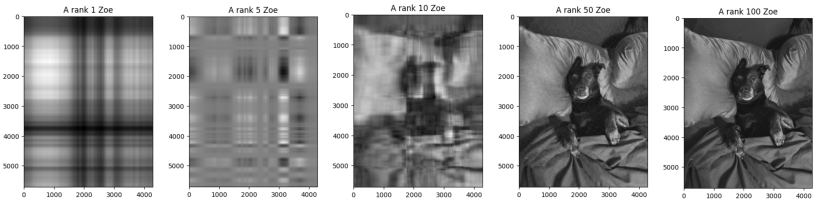
MATH 4042U Project 1

Brianna Menzie 100836923

February 14, 2025

1 Image Compression with Singular Value Decomposition

I am performing image compression using singular value decomposition on a picture of my dog, Zoe. There were no issues with any entries of my array, which was nice! The size of the array is 5712 $\times$ 4284, and its rank is 4284. I then calculate S, U, and V^T with `np.linalg.svd(img, full_matrices=False)`, where `img` is my array for the image, to do the closest *rank k* approximation for 5 different values of k ($k = 1$, 5, 10, 50, and 100).



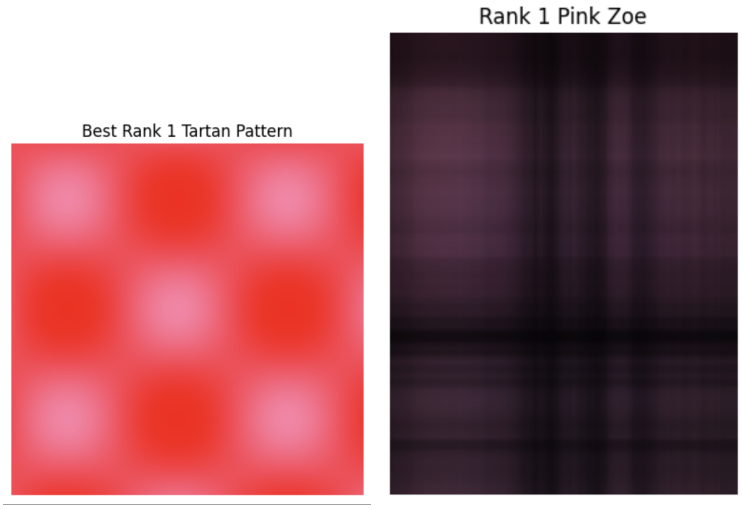
(a) Rank 1 (b) Rank 5 (c) Rank 10 (d) Rank 50 (e) Rank 100

Figure 1: Rank k approximations of Zoe

The bonus question asked to create the nicest rank-one tartan pattern and why I think it is nice. I am not sure if you wanted us to make the nicest rank-one tartan pattern based off an image or not, so I have created two that I think are very nice looking: the first is just from a patterned matrix and the second is using the image of Zoe.

- For the first tartan-pattern, I created a $300x300$ patterned matrix, where each element is the product of sine functions applied to the row and column indices (`sin(x) * sin(y)`). Then, when I do the rank one approximation, instead of using grayscale (which is a bit boring), I wanted to make it pink, so I used the three colour channels (red, green, and blue).

- For the second tartan-pattern, I wanted to make it pink, but instead did SVD for each colour channel, then layered/stacked the three rank 1 images to make it pink.
- Personally, I think the first one looks a lot cooler because of the gradient affect that the sine function gives it, but I wanted to create two rank-one tartan patterns, one via patterned matrix, and another via image compression.



(a) Rank-One of Pattern Matrix (b) Rank-One of Zoe

Figure 2: Bonus: Nicest Rank-One Tartan Patterns

The code is attached at the end of this LaTeX document.

2 Principal Component Analysis with NBA Player Statistics

2.1 Choosing your dataset

The dataset I chose to apply principal component analysis on is an NBA player statistics dataset from the 2023-2024 season. I chose this dataset for a few reasons:

1. I have done an exploratory data analysis on a similar dataset for a previous class, and I love continuing exploration on a dataset to get the absolute most out of it. Previously, I found basic linear trends between two features/variables.

2. Basketball is a great passion of mine. I have played basketball my whole life, so I find it fascinating now that after all of these years, I am able to analyze relationships and trends with a variety of techniques.

3. It's a very good dataset due to its size (number of players and number of features) and the type of data it contains – most is numeric, but we have a few categorical variables such as team and position.

I think PCA will provide some interesting insights to this data. If you think about what PCA will do with this dataset, it essentially will start with the mean vector, which in this case is the average NBA player (average points per game, average number of games played, etc.), then see how much each player varies from the average player (represented by the principal components). At first, I will apply PCA to the original dataset, then, based on my results, I may filter the data by position, then apply PCA again. The reason for this is that, an average NBA player won't provide too much significance, as each position varies in what their role is on the court. For example, on average, a point guard (PG) will have more assists than a center (C), and moreover, a center will have more rebounds than a point guard. Therefore, it may be beneficial if we filter based on the players' position, but let's find out!

2.1.1 About My Data

The dataset consists of more than 500 rows and 30 columns. The descriptions of the columns are listed below:

- **Rk**: Rank
- **Player**: Player's name
- **Pos**: Position
- **Age**: Player's age
- **Tm**: Team

- **G**: Games played
- **GS**: Games started
- **MP**: Minutes played per game
- **FG**: Field goals per game
- **FGA**: Field goal attempts per game
- **FG%**: Field goal percentage
- **3P**: 3-point field goals per game
- **3PA**: 3-point field goal attempts per game
- **3P%**: 3-point field goal percentage
- **2P**: 2-point field goals per game
- **2PA**: 2-point field goal attempts per game
- **2P%**: 2-point field goal percentage
- **eFG%**: Effective field goal percentage
- **FT**: Free throws per game
- **FTA**: Free throw attempts per game
- **FT%**: Free throw percentage
- **ORB**: Offensive rebounds per game
- **DRB**: Defensive rebounds per game
- **TRB**: Total rebounds per game
- **AST**: Assists per game
- **STL**: Steals per game
- **BLK**: Blocks per game
- **TOV**: Turnovers per game
- **PF**: Personal fouls per game
- **PTS**: Points per game

2.1.2 Cleaning My Data

This dataset was quite nice in the sense that there wasn't too much data cleaning to do. All I did was remove duplicate players (eg. if a player got traded to another team, they would show up twice in the dataset), attempt to drop all rows with "NaN" values (but there wasn't, so I didn't need to do this step), and lastly, I removed columns that I was not interested in for this analysis (for example: team/TM). After cleaning my data, it is now a 572 x 18 sized matrix.

