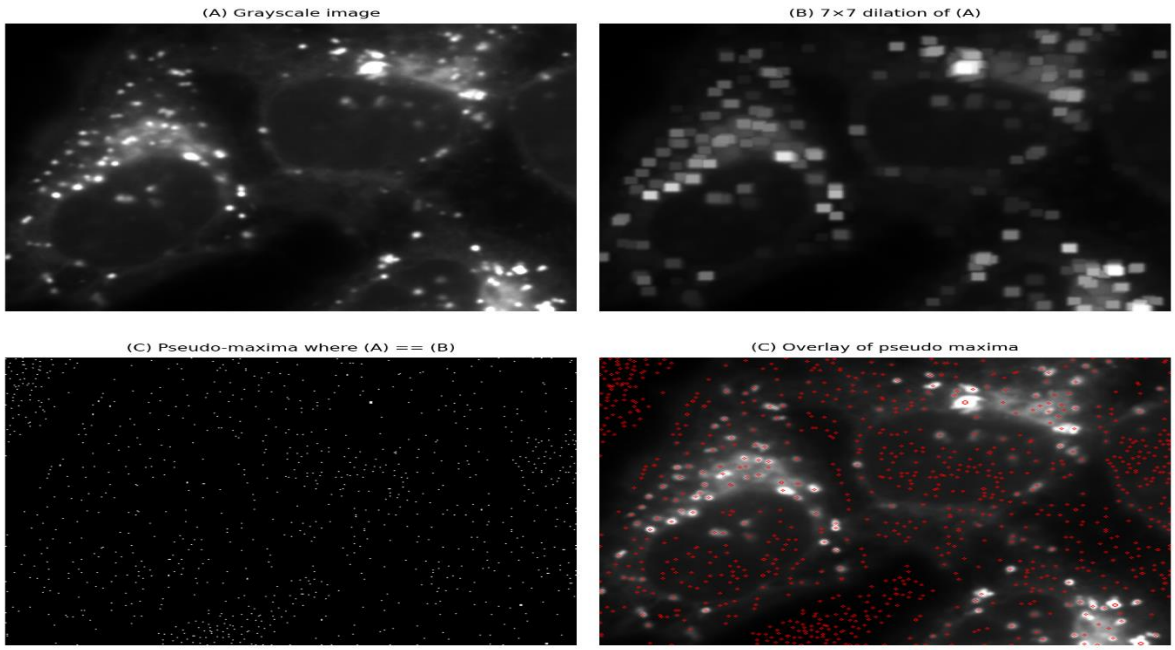


Fig. 107 Identifying local maxima with the help of a larger dilation (here, 7×7 pixels) can sometimes give better results than using a smaller dilation Fig. 106.

More morphological operations

Area opening

Area opening is similar to *opening*, except it avoids the need for any kind of maximum or minimum filtering.

It works by identifying **connected components** in the binary image, which are contiguous regions of foreground pixels. For each connected component, the number of pixels is counted to give an area in px². If the area of a component falls below a specified area threshold, the pixels for that component are set to the background, i.e. the component is removed.

Area opening is often preferable to *opening*, because it has *no impact* on the shape of any structures larger than the area threshold. It simply applies a minimum area threshold, removing everything smaller.

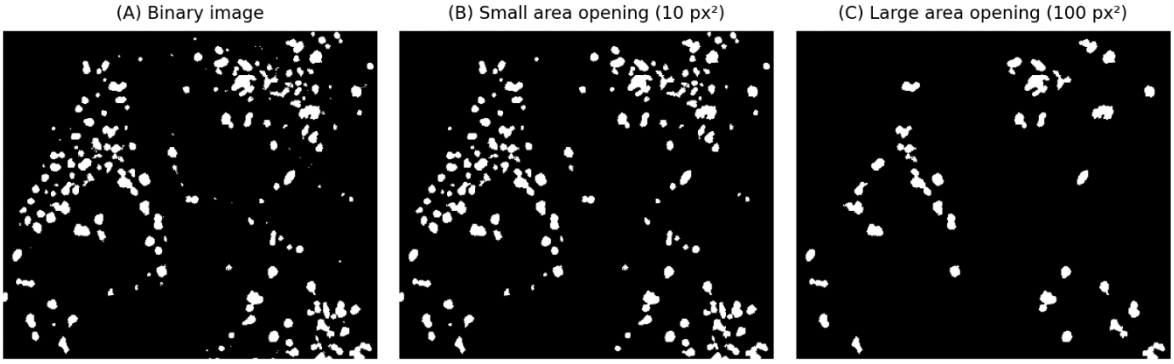


Fig. 108 Using area opening to remove small objects.

