Contents

CHAPTER 1

The Gram-Schmidt Process

The Gram-Schmidt Process is a method use to transform a set of linearly independent vectors to a set of orthogonal (perpendicular) vectors. The original vectors and the transformed vectors span the same subspace.

The process was named after two mathematicians: Jørgen Pedersen Gram, a Danish actuary mathematician, and Erhard Schmidt, a German mathematician. The men developed the orthogonalization process independently. Gram introduced the process in 1883, whereas Schmidt did his work in 1907. It was not named the Gram-Schmidt process until sometime later, after both mathematicians became well known in the mathematical community.

In the last chapter, you learned how to calculate a projection of one vector on another. Given two vectors, u and v, the projection separates u into the part that is orthogonal to v and the part that has a dependency with v. The Gram-Schmidt Process builds on the notion of projections to iteritively strip away dependencies between vectors until the vectors that remain are orthogonal. If there happens to be a vector that is a linear combination of the others, that vector will be reduced to the zero vector. The vectors that remain define a new basis for space spanned by the original set of vectors.

Gram-Schmidt has many practical applications in science and engineering, such as:

1. In signal processing, it can represent an audio signal with fewer components making it easier to isolate and remove noise.
2. In statistics and data analysis, it can reduce the complexity of a dataset so that it is easier to see which aspects or features contribute to the analysis.

1.1 The Process

The Gram-Schmidt Process orthonormalizes a set of vectors in an inner product space, most commonly the Euclidean space $\mathbb{R}^n$. The process takes a finite, linearly independent set $S = \{v_1, v_2, \ldots, v_k\}$ for $k \leq n$, and generates an orthogonal set $S' = \{u_1, u_2, \ldots, u_k\}$ that spans the same k-dimensional subspace of $\mathbb{R}^n$ as S.

Let's look at how the process works. Given a set of vectors $S = \{v_1, v_2, \ldots, v_k\}$, the Gram-Schmidt Process is as follows:

1. Let $u_1 = v_1$.
2. For $j = 2, 3, \ldots, k$:
   (a) Let $w_j = v_j - \sum_{i=1}^{j-1} \frac{\langle v_j, u_i \rangle}{\langle u_i, u_i \rangle} u_i$
   (b) Let $u_j = w_j$

Here, $\langle ., . \rangle$ denotes the inner product.

The set of vectors $S' = \{u_1, u_2, \ldots, u_k\}$ obtained from this process is orthogonal, but not necessarily orthonormal. To create an orthonormal set, you simply need to normalize each vector u_i to unit length. That is, $u_i' = \frac{u_i}{\|u_i\|}$, where $\|.\|$ denotes the norm (or length) of a vector.

Among other things, making vectors orthonormal simplifies calculations makes it easier to define rotations and transformations, and provides a framework for calculations in fields such as quantum mechanics.

1.2 Example Calculation

Given a set of linearly independent vectors, we will use the Gram-Schmidt Process to find an orthogonal basis.

Let

$$W = Span(x_1, x_2, x_3)$$

where

$$x_1 = (1, 2, -2)$$
$$x_2 = (1, 0, -4)$$
$$x_3 = (5, 2, 0)$$

The three orthogonal vectors will define the same subspace as the original vectors.

The first vector of the orthogonal subspace is easy to define. We set it to be the same as x_1.

$$v_1 = x_1 = (1, 2, -2)$$

The second orthogonal vector is a projection of x_2 onto v_1. You learned projections in the last chapter, so this should be fairly straightforward.

$$v_2 = x_2 - \frac{x_2 v_1}{v_1 v_1} v_1$$

Substitute the values:

$$v_2 = (1, 0, -4) - \frac{(1, 0, -4)(1, 2, -2)}{(1, 2, -2)(1, 2, -2)} (1, 2, -2)$$

Calculate the coefficient for v_1:

$$v_2 = (1, 0, -4) - \frac{9}{9}(1, 2, -2)$$

Perform the subtraction:

$$v_2 = (0, -2, -2)$$

The third vector for the orthogonal subspace is a projection onto v_1 and v_2.