	Age	G	MP	FG%	3P%	2P%	eFG%	ORB	DRB	TRB	AST	STL	BLK	TOV	PF	PTS
0	24	74	21.9	0.501	0.268	0.562	0.529	2.6	4.0	6.6	1.3	0.6	0.9	1.1	1.9	7.6
3	26	71	34.0	0.521	0.357	0.528	0.529	2.2	8.1	10.4	3.9	1.1	0.9	2.3	2.2	19.3
4	23	78	21.0	0.411	0.294	0.523	0.483	0.9	1.8	2.8	1.1	0.6	0.6	0.8	1.5	5.8
7	23	61	26.5	0.435	0.349	0.534	0.528	1.2	4.6	5.8	2.3	0.7	0.9	1.1	1.5	10.7
8	25	82	23.4	0.439	0.391	0.517	0.560	0.4	1.6	2.0	2.5	0.8	0.5	0.9	1.7	8.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
728	35	33	13.3	0.602	0.143	0.634	0.606	1.4	1.7	3.1	1.7	0.7	0.2	0.5	1.5	4.2
731	25	54	36.0	0.430	0.373	0.479	0.516	0.4	2.3	2.8	10.8	1.3	0.2	4.4	2.0	25.7
732	25	48	11.4	0.538	0.208	0.588	0.552	1.5	2.8	4.3	0.6	0.2	0.4	0.8	1.1	4.6
733	31	43	7.4	0.419	0.333	0.424	0.427	1.1	1.5	2.6	0.9	0.2	0.1	0.4	1.0	1.8
734	26	68	26.4	0.649	0.000	0.649	0.649	2.9	6.3	9.2	1.4	0.3	1.2	1.2	2.6	11.7

Figure 3: Dataset after cleaning

2.2 Constructing Your Data Matrix

Below, I calculate the mean vector, covariance matrix, and correlation matrix. As mentioned above, the mean vector represents the "average" NBA player from the 2023-2024 season across all 16 features.

The covariance matrix shows how two variables/features vary together. A positive covariance indicates that the two variables tend to increase or decrease together (proportional relationship), while a negative covariance suggests an inverse relationship (one increases as the other decreases). If the covariance is close to zero, it means there is little to no linear relationship between the two features. The correlation matrix is just a standardized version of the covariance matrix, making the relationships easier to interpret.

That being said, this means that if the covariance matrix and the correlation matrix look very different, this is a strong indicator that we should standardize our data, as the variables are on different scales. When looking at `cov_matrix` vs. `correlation_matrix`, we see that these matrices are extremely different, therefore indicating that we should standardize our data. This makes sense, because we have data on many different scales (for example: points per game vs. shooting percentages). When we standardize the data, we ensure that all variables have equal contributions when performing principal component analysis, as PCA relies on the variation of the data to determine the principal components. Therefore, if we did not standardize, the principal components would be dominated by the features that have a larger magnitude of variation.

2.3 Perform principal component analysis on your data matrix

Below are the first three principal components, which account for about 74% of the variation, along with the associated principal values.

	PC1	**PC2**	**PC3**
Age	0.0542	0.0799	-0.1486
G	0.2543	0.0394	-0.1912
MP	0.3275	0.1867	-0.0625
FG%	0.2020	-0.4762	-0.1798
3P%	0.0927	0.1077	-0.5170
2P%	0.1493	-0.4682	-0.2256
eFG%	0.1961	-0.4193	-0.3587
ORB	0.2321	-0.2208	0.4202
DRB	0.3227	-0.0023	0.2192
TRB	0.3149	-0.0695	0.2947
AST	0.2498	0.3139	-0.1671
STL	0.2646	0.2051	-0.0746
BLK	0.2319	-0.1661	0.3128
TOV	0.2949	0.2394	-0.0246
PF	0.2992	0.0239	0.0882
PTS	0.3085	0.2101	-0.0839

Table 1: Principal Component Loadings

Principal Component	**Explained Variance (%)**	**Cumulative Variance (%)**	**Eigenvalue**
PC1	48.17	48.17	7.7207
PC2	15.23	63.40	2.4409
PC3	10.92	74.32	1.7506

Table 2: Explained Variance and Eigenvalues

2.3.1 How do these vectors and scalars relate to your matrix in part 2?

The original data set is made up of players (rows) and NBA statistics (columns). Many of these features are correlated some how, meaning that there is redundancy in the data. For example, a player who scores more points per game on average likely has a higher field goal percentage. What principal component analysis (PCA) does is it transforms the original (standardized) data into a new sort of coordinate system, where the principal components (PCs) are like the axes – which capture the most variance in the data with as few dimensions as possible. The eigenvalues (λ_i) represent how much variance each PC accounts for or explains. A larger λ_i indicates more importance in capturing the dataset's structure/variation. The total variation is given by the sum of the eigenvalues, or:

$$\text{Total Variance} = \sum_{i=1}^{16} \lambda_i$$

Then, this means that the proportion of variation explained by each PC is given by:

$$\frac{\lambda_i}{\sum \lambda} \times 100$$

The first three principal components account for about 74% of the total variation, meaning they capture most of the meaningful structure and insights in the data.