Filling holes

Filling holes involves identifying connected components of *background pixels* that are entirely surrounded by foreground pixels. These components are then 'flipped' to become foreground pixels instead.

Should we then want to identify the holes themselves, we can subtract the original image from the filled image.

Show code cell content

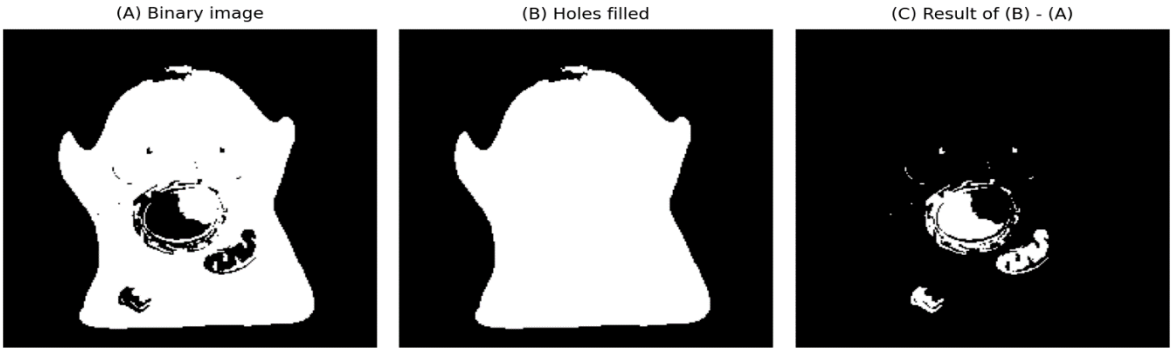


Fig. 109 Filling holes in a binary image. Image subtraction makes it possible to extract the holes themselves.

Show code cell content

Fig. 110 Small holes filled.

Question

We don’t always want to fill *all* the holes within a binary image, but rather only the smaller ones. Can you think of a way to fill *only holes smaller than 1000 px²*, using area opening?

You’ll need at least one operation described in previous chapter.

Answer

Thinning & skeletonization

Thinning and **skeletonization** are related operations that aim to ‘thin down’ objects in a binary image to just their centerlines. They are particularly useful with filamental or tube-like structures, such as axons or blood vessels.

Show code cell content

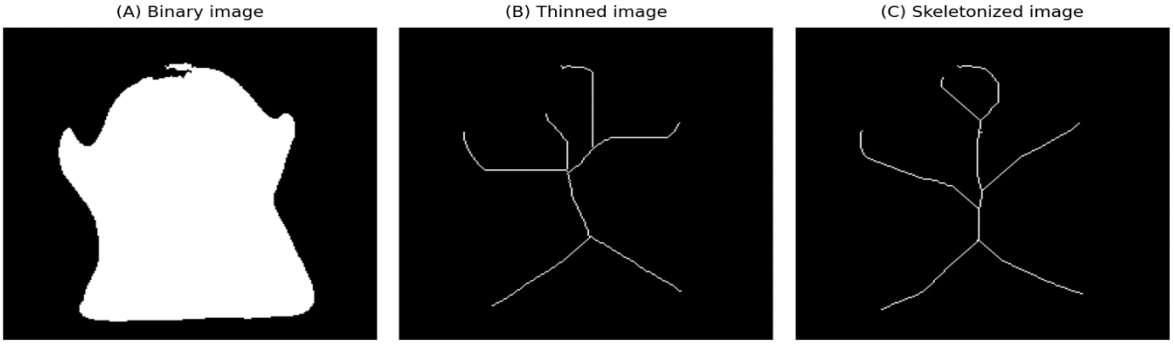


Fig. 111 The effects of thinning and skeletonization on a binary image.

What’s the difference between thinning & skeletonization?