$$v_3 = x_3 - \frac{x_3 v_1}{v_1 v_1} v_1 - \frac{x_3 v_2}{v_2 v_2} v_2$$

Substitute the values:

$$v_3 = (5, 2, 0) - \frac{(5, 2, 0)(1, 2, -2)}{(1, 2, -2)(1, 2, -2)}(1, 2, -2) - \frac{(5, 2, 0)(1, 0, -4)}{(1, 0, -4)(1, 0, -4)}(1, 0, -4)$$

$$v_3 = (5, 2, 0) - (9/9)(1, 2, -2) - (-4/8)(1, 0, -4)$$

$$v_3 = (5, 2, 0) - (1, 2, -2) + (1/2)(1, 0, -4)$$

$$v_3 = (5, 2, 0) - (1, 2, -2) + (0, -1, -1)$$

$$v_3 = (4, -1, 1)$$

This set of vectors is orthogonal, so we need to normalize them so that the vectors are orthonormal. Recall that an orthonormal vector has a length of 1 and is computed using this formula:

$$normalizedVector = vector/np.sqrt(np.sum(vector**2))$$

Thus the normalized set of vectors is:

$$v_1 = (0.33, 0.67, -0.67)$$

$$v_2 = (0.0, -0.71, -0.71)$$

$$v_3 = (0.94, -0.24, 0.24)$$

Exercise 1 Gram-Schmidt Process

Use the Gram-Schmidt Process to to find an orthogonal basis for the span defined by x_1, x_2 where:

$$x_1 = (1, 1, 1)$$

$$x_2 = (0, 1, 1)$$

Working Space

Answer on Page 25

1.3 The Gram-Schmidt Process in Python

Create a file called `vectors_gram-schmidt.py` and enter this code:

```
# import numpy to perform operations on vector
import numpy as np

# Find an orthogonal basis for the span of these three vectors
x1 = np.array([1, 2, -2])
x2 = np.array([1, 0, -4])
x3 = np.array([5, 2, 0])

# v1 = x1
v1 = x1
print("v1 = ",v1)

# v2 = x2 - (the projection of x2 on v1)
v2 = x2 - (np.dot(x2,v1)/np.dot(v1,v1))*v1
print("v2 = ", v2)

# v3 = x3 - (the projection of x3 on v1) - (the projection of x3 on v3)
v3 = x3 - (np.dot(x3,v1)/np.dot(v1,v1))*v1 - (np.dot(x3,v2)/np.dot(v2,v2))*v2
print("v3 =", v3)

# Next, normalize each vector to get a set of vectors that is both orthogonal and orthonormal:
v1_norm = v1 / np.sqrt(np.sum(v1**2))
v2_norm = v2 / np.sqrt(np.sum(v2**2))
v3_norm = v3 / np.sqrt(np.sum(v3**2))
```

```
print("v1_norm = ", v1_norm)
print("v2_norm = ", v2_norm)
print("v3_norm = ", v3_norm)
```

1.4 Where to Learn More

Watch this video from Khan Academy about the Gram-Schmidt Process: https://www.youtube.com/watch?v=rHonltF77zI

Chapter 2

Singular Value Decomposition

In the previous chapter, you learned how to calculate eigenvalues and eigenvectors. However, not every matrix has them. For those matrices, singular values and singular vectors are analogous features.

Singular Value Decomposition (SVD) is a matrix factorization technique that breaks down a matrix into three matrices that represent the structure and properties of the original matrix. The decomposed matrices make calculations easier and provide insight into the original matrix. Essentially, SVD can transform a high dimension, highly variable set of data into a set of uncorrelated data points that reveal subgroupings that you might not have noticed in the original data. SVD tells us that a linear transformation can be thought of as a rotation, scaling, and another rotation.

2.1 Definition

For any $m \times n$ matrix A, SVD decomposes the matrix into three matrices.

$$A = U\Sigma V^T \tag{2.1}$$

- U is an orthogonal matrix whose size is $m \times m$. Its columns are the eigenvectors of AA^T. These are the left singular vectors of A. Because U is orthogonal, $U^TU = I$.
- V is an orthogonal matrix whose size is $n \times n$ matrix. Its columns are the eigenvectors of A^TA. These are the right singular vectors of A. Because V is orthogonal, $V^TV = I$.
- Σ is a diagonal matrix that is the same size as A. Its diagonal contains the singular values of A, arranged in descending order. These values are the square roots of the eigenvalues of both A^TA and AA^T.

2.2 Applications of SVD

SVD has numerous applications:

- It is used in machine learning and data science to perform dimensionality reduction, particularly through a technique known as Principal Component Analysis (PCA).