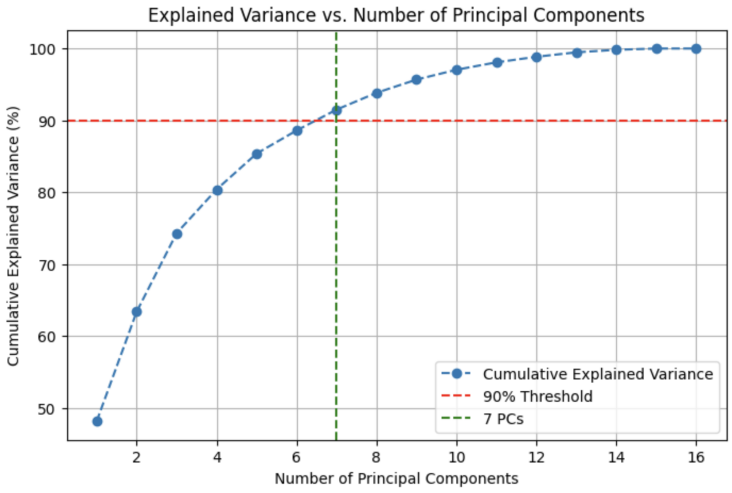


Figure 4: Explained Variance vs. Number of Principal Components

2.3.2 PC1: Overall Player Activity (48%)

- PC1, which accounts for 48% of the variation, is heavily influenced by all features, except Age, 3P% 2P%, and eFG%.
- The highest loading is minutes played (MP), which makes sense, as players who play more minutes tend to have more opportunities for scoring, rebounding, etc.
- All loadings are positive, meaning features increase together, or are proportional to one another. For example:
  - Players who play more minutes tend to score more points
  - Players who get more rebounds (TRB – total rebounds) generally do so on both, offensive and defensive ends.
  - These kinds of arguments and relationships can generally be made for most features.
- Shooting percentages do not contribute much to PC1 because they vary less among NBA players.
  - The lack of variation does not surprise me, as if you are good enough to be in the NBA, your shooting percentages have to be relatively high (around 40-50%).
  - Shooting percentages don't have large variation from one player to another, but what does is average number of points (or shots made).
  - If you think about it, two players could have the exact same 2P%, but Player 1 could have only made 4/10 shots, where as Player 2 could have made 40/100 shots.
  - Clearly, Player 2 plays and shoots a lot more than Player 1, meaning he's probably a better player, even though their 2P% is the same/similar.

2.3.3 PC2: Playmakers and Guards (15%)

- Field goal percentage, two point percentage, and effective field goal percentage (FG%, 2P%, and eFG%, respectively) have strong negative loadings. Therefore, this would indicate that players with a larger contribution of PC2 have lower shooting effiiency.
- Assists, steals, and turnovers (AST, STL, and TOV) have positive loadings, indicating that players with higher PC2 values can be seen as "playmakers", who handle the ball a lot.
- I think PC2 separates efficient scorers from ball handlers. What I mean by that is players with low PC2 values may excel in shooting efficiency, but also may not generate many plays (via assists or steals). On the contrary,

players with high PC2 values are likely to be pointguards, as they are seen as playmakers, who assist more, but also turn the ball over more frequently, as they typically take risks that sometimes don't pay off.

2.3.4 PC3: Centers vs. Perimeter Shooters (11%)

- - Three point percentage (3P%) has the strongest (negative) loading. Offensive rebounds, blocks, and total rebounds (ORB, BLK, and TRB) have positive loadings, meaning that players who are near the basket often (eg. centers) likely have a high PC3 value.
- From the difference in large, negative 3P% vs. large positve ORB/BLK/TRB, I think that PC3 distinguishes centers or "big men" from perimeter shooters.
  - Player with high PC3 values are strong rebounders and shot blockers, but don't shoot three-pointers often.
  - The opposite argument can be made for low PC3 values.

2.3.5 Conclusion

Although the first three principal components only account for 74% of the total variation, I believe they are the only ones that provide clear meaning and insights. To capture 90% or more of the variation, we would need to consider the first seven principal components. However, I find it difficult to point out significant insights for PCs 4-7.

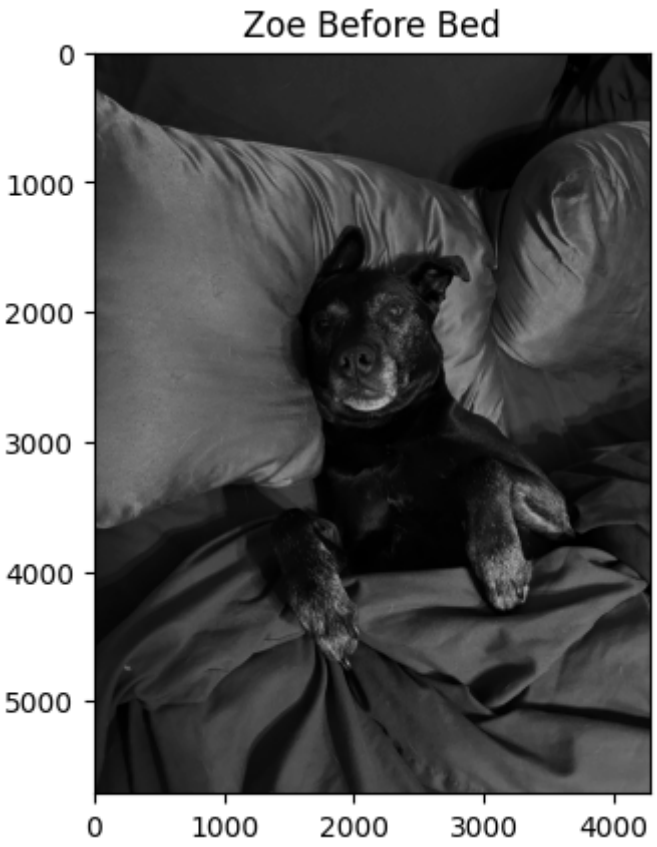
That being said, since the first seven principal components explain over 90% of the total variation, we can use PCA to reduce the dataset's dimension from 16 to 7. This reduction is allowed because the remaining nine components contribute less than 10% of the dataset's variation, meaning they likely capture redundant information. By keeping only seven dimensions, we make the dataset smaller, improving computational efficiency while keeping the integrity and accuracy of the original data.