The truth is: I’m not entirely sure. There is quite a bit of overlap in the literature, and I’ve seen the same algorithm referred to by both names. Furthermore, there are different thinning algorithms that give different results; the situation is similar for skeletonization algorithms.

Software occasionally offers both thinning and skeletonization, but often just offers one or the other. It’s worth trying any thinning/skeletonization methods available to see which performs best for any particular application.

Morphological reconstruction

Morphological reconstruction is a somewhat advanced technique that underpins several powerful image processing operations. It’s useful with both grayscale and binary images.

Morphological reconstruction requires two images of the same size: a **marker** image and a **mask** image. The pixel in the *mask* image should all have values greater than or equal to the corresponding pixels in the *marker* image.

The reconstruction algorithm progressively *dilates* the marker image (e.g. applies a 3×3 maximum filter), while constraining the marker to remain ‘within’ the mask; that is, the pixel values in the marker are never allowed to exceed the values in the mask. This dilation is repeated iteratively until the marker cannot change any further without exceeding the mask. The output is the new marker image, after all the dilations have been performed.

Some examples will help demonstrate how this works and why it's useful. The crucial difference in the methods below is how the marker and mask images are created.

Hysteresis thresholding

One use of morphological reconstruction is to implement a **double threshold**, also known as **hysteresis thresholding**.

For *low threshold* and *high threshold*, I assume we're detecting light structures on a dark background.
This involves defining both a **low threshold** and a **high threshold.** The low threshold operates like any global threshold to identify regions. However, a region is discarded from the binary image if it does not also contain at least one pixel that exceeds the high threshold.

This is achieved using morphological reconstruction by defining the *marker* as all pixels exceeding the high threshold, and the *mask* as all pixels exceeding the low threshold. The markers will expand to fill the mask regions that contain them. But any mask regions that don't contain marker pixels are simply ignored.

Show code cell content

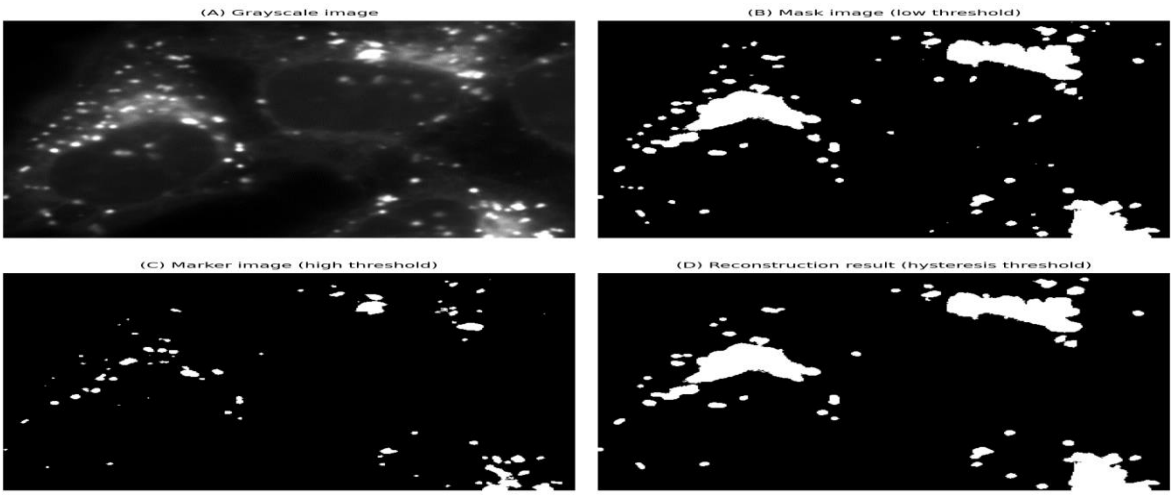


Fig. 112 Applying a hysteresis threshold to an image. The size and area of the objects detected by this method are determined by the low threshold, but at least one of the pixel values within the object must exceed the high threshold. This slightly mitigates the problem of a single global threshold having a huge impact on analysis results, by the same threshold simultaneously influencing both what is detected and its size.

H-Maxima & H-Minima

We saw previously that we could (kind of) identify local maxima in a very simple way using an image dilation, but the results are often too inaccurate to be useful.