- In numerical linear algebra, SVD is used to solve linear equations and compute matrix inverses in a more numerically stable way.
- It is used in image compression, where low-rank approximations of an image matrix provide a compressed version of the original image.

2.3 Calculating SVD Manually

You might be inclined to skip this example because the computations are lengthy. Why would anyone do this when they can use a computing language, like Python, to calculate the SVD with essentially one command? We show this so you can understand what goes on "under the hood" when you compute SVD programmatically.

After you read through this example, you will see how to use Python to compute SVD. Next, you will see an example of using SVD for image compression. Finally, you will be given an exercise to compute the SVD. For this, you will need to write your own Python script.

Let's find the SVD for matrix A. Recall that we want to find U. Σ, and V^T such that:

$$A = U\Sigma V^T \tag{2.2}$$

$$A = \begin{bmatrix} 3 & 1 & 1 \\ -1 & 3 & 1 \end{bmatrix}$$

$$U = AA^T$$

Calculating

$$AA^T$$

will give us a square matrix:

$$A^T = \begin{bmatrix} 3 & -1 \\ 1 & 3 \\ 1 & 1 \end{bmatrix}$$

$$AA^T = \begin{bmatrix} 3 & 1 & 1 \\ -1 & 3 & 1 \end{bmatrix} \begin{bmatrix} 3 & -1 \\ 1 & 3 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 11 & 1 \\ 1 & 11 \end{bmatrix}$$

Next, we will find the eigenvalues and eigenvectors of A^T. This is a chance to apply what you learned in the previous chapter. We know that:

$$Av = \lambda v \tag{2.3}$$

So:

$$\begin{bmatrix} 11 & 1 \\ 1 & 11 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \lambda \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Rewrite as a set of equations:

$$11x_1 + x_2 = \lambda x_1$$

$$x_1 + 11x_2 = \lambda x_2$$

Then rearrange:

$$(11-\lambda)x_1 + x_2 = 0$$

$$x_1 + (11-\lambda)x_2 = 0$$

Solve for λ:

$$\begin{bmatrix} (11-\lambda), 1 \\ 1, (11-\lambda) \end{bmatrix} = 0$$

And as equations:

$$(11-\lambda)(11-\lambda)-1\cdot 1 = 0$$

$$(\lambda-10)(\lambda-12) = 0$$

These are the eigenvalues.

$$\lambda = 10$$

$$\lambda = 12$$

When substituted into the original equations, you get the eigenvectors. For

$$\lambda = 10$$

:

$$(11-10)x_1 + x_2 = 0$$

$$x_1 = -x_2$$

We will set

$$x_1$$

to 1 and get this eigenvector:

$$[1, -1]$$

For

$$\lambda = 10$$

:

$$(11-12)x_1 + x_2 = 0$$

$$x_1 = x_2$$

We will set

$$x_1$$

to 1 and get this eigenvector:

$$[1, 1]$$

The matrix is:

$$\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

Next, you need to apply the Gram-Schmidt process to the column vectors. After that, you will have U, the $m \times m$ matrix whose columns are eigenvectors of AA^T. These are the left singular vectors of A. After you apply Gram-Schmidt, you should end up with:

$$U = \begin{bmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{bmatrix}$$

The process for calculating V is the same as the calculation for U, except:

$$V = A^T A$$

$$A^T A = \begin{bmatrix} 3 & -1 \\ 1 & 3 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 3 & 1 & 1 \\ -1 & 3 & 1 \end{bmatrix} = \begin{bmatrix} 10 & 0 & 2 \\ 0 & 10 & 4 \\ 2 & 4 & 2 \end{bmatrix}$$

After applying the process we applied to solve for U, you get:

$$V = \begin{bmatrix} 1/\sqrt{6} & 2/\sqrt{5} & 1/\sqrt{30} \\ 2/\sqrt{6} & -1/\sqrt{5} & 2/\sqrt{30} \\ 1/\sqrt{6} & 0 & -5/\sqrt{30} \end{bmatrix}$$

However, you want V_T:

$$V_T = \begin{bmatrix} 1/\sqrt{6} & 2/\sqrt{6} & 1/\sqrt{6} \\ 2/\sqrt{5} & -1/\sqrt{5} & 0 \\ 1/\sqrt{30} & 2/\sqrt{30} & -5/\sqrt{30} \end{bmatrix}$$

You have only to calculate Σ, a diagonal matrix that is the same size as A. The diagonal contains the singular values of A, arranged in descending order. These are the square roots of the eigenvalues of both $A^T A$ and AA^T.