MATH 4042U Project 1 - Image Compression

In [45]:
```
# importing my libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import cv2
```

In [46]:
```
# importing my image of my cutie little dog, Zoe

img = cv2.imread('zoe.jpg', cv2.IMREAD_GRAYSCALE)

print('The object type of the imported image is:', type(img))

plt.imshow(img, cmap='gray')
plt.title('Zoe Before Bed')
plt.show()
```

```
The object type of the imported image is: <class 'numpy.ndarray'>
```



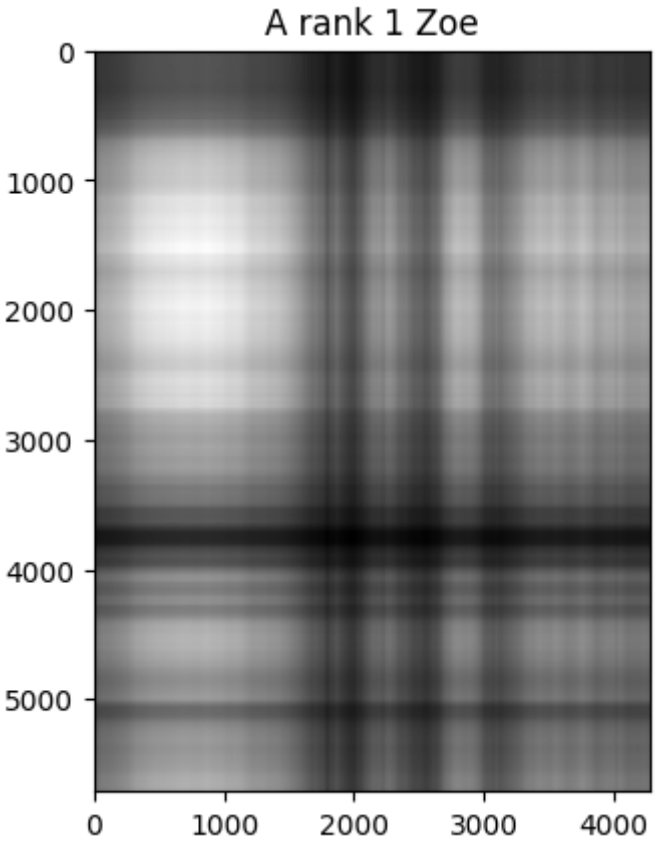
In [47]:
```
# now converting to array

print(img)
print('The size of this array is', np.shape(img))
print('The rank of this matrix is', np.linalg.matrix_rank(img))
```

```
[[ 17  11  11 ...  32  33  35]
 [ 18  11  12 ...  31  32  34]
 [ 18  11  12 ...  31  32  33]
 ...
 [ 22  22  19 ... 108 113 110]
 [ 20  22  21 ... 117 122 119]
 [ 20  23  23 ... 122 125 123]]
The size of this array is (5712, 4284)
The rank of this matrix is 4284
```

In [48]:
```
# calculating SVD

U, S, Vt = np.linalg.svd(img, full_matrices=False)
U.shape, S.shape, Vt.shape
```

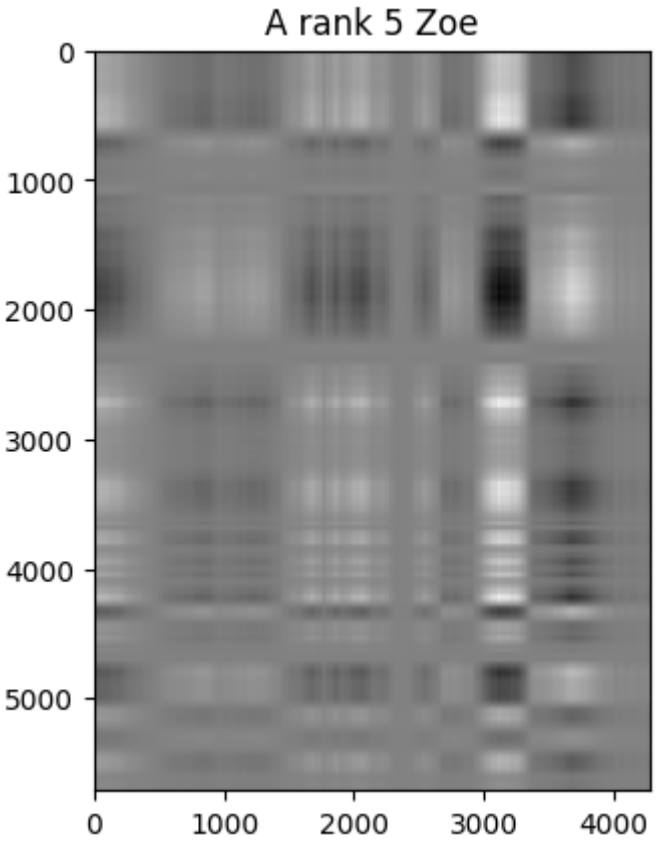
Out[48]: ((5712, 4284), (4284,), (4284, 4284))

In [49]:
```
# rank 1 image of zoe

reconstructed_image = np.matrix(U[:,:1]) * np.diag(S[:1]) * np.matrix(Vt[:1,
plt.imshow(reconstructed_image, cmap='gray')
plt.title('A rank 1 Zoe')
plt.show()
```