H-Maxima and **H-Minima** can help us overcome this. These operations both require only one intuitive parameter: they enable us to identify maxima or minima using a local intensity threshold *H*.

This is achieved using morphological reconstruction. For H-maxima, the process is:

- Set the original grayscale image as the *mask*
- Subtract *H* from the mask to create the *markers*
- Apply morphological reconstruction using the markers and mask
- Subtract the reconstruction result from the *mask*
- Threshold the subtracted image with a global threshold of *H*

The main steps are illustrated in Fig. 113. We can apply the same process to an inverted image to find *H-minima*.

Show code cell content

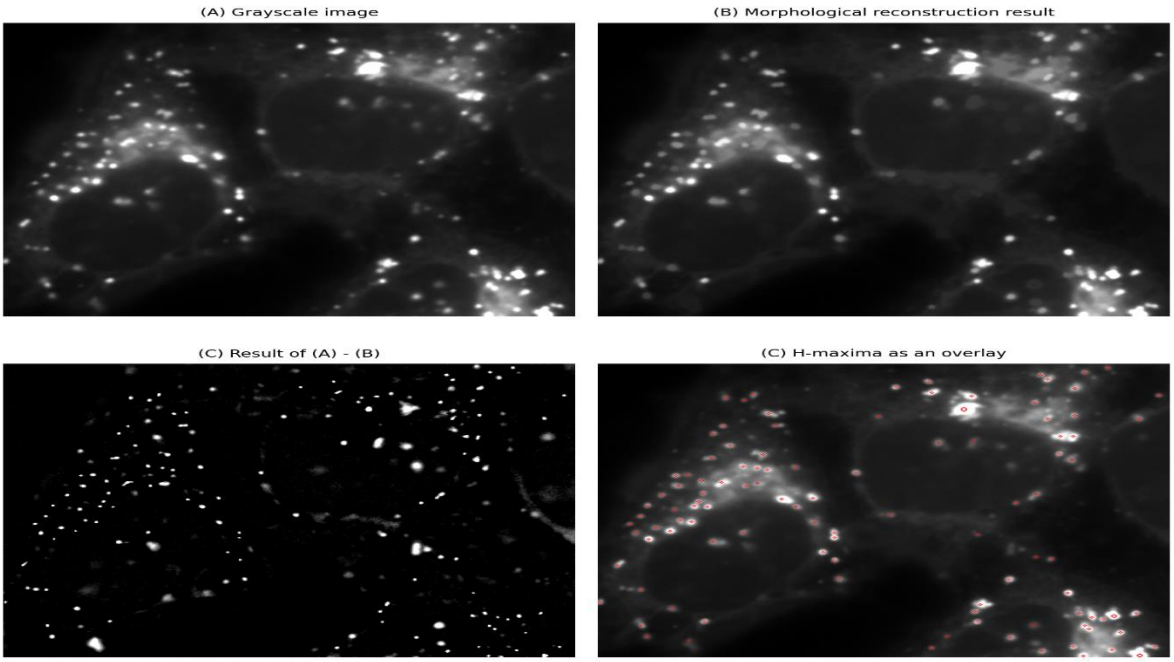


Fig. 113 Calculating H-maxima using morphological reconstruction. Here, *H* is set (arbitrarily) to be the image standard deviation.

Opening & closing by reconstruction

H-maxima and H-minima use morphological reconstruction to effectively generate a background image that can be subtracted from the original. We do this by subtracting a constant *H*, which acts as a local intensity threshold.

We can also use morphological reconstruction to generate a background image based upon spatial information, rather than an intensity threshold *H*, by using **opening by reconstruction**. This effectively introduces a size component into our local threshold. **Closing by reconstruction** is an analogous operation that can be defined using morphological closing.

The starting point for opening by reconstruction is a *morphological opening* as defined above, i.e. an erosion followed by a dilation. This defines the marker image. The original image is used as the mask.