Because the non-zero eigenvalues of U are the same as V, let's use the eigenvalues we calculate for U, 10 and 12. Note that Σ will not be of the correct dimension to reconstruct the orignal matrix unless we add a column. By adding a zero column you'll be able to multiply between U and V:

$$\Sigma = \begin{bmatrix} \sqrt{12} & 0 & 0 \\ 0 & \sqrt{12} & 0 \end{bmatrix}$$

You can check your work by multiplying the decomposed matrices. This should return the orginal matrix.

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

$$= U = \begin{bmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{bmatrix} \begin{bmatrix} \sqrt{12} & 0 & 0 \\ 0 & \sqrt{12} & 0 \end{bmatrix} \begin{bmatrix} 1/\sqrt{6} & 2/\sqrt{6} & 1/\sqrt{6} \\ 2/\sqrt{5} & -1/\sqrt{5} & 0 \\ 1/\sqrt{30} & 2/\sqrt{30} & -5/\sqrt{30} \end{bmatrix}$$

$$= \begin{bmatrix} \sqrt{12}/\sqrt{2} & \sqrt{10}/\sqrt{2} & 0 \\ \sqrt{12}/\sqrt{2} & -\sqrt{10}/\sqrt{2} & 0 \end{bmatrix} \begin{bmatrix} 1/\sqrt{6} & 2/\sqrt{6} & 1/\sqrt{6} \\ 2/\sqrt{5} & -1/\sqrt{5} & 0 \\ 1/\sqrt{30} & 2/\sqrt{30} & -5/\sqrt{30} \end{bmatrix}$$

$$= \begin{bmatrix} 3 & 1 & 1 \\ -1 & 3 & 1 \end{bmatrix}$$

2.4 Singular Value Decomposition with Python

Create a file called `vectors_decomposition.py` and enter this code:

```
# Singular-value decomposition
import numpy as np
from numpy import array
from scipy.linalg import svd
from numpy import diag
from numpy import dot
from numpy import zeros

# Define a matrix
A = array([[1, 2], [3, 4], [5, 6]])

print("Matrix (3x2) to be decomposed: ")
print(A)

# CalculateSVD
U, S, VT = svd(A)
print("Matrix (3x3) that represents the left singular values of A:")
print(U)
print("Singular values:")
print(S)
print("Matrix (2x2) that represents the right singular values of A:")
print(VT)

# Check if the decomposition by rebuilding the original matrix
# The singular values must be in an m x n matrix
# Create a zero matrix with the same dimension as A
Sigma = zeros((A.shape[0], A.shape[1]))
# Populate Sigma with n x n diagonal matrix
Sigma[:A.shape[1], :A.shape[1]] = diag(S)
# Reconstruct the original matrix
A_Rebuilt = U.dot(Sigma.dot(VT))
print("Original matrix:")
print(A_Rebuilt)
```

2.5 Sign Ambiguity

You might notice that at times, the absolute values in the U and V^T matrices are correct, but that the signs vary from what you see as the answer. For example, when you compare a manually calculated SVD with one done in Python the signs might not agree. Both decompositions of A are valid. Both decompositions will satisfy:

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

Note that the S diagonal values will always be positive.

The sign ambiguity has implications. For example, when using SVD to compress data, if some of the signs are flipped, the data can have artifacts. At this point in your education, you don't need to concern yourself with it, except when you are comparing SVD results for the same matrix.

Exercise 2 Single Value Decomposition

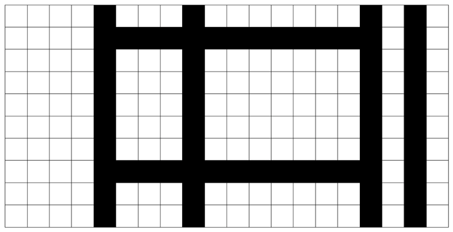
Modify your Python code to calculate SVD for the matrix in the worked out example. Did you arrive at the same answer? Keep in mind that Python will compute square roots and present fractions as decimal. Take a look at the signs for the values in the U and V^T matrices. Are they the same, or is this an example of sign ambiguity?