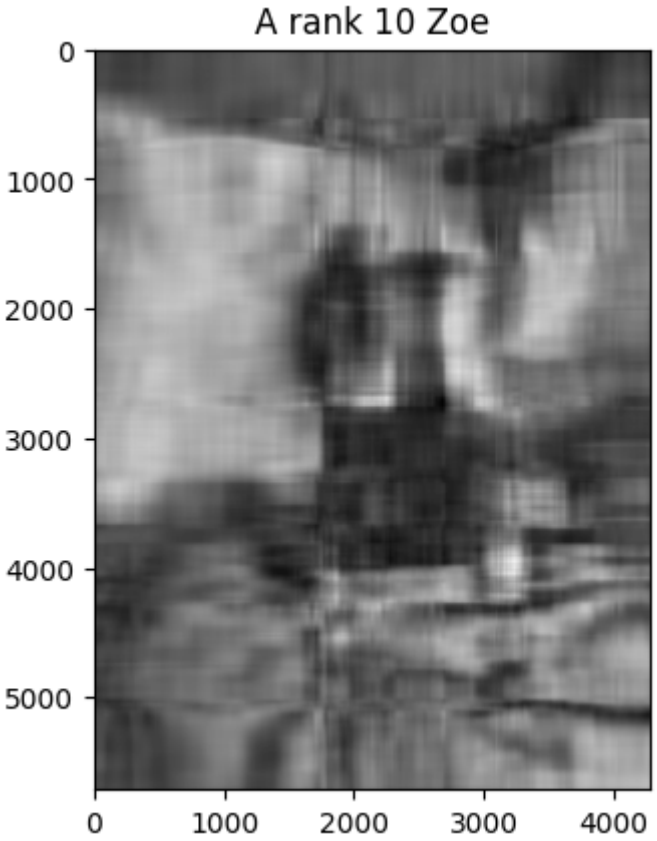


In [50]:
```
# rank 5 image of zoe

reconstructed_image = np.matrix(U[:,4:5]) * np.diag(S[4:5]) * np.matrix(Vt[4
plt.imshow(reconstructed_image, cmap='gray')
```

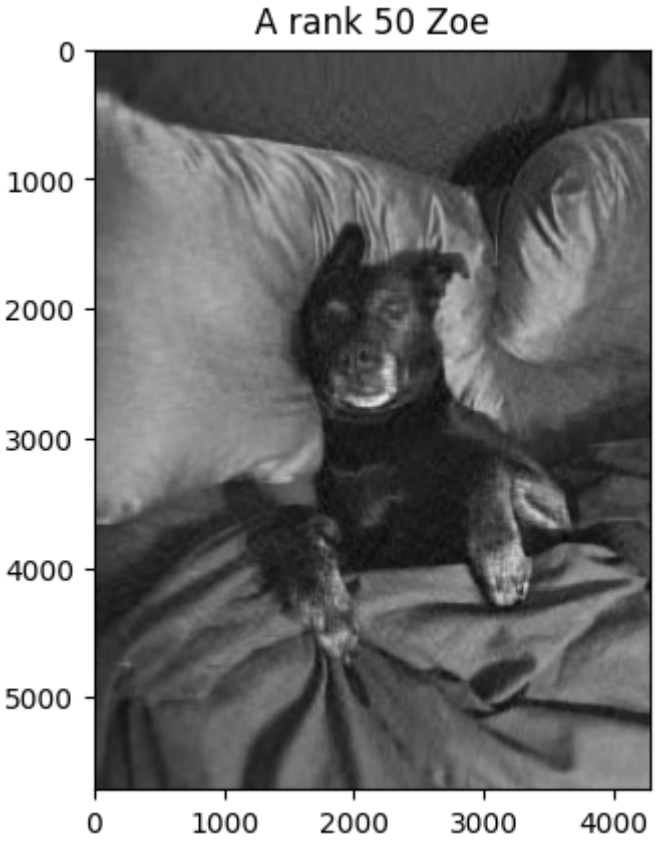
```
plt.title('A rank 5 Zoe')
plt.show()
```



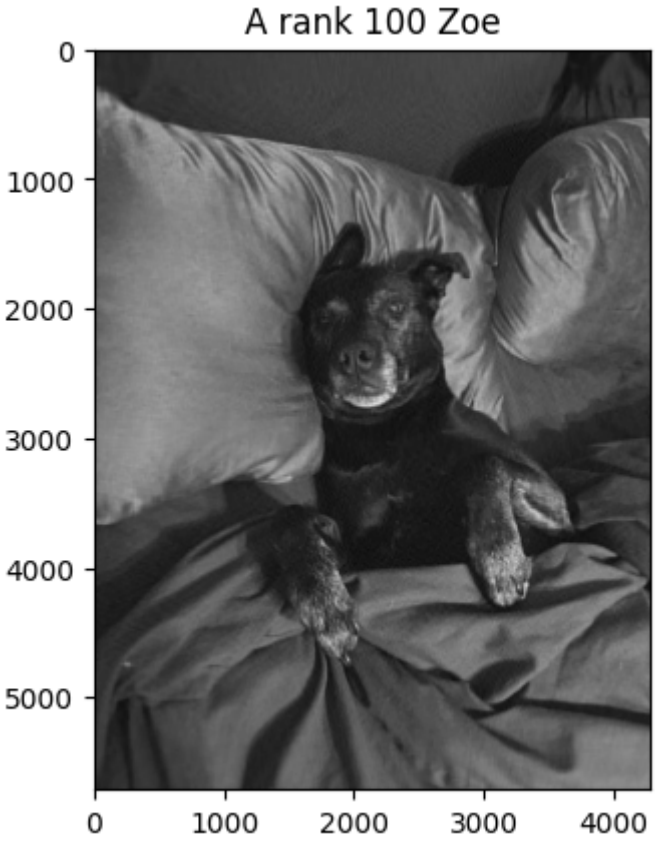
In [51]:
```
# rank 10 image of zoe

reconstructed_image = np.matrix(U[:,:10]) * np.diag(S[:10]) * np.matrix(Vt[:
plt.imshow(reconstructed_image, cmap='gray')
plt.title('A rank 10 Zoe')
plt.show()
```



In [52]:
```
# rank 50 image of zoe

reconstructed_image = np.matrix(U[:,:50]) * np.diag(S[:50]) * np.matrix(Vt[:
plt.imshow(reconstructed_image, cmap='gray')
plt.title('A rank 50 Zoe')
plt.show()
```



In [53]:
```
# rank 100 image of zoe

reconstructed_image = np.matrix(U[:,:100]) * np.diag(S[:100]) * np.matrix(Vt
plt.imshow(reconstructed_image, cmap='gray')
plt.title('A rank 100 Zoe')
plt.show()
```



BONUS: Make the nicest rank-one tartan pattern you can and why you think it is nice.

I am not sure if you wanted us to make the nicest rank-one tartan pattern based off an image or not, so I have created two that I think are very nice looking: the first is just from a patterned matrix and the second is using the image of Zoe.