Show code cell content

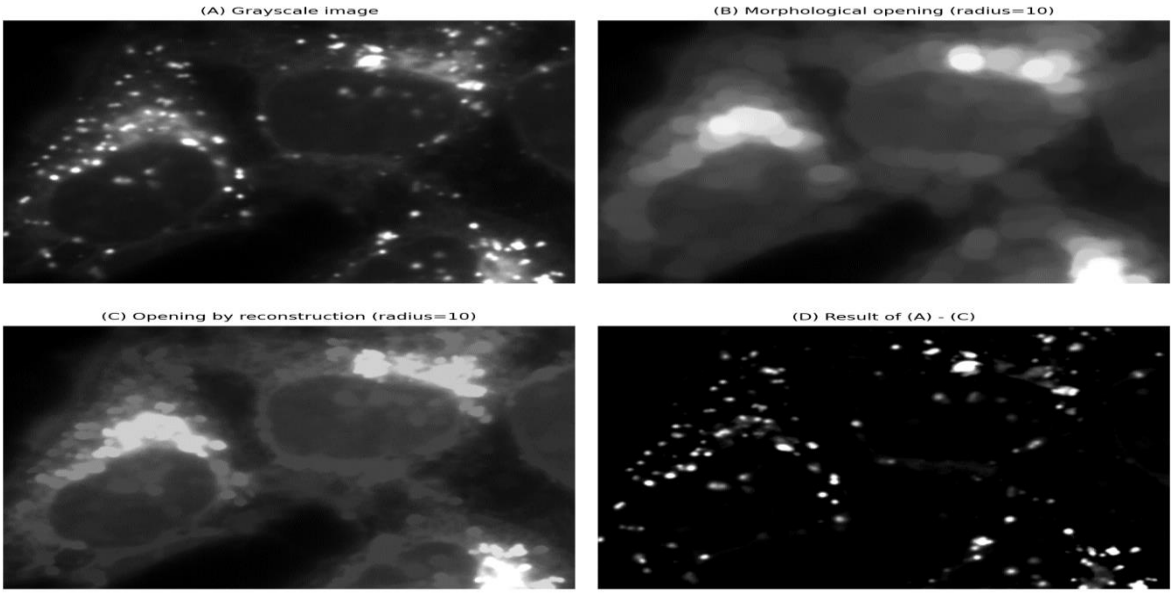


Fig. 114 Using opening by reconstruction to obtain a background estimate. The estimate can be subtracted from an image before applying a global threshold.

As before, opening alone removes structures that are smaller than the structuring element, while slightly affecting the shapes of everything else. Opening by reconstruction essentially adds some further (constrained) dilations so that the structures that were *not* removed are more similar to how they were originally. This can make opening by reconstruction more attractive for generating background images that will be used for subtraction.

Opening by reconstruction can also be applied to binary images as an alternative to *opening* and *area opening*. Like area opening, opening by reconstruction is able to remove some objects while retaining the shapes of larger objects exactly.

Show code cell content

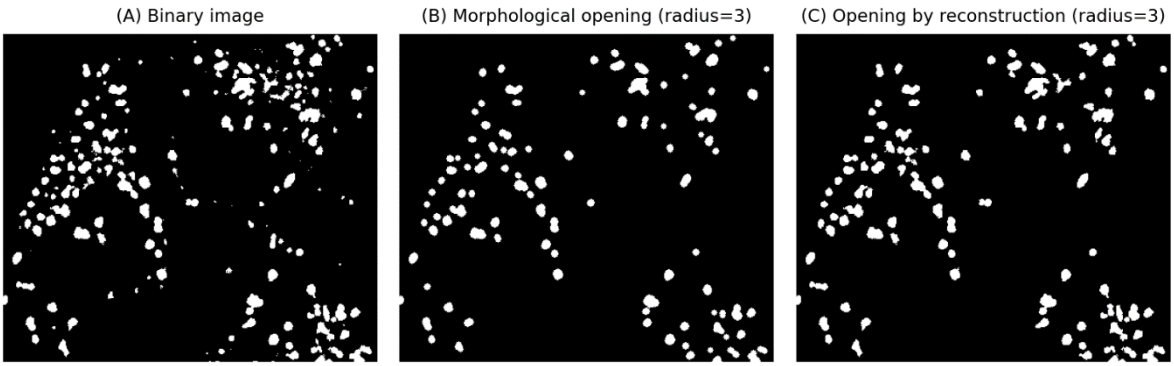


Fig. 115 Using opening by reconstruction to remove small (and thin) objects from a binary image, while retaining the original shape of everything that remains.#