Working Space

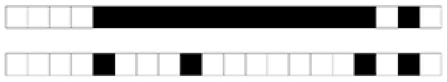
Answer on Page 25

2.6 SVD Applied to Image Compression

This image consists of a grid of 20 by 10 pixels, each of which is either black or white.



It is a simple image that has only two types of columns–ideal for data compression. A row is either the first pattern or the second.



We can represent the data as a 20 by 10 matrix whose 200 entries are either 0 for black or 1 for white.

$$\begin{bmatrix} 00001000100000001010 \\ 00001111111111111010 \\ 00001000100000001010 \\ 00001000100000001010 \\ 00001000100000001010 \\ 00001000100000001010 \\ 00001000100000001010 \\ 00001111111111111010 \\ 00001000100000001010 \\ 00001000100000001010 \end{bmatrix}$$

When you perform an SVD on this matrix, there are only two non-zero singular values, 6.79 and 3.72. (You are welcome perform the calculation in Python.) Thus, you can represent the matrix as:

$$A = U_1 S_1 V_1 + U_2 S_2 V_2$$

This means there are two u vectors, each with 20 entries, two v vectors each with 10 entries, and two singular values. Add those up: 2*20 + 2*10 + 2 = 62. This implies that the image can be represented by 62 values instead of 200. If you look back at the image, you can see that there are many dependent columns and very few independent ones.

This is a simple image and a small pixel matrix, but it should give you a sense of how SVD can decompose an image in a way that identifies how much of the image is redundant, and therefor can be compressed.

2.7 Where to Learn More

We Recommend a Singular Value Decomposition. This American Mathematical Society publication focuses the geometry of SVD. What we like about the article is that it shows both graphically and numerically how SVD can be used for data compression on images and for noise reduction. The data compression example in your workbook is based on this article. https://www.ams.org/publicoutreach/feature-column/fcarc-svd

Sign Ambiguity in Singular Value Decomposition (SVD). This is a good article for those who want a deeper understanding of sign ambiguity. https://www.educative.io/blog/sign-ambiguity-in-singular-value-decomposition

Singular Value Decomposition Tutorial. This PDF starts by defining points, space, and vectors and works through all the concepts you need to tackle SVD. It is one of the few resources that has a completely worked out example of manually calculating SVD. The example in this chapter is from that tutorial. If you read the entire paper, you will find that it is a good review of the concepts you have studied in previous chapters. https://rb.gy/j6s0w

Chapter 3

Tackling Difficult Problems: Positive Semidefinite Matrices

With all the computing power available today, you would think no problem would be too difficult to tackle. However, this is not the case. There is a category of problems called NP (nondeterministic polynomial time) whose solution is easy to verify, but whose computation is difficult to perform. This is because there is no straightforward algorithm and any brute-force method would take too much time. For these problems, all we can do is to develop an efficient algorithm that can find a solution in a reasonable amount of time. While the solution might not be the most optimal, the goal is to find the best solution possible in a short time.

As you learn more about optimization techniques, you will come across many efficiency algorithms that have been used throughout the years. In the 1990s, the field of optimization changed with the discovery that algorithms based on semidefinite positive matrices can achieve a higher efficiency than seen in the past. Today, there is an entire field of programming called Semidefinite Programming that is based on the use of semidefinite positive matrices.

First we will take a look at what some of the NP problems are. Next, we will describe the intuition behind semidefinite positive matrices. We will take a look at one NP problem, then introduce the Python module to use for solving these problems.

3.1 NP Problems

In the world of mathematics, easy problems are referred to as P, or polynomial time, problems. In simple terms, this means the problem can be solved quickly, and it is easy to verify that the solution is correct. Addition, subtraction, division, multiplication, square roots, and matrix-vector multiplication are just a few examples. NP problems, as stated in the introduction, can't be computed in a reasonable amount of time, but solutions are usually easy to verify. The game of Sudoku is one such example. It is easy to verify a correct solution, but writing a generalized algorithm to solve any Sudoku game is an NP problem.

NP problems show up in many other situations, such as:

- Designing robust communication networks

- Scheduling tasks without conflicts
- Managing supply chains
- Detecting patterns in biological networks
- Figuring out subgroups in social networks
- Predicting the structure of proteins

For each of these , think of large scale problems for which there are many variables. A cloud computing company that provides AI services to thousands of clients must be able to schedule tasks efficiently and in a timely manner, as well as manage the power needed for the computers and cooling the data center. Supply chain management is crucial to figuring out how to pick up, transport, and distribute goods to help provide disaster relief. Understanding protein structure is important for designing drugs that can tackle specific conditions. All we can do for each of these situations is to find an optimal solution, but we cannot find the definitive solution.