For the first tartan-pattern, I created a 300x300 patterned matrix, where each element is the product of sine functions applied to the row and column indices (`sin(x) * sin(y)`). Then, when I do the rank one approximation, instead of using grayscale (which is a bit boring), I wanted to make it pink, so I used the three colour channels (red, green, and blue).

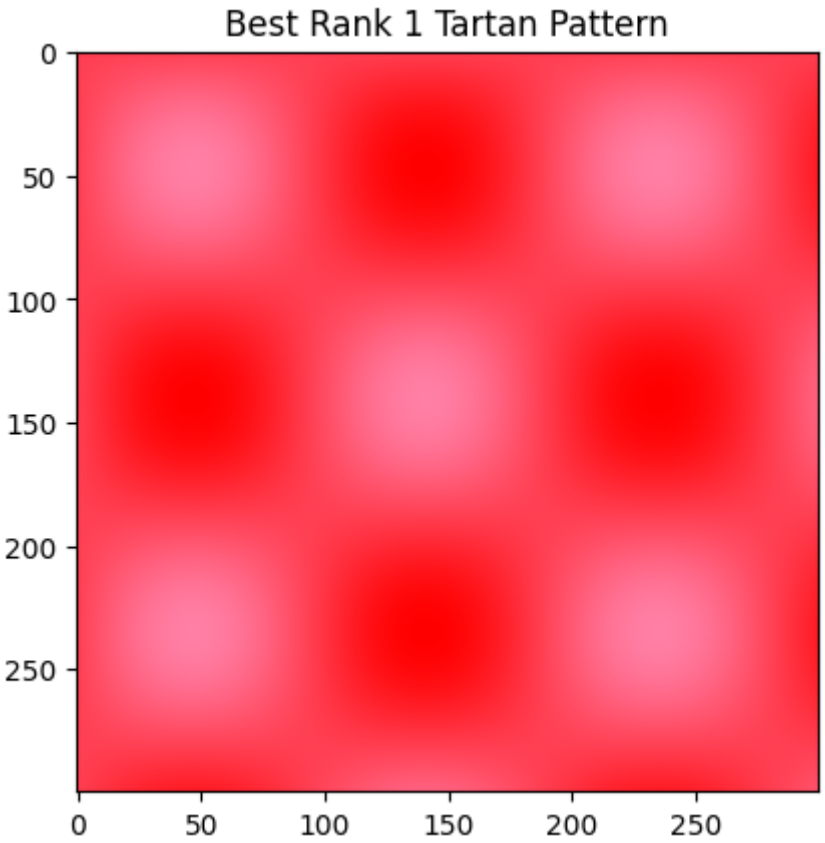
For the second tartan-pattern, I wanted to make it pink, but instead did SVD for each colour channel, then layered/stacked the three rank 1 images to make it pink.

Personally, I think the first one looks a lot cooler because of the gradient affect that the sine function gives it, but I wanted to create two rank-one tartan patterns, one via patterned matrix, and another via image compression.

In [54]:
```
# create matrix with pattern in numbers
rows = 300
columns = 300
x = np.linspace(0,10,rows)
y = np.linspace(0,10,columns)
```

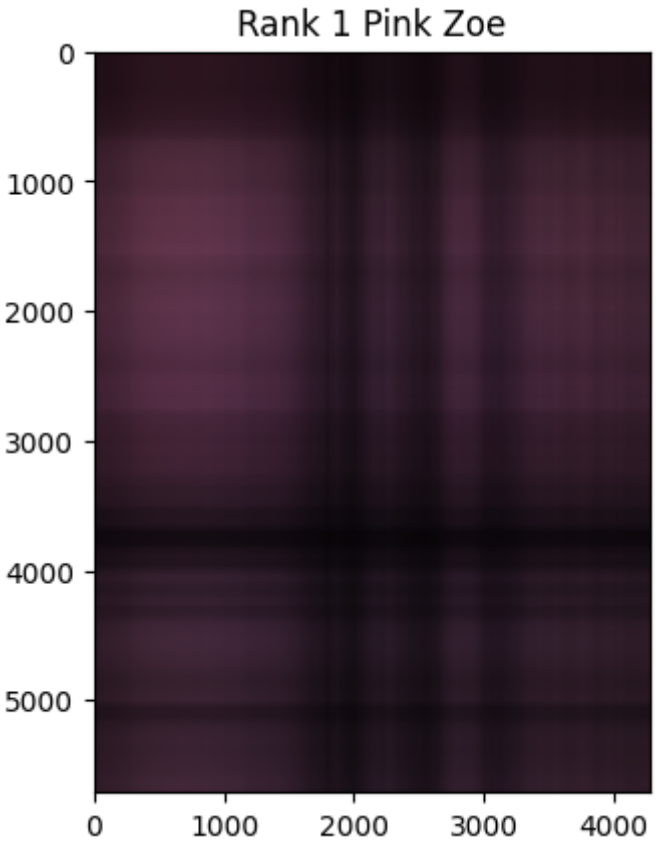
```
A = np.sin(x).reshape(-1,1) * np.sin(y).reshape(1,-1)
print(A)

# svd
U, S, Vt = np.linalg.svd(A, full_matrices=False)

# rank 1 approx
rank_one_A = np.matrix(U[:,:1]) * np.diag(S[:1]) * np.matrix(Vt[:1,:])

# normalize values to [0,1] for colour mapping
rank_one_A = (rank_one_A - rank_one_A.min()) / (rank_one_A.max() - rank_one_

# making it pink
pink_A = np.zeros((rows, columns, 3)) # empty 3D array for RGB
pink_A[:, :, 0] = 1.0  # red channel 100%
pink_A[:, :, 1] = 0.5 * rank_one_A  # reduce green channel (50%) for pink
pink_A[:, :, 2] = 0.65 * rank_one_A  # reduce blue channel (65%) for pink

# display
plt.imshow(pink_A)
plt.title("Best Rank 1 Tartan Pattern")
plt.show()
```

```
[[ 0.          0.          0.         ... -0.         -0.
  -0.        ]
 [ 0.          0.00111814  0.00223503 ... -0.01627527 -0.01724292
  -0.01819129]
 [ 0.          0.00223503  0.00446755 ... -0.03253233 -0.03446656
  -0.03636224]
 ...
 [-0.         -0.01627527 -0.03253233 ...  0.23689751  0.25098241
   0.2647866 ]
 [-0.         -0.01724292 -0.03446656 ...  0.25098241  0.26590474
   0.28052966]
 [-0.         -0.01819129 -0.03636224 ...  0.2647866   0.28052966
   0.29595897]]
```