3.2 A Past Approach: Minimizing Errors in Neural Networks

When neural networks were first being developed to recognize things (faces, letters, music, and so on), the goal was to calculate weights between network nodes that would minimize the recognition error. The error space could be visualized as a surface of valleys and hills. The lowest point would have the least error. The idea behind the iterative weight calculations for training the network was to descend down the gradient until reaching the low point. Without getting into the mathematics, you can see by looking at this figure that there are two valleys. Some neural network training resulted in ending at a low point, but not the lowest point. Wouldn't it be great if you could formulate an optimization problem so that you would be guaranteed to land at the lowest point? That is where semidefinite programming can help.

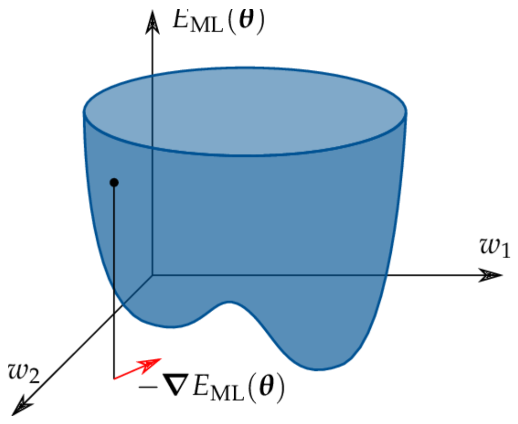
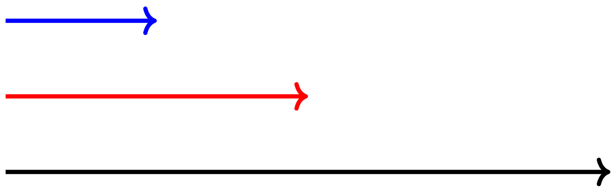

$E_{\mathrm{ML}}(\boldsymbol{\theta})$
w_1
w_2
$-\nabla E_{\mathrm{ML}}(\boldsymbol{\theta})$

Figure 3.1: FIXME.

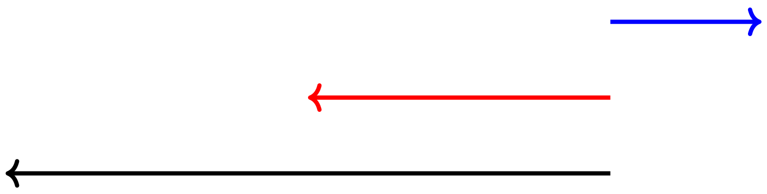
3.3 Positive Definite and Semidefinite Matrices

Unlike matrices that are defined by their content (such as identity matrix, zero matrix, and diagonal matrix), positive definite and semidefinite matrices are characterized by the result they produce. They have an analog in the scalar world, so let's first look at that. Let's take the scalars a and b and treat them as vectors. ab is then the dot product. If $a > 0$, then ab will take on the sign of b. If $a < 0$, then ab will have the opposite sign of b. If you look at this in a graph, you can see that in the first case, ab stays on the same side of the origin, but in the second case, ab flips

For $a = [2], b = [4]$



For $a = [3], b = [-4]$. the result of multiplication flips a to the other side of the graph.



The notion of "staying on the same side" is positive definite. The notion of flipping is negative definite. Positive definite means that $x > 0$, so the result is a positive number. Positive semidefinite means that $x >= 0$, so the result is a non-negative number.

If you can formulate a problem as a positive semidefinite matrix, then you automatically constrain the result to the "same side." This constraint results in higher algorithmic efficiency.

Let's look at the formal definition:

A matrix is positive definite if, and only if:

$$x^T Ax > 0, x, x \neq 0$$

A matrix is positive semidefinite, if, and only if:

$$x^T Ax \geq 0$$

Further, a positive semidefinite matrix had an important property. Its eigenvalues are ≥ 0. The eigenvalues of a positive definite matrix are > 0.