In [55]:
```
# Load image
colour_image = cv2.imread('zoe.jpg')

# Convert BGR to RGB (since OpenCV loads images in BGR format)
colour_image = cv2.cvtColor(colour_image, cv2.COLOR_BGR2RGB)

# Split into R, G, B channels
B, G, R = cv2.split(colour_image)

# SVD for each channel
U_R, S_R, Vt_R = np.linalg.svd(R, full_matrices=False)
U_G, S_G, Vt_G = np.linalg.svd(G, full_matrices=False)
U_B, S_B, Vt_B = np.linalg.svd(B, full_matrices=False)

n = 1  # rank 1 approx

# getting the RGB rank 1 approx's
R_compressed = (U_R[:, :n] @ np.diag(S_R[:n]) @ Vt_R[:n, :])
G_compressed = (U_G[:, :n] @ np.diag(S_G[:n]) @ Vt_G[:n, :])
B_compressed = (U_B[:, :n] @ np.diag(S_B[:n]) @ Vt_B[:n, :])

# reducing green and blue
G_compressed *= 0.5
B_compressed *= 0.65

# stack the red, green, and blue colour channels to get pink
pink_image = np.dstack((R_compressed, G_compressed, B_compressed))

# normalize for proper display
```

```
pink_image = np.clip(pink_image / 255.0, 0, 1)

# display
plt.imshow(pink_image)
plt.title('Rank 1 Pink Zoe')
plt.show()
```



MATH 4042U Project 1 - Principal Component Analysis on NBA Statistics

1. Choosing your dataset.

The dataset I chose to apply principal component analysis on is an NBA player statistics dataset from the 2023-2024 season. I chose this dataset for a few reasons:

1. I have done an exploratory data analysis on a similar dataset for a previous class, and I love continuing exploration on a dataset to get the absolute most out of it. Previously, I found basic linear trends between two feautures/variables.
2. Basketball is a great passion of mine. I have played basketball my whole life, so I find it fascinating now that after all of these years, I am able to analyze relationships and trends with a variety of techniques.
3. It's a very good dataset due to its size (number of players and number of feautures) and the type of data it contains -- most is numeric, but we have a few categorical variables such as team and position.

I think PCA will provide some interesting insights to this data. If you think about what PCA will do with this dataset, it essentially will start with the mean vector, which in this case is the average NBA player (average points per game, average number of games played, etc.), then see how much each player varies from the average player (represented by the principal components). At first, I will apply PCA to the original dataset, then, based on my results, I may filter the data by position, then apply PCA again. The reason for this is that, an average NBA player won't provide too much significance, as each position varies in what their role is on the court. For example, on average, a point guard (PG) will have more assists than a center (C), and moreover, a center will have more rebounds than a point guard. Therefore, it may be beneficial if we filter based on the players' position, but let's find out!

In [2]:
```
# importing my libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from IPython.display import display
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler


# importing dataset from kaggle (https://www.kaggle.com/datasets/vivovinco/2
data = pd.read_csv("regular_player_stats.csv", sep = ";", encoding="ISO-8859
display(data) # 735 x 30
```

	Rk	Player	Pos	Age	Tm	G	GS	MP	FG	FGA	...	FT%	ORB	DRB	T
0	1	Precious Achiuwa	PF-C	24	TOT	74	18	21.9	3.2	6.3	...	0.616	2.6	4.0	6
1	1	Precious Achiuwa	C	24	TOR	25	0	17.5	3.1	6.8	...	0.571	2.0	3.4	5
2	1	Precious Achiuwa	PF	24	NYK	49	18	24.2	3.2	6.1	...	0.643	2.9	4.3	
3	2	Bam Adebayo	C	26	MIA	71	71	34.0	7.5	14.3	...	0.755	2.2	8.1	10
4	3	Ochai Agbaji	SG	23	TOT	78	28	21.0	2.3	5.6	...	0.661	0.9	1.8	2
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
730	568	Thaddeus Young	PF	35	PHO	10	0	8.9	1.1	2.1	...	0.333	1.7	1.1	2
731	569	Trae Young	PG	25	ATL	54	54	36.0	8.0	18.7	...	0.855	0.4	2.3	2
732	570	Omer Yurtseven	C	25	UTA	48	12	11.4	2.1	3.8	...	0.679	1.5	2.8	4
733	571	Cody Zeller	C	31	NOP	43	0	7.4	0.6	1.4	...	0.605	1.1	1.5	2
734	572	Ivica Zubac	C	26	LAC	68	68	26.4	5.0	7.6	...	0.723	2.9	6.3	9

735 rows × 30 columns

About My Dataset

+500 rows and 30 columns. Columns' description are listed below.