Take the triplet (a, b, c) and the symmetric matrix:

$$\begin{bmatrix} 1 & a & b \\ a & 1 & c \\ b & c & 1 \end{bmatrix} \geq 0$$

For what values of a, b, c is this matrix positive semidefinite? If you iterate through all possible combinations of a,b,c, then compute the eigenvalues for each matrix, you will find that some (like $0, 0, 0$) result in a postive semidefinite matrix and some (like $2, 2, 2$) are not positive semidefinite. If you plot the set of triplets that result in a semidefinite matrix, you will see an elliptope. This shape guarantees an optimal solution.

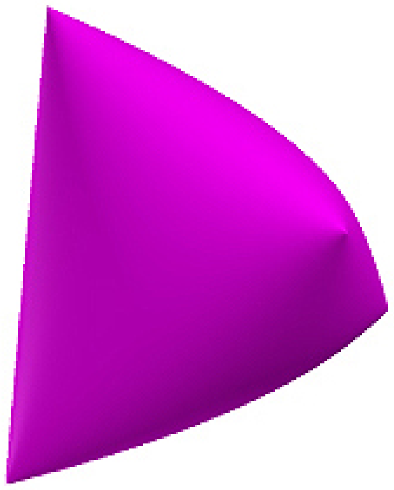


Figure 3.2: Elliptope visualized.

3.4 Identifying and Constructing a Positive Semidefinite Matrix

In the last section, you saw that being symmetric does not guarantee a positive semidefinite matrix. You also saw that a matrix of positive values does not guarantee a positive semidefinite matrix. The only way to check for positive semidefinite is to calculate the eigenvalues, which you learned in a previous workbook.

A surefire way to construct a positive semidefinite matrix is:

$$AA^T$$

Exercise 3 Figuring out if a matrix is positive semidefinite

Is this matrix positive definite? Show your work.

$$\begin{bmatrix} 2 & 2 \\ 2 & 0 \end{bmatrix}$$

Working Space

Answer on Page 26

Exercise 4 Creating a positive semidefinite matrix

Using any 3 by 3 matrix, create a positive semidefinite matrix.Then show it is positive semidefinite by calculating its eigenvalues. You can either compute this by hand or using Python. In either case, show your work.

Working Space

Answer on Page 26

3.5 The Max Cut Problem

A famous NP problem is Max Cut. Given a graph of interconnected nodes, cut the graph to create two sets of nodes, such that the cut goes through as many edges as possible. (You can't cut an edge more than once.) Max Cut is important for binary classification in machine learning, circuit design, statistical physics, and more. There is no algorithm that will provide an exact solution. (If you could find one, you would be eligible to win a huge prize from the Clay Mathematics Institute!) Instead, you will see how to approximate a solution to this problem using a positive semidefinite matrix and a technique developed by mathematicians Michel Goemans and David Williamson.

You won't see all the complete details here, as this section is meant to be a quick introduction to how you can apply positive semidefinite matrices.

Take this simple graph of five nodes and six edges. Each node in the graph will take on one of two values (1 or -1) to show which set they fall into after a cut is made. For any two connected nodes, $x_i \cdot x_j = 1$ if $x_i = x_j$ and -1 otherwise.

If you randomly assign 1 and -1 to the nodes, the chance of making the max cut is 0.5. By using semidefinite programming, you can achieve an algorithmic efficiency of 0.87.

The Goemans-Williamson technique can be used for any optimization problem where the variables take on the values of 1 and -1.

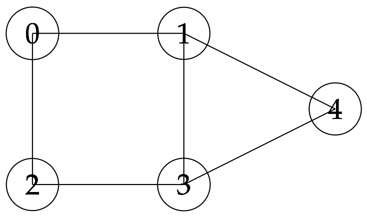


Figure 3.3: MaxCut algorithm visualized.

As a list of edges the graph is:

$$edges = [(0,1),(0,2),(1,3),(1,4),(2,3),(3,4)]$$

The optimization problem can be formulated as:

$$Max \sum_{edges(i,j)} \frac{1 - x_i x_j}{2}$$

for

$$x_i \in \{-1, 1\}$$

However, instead of allowing x_i to be scalar, Goemans-Williamson defines x_i as unit vectors.

$$x_i \in R^n, x_i = 1$$

and that makes the optimization equation:

$$Max \sum_{edges(i,j)} \frac{1 - x_i^T x_j}{2}$$

It is this "relaxation" that gets us to a semidefinite matrix, because we can now rewrite the problem as a positive semidefinite matrix:

$$X = \left[x_i^T x_j\right]_{i,j}$$

Python has a module for solving optimization problems. Using this, you will get an optimum matrix, but to get the unit vectors, you will need to take the square root of the matrix.

$$X = [x_1 ... x_{1n}]\,[x_1 ... x_{1n}]^T$$

Next we need to go from unit vectors to scalars, using a process called rounding.

$$x^i \in R^n \rightarrow x^i \in \{-1, 1\}$$

Goemans-Williamson leveraged the fact that the end point of a unit vector is on a sphere. They generated a random plane to bisect the sphere. A vector on one side of the plane is assigned the value of 1 and a vector on the other side a value of -1.

You can then assign the scalar values to the nodes and make the cut accordingly. This particular cut will be 5, as shown.

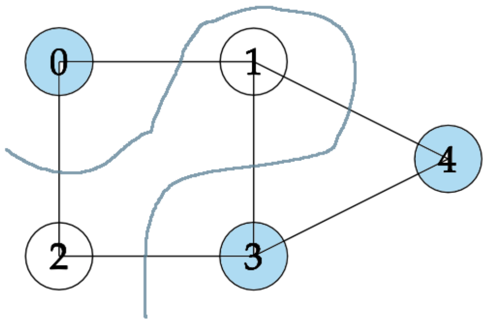


Figure 3.4: The result of the MaxCut.

3.6 The Max Cut Problem Solved in Python

```
# MaxCut Problem
import numpy as np
import scipy
from scipy.linalg import sqrtm
# cvxpy is a python module for solving optimization problems
import cvxpy as cp

# define the edges of the graph
edges = [(0,1),
         (0,2),
         (1,3),
        (1,4),
        (2,3),
        (3,4)]

#  Declare the matrix X to be positive semidefinite
X = cp.Variable((5,5),symmetric=True)
constraints = [X >> 0]

# Set diagonals to 1 to get unit vectors
constraints += [
    X[i,i] == 1 for i in range(5)
]

# Set the objective function
objective = sum(05.*(1 - X[i,j]) for (i,j)in edges)

# Set up the problem to maximize using the objective function and
# keeping within the set constraints
prob = cp.Problem(cp.Maximize(objective), constraints)

# Returns a positive semidefinite matrix
print(prob.solve())

# To get the vectors, take square root of the matrix
x = sqrtm(X.value)

# Generate a random hyperplane
u = np.random.randn(5) # normal to random hyperplane

# Pick values according to which side of the hyperplane the vectors are on
x = np.sign(x @ u)
```

Appendix A

Answers to Exercises

Answer to Exercise 1 (on page 6)

The first vector of the orthogonal subspace is:

$$v_1 = x_1 = (1, 1, 1)$$

The second vector of the subspace is a projection of x_2 onto v_1.

$$v_2 = x_2 - \frac{x_2 v_1}{v_1 v_1} v_1$$

Substitute the values:

$$v_2 = (0, 1, 1) - \frac{(0, 1, 1)(1, 1, 1)}{(1, 1, 1)(1, 1, -1)}(1, 1, 1)$$

$$v_2 = (0, 1, 1) - (2/3)(1, 1, 1)$$

$$v_2 = (-2/3, 1/3, 1/3)$$

Normalize:

$$v_1 = v_1/\sqrt{|v_1|}$$

$$v_1 = (1, 1, 1)/\sqrt{|v_1|}$$

$$v_1 = (0.577, 0.577, 0.577)$$

$$v_2 = v_2/\sqrt{|v_2|}$$

$$v_2 = (0, 1, 1)\sqrt{|v_2|}$$

$$v_2 = (-0.816, 0.408, 0.408)$$

Answer to Exercise 2 (on page 14)

$$U = \begin{bmatrix} -0.70710678 & -0.70710678 \\ -0.70710678 & 0.70710678 \end{bmatrix}$$

$$Singularvalues = [3.464101623.16227766]$$

$$V^T = \begin{bmatrix} -0.408 & -0.816 & -0.408 \\ -0.894 & 0.447 & 0.0 \\ -0.183 & -0.365 & 0.9129 \end{bmatrix}$$

Answer to Exercise 3 (on page 21)

Yes. Its eigenvalues are

$$2, 2$$

Answer to Exercise 4 (on page 21)

The answer depends on the 3 by 3 matrix you chose.

Index