Rk : Rank

Player : Player's name

Pos : Position

Age : Player's age

Tm : Team

G : Games played

GS : Games started

MP : Minutes played per game

FG : Field goals per game

FGA : Field goal attempts per game

FG% : Field goal percentage

3P : 3-point field goals per game

3PA : 3-point field goal attempts per game

3P% : 3-point field goal percentage

2P : 2-point field goals per game

2PA : 2-point field goal attempts per game

2P% : 2-point field goal percentage

eFG% : Effective field goal percentage

FT : Free throws per game

FTA : Free throw attempts per game

FT% : Free throw percentage

ORB : Offensive rebounds per game

DRB : Defensive rebounds per game

TRB : Total rebounds per game

AST : Assists per game

STL : Steals per game

BLK : Blocks per game

TOV : Turnovers per game

PF : Personal fouls per game

PTS : Points per game

In [3]:
```
# cleaning up my data

# removing duplicate players (if they got traded to another team, they would
data = data.drop_duplicates("Player")
# 572 x 30 dataset

# drop all rows with NaN values
data = data.dropna()
# there were no NaN values, so size remains the samne
```

```
# now, I am removing columns that I am not interested in (not needed for thi
data = data.drop(data.columns[[0, 1, 2, 4, 6, 8, 9, 11, 12, 14, 15, 18, 19,
# 572 x 18 dataset

display(data)
```

	Age	G	MP	FG%	3P%	2P%	eFG%	ORB	DRB	TRB	AST	STL	BLK	TOV
0	24	74	21.9	0.501	0.268	0.562	0.529	2.6	4.0	6.6	1.3	0.6	0.9	1.1
3	26	71	34.0	0.521	0.357	0.528	0.529	2.2	8.1	10.4	3.9	1.1	0.9	2.3
4	23	78	21.0	0.411	0.294	0.523	0.483	0.9	1.8	2.8	1.1	0.6	0.6	0.8
7	23	61	26.5	0.435	0.349	0.534	0.528	1.2	4.6	5.8	2.3	0.7	0.9	1.1
8	25	82	23.4	0.439	0.391	0.517	0.560	0.4	1.6	2.0	2.5	0.8	0.5	0.9
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
728	35	33	13.3	0.602	0.143	0.634	0.606	1.4	1.7	3.1	1.7	0.7	0.2	0.5
731	25	54	36.0	0.430	0.373	0.479	0.516	0.4	2.3	2.8	10.8	1.3	0.2	4.4
732	25	48	11.4	0.538	0.208	0.588	0.552	1.5	2.8	4.3	0.6	0.2	0.4	0.8
733	31	43	7.4	0.419	0.333	0.424	0.427	1.1	1.5	2.6	0.9	0.2	0.1	0.4
734	26	68	26.4	0.649	0.000	0.649	0.649	2.9	6.3	9.2	1.4	0.3	1.2	1.2

572 rows × 16 columns

2. Constructing your data matrix.

Below, I calculate the mean vector, covariance matrix, and correlation matrix. As mentioned above, the mean vector represents the "average" NBA player from the 2023-2024 season across all 16 features.

The covariance matrix shows how two variables/features vary together. A positive covariance indicates that the two variables tend to increase or decrease together (proportional relationship), while a negative covariance suggests an inverse relationship (one increases as the other decreases). If the covariance is close to zero, it means there is little to no linear relationship between the two features. The correlation matrix is just a standardized version of the covariance matrix, making the relationships easier to interpret.

That being said, this means that if the covariance matrix and the correlation matrix look very different, this is a strong indicator that we should standardize our data, as the variables are on different scales. When looking at `cov_matrix` vs. `correlation_matrix`, we see that these matrices are extremely different, therefore indicating that we should standardize our data. This makes sense, because we have data on many different scales (for example: points per game vs. shooting percentages). When we standardize the data, we ensure that all variables have equal contributions when

performing principal component analysis, as PCA relies on the variation of the data to determine the principal components. Therefore, if we did not standardize, the principal components would be dominated by the feautures that have a larger magnitude of variation.

In [4]:
```
# converting my dataset into a matrix
# row: player
# column: unique variables for each player/row

A = data.to_numpy()
print(A)
print(A.shape)
```

```
[[24.  74.  21.9 ...  1.1  1.9  7.6]
 [26.  71.  34.  ...  2.3  2.2 19.3]
 [23.  78.  21.  ...  0.8  1.5  5.8]
 ...
 [25.  48.  11.4 ...  0.8  1.1  4.6]
 [31.  43.   7.4 ...  0.4  1.   1.8]
 [26.  68.  26.4 ...  1.2  2.6 11.7]]
(572, 16)
```

Mean Vector, Covariance Matrix, and Correlation Matrix

In [5]:
```
# mean vector
mean_vector = A.mean(axis=0)

# centre data (subtract mean from each column)
A_cent = A - mean_vector

# covariance matrix
n = A.shape[0]  # Number of observations (rows)
cov_matrix = (1 / (n - 1)) * np.matmul(np.transpose(A_cent), A_cent)

# correlation matrix
std_dev = np.std(A, axis=0, ddof=1)  # Standard deviation of each column
correlation_matrix = cov_matrix / np.outer(std_dev, std_dev)

# print results
print("Mean Vector:\n", mean_vector)
print("\nCovariance Matrix:\n", cov_matrix)
print("\nCorrelation Matrix:\n", correlation_matrix)
```

```
Mean Vector:
 [25.74300699 46.15384615 18.6506993   0.44958392  0.29948077  0.51967483
  0.51649825  0.85769231  2.5229021   3.37307692  2.0013986   0.59125874
  0.4027972   0.98461538  1.49090909  8.42325175]

Covariance Matrix:
 [[ 1.78515272e+01  1.85702546e+01  6.84212399e+00  1.98333374e-02
   5.61062576e-02 -4.72994869e-03  4.16729300e-02  1.01239391e-02
   8.86981801e-01  8.94119628e-01  1.44764552e+00  2.15613021e-01
   3.69723097e-02  2.62764381e-01  2.71389906e-01  3.39933101e+00]
 [ 1.85702546e+01  6.52084871e+02  1.77454358e+02  1.00908689e+00
   1.37325515e+00  9.35782164e-01  1.25733127e+00  6.01597737e+00
   2.55236158e+01  3.15150074e+01  2.23148457e+01  4.83374646e+00
   3.63266873e+00  1.01134043e+01  1.18807356e+01  1.02654385e+02]
 [ 6.84212399e+00  1.77454358e+02  9.81470923e+01  3.29542673e-01
   4.71693446e-01  2.75185166e-01  3.97839318e-01  3.15502088e+00
   1.41250890e+01  1.72626283e+01  1.38910673e+01  2.96969089e+00
   1.88999804e+00  6.42824195e+00  6.08436714e+00  6.00340555e+01]
 [ 1.98333374e-02  1.00908689e+00  3.29542673e-01  1.28498266e-02
   1.74827745e-03  1.28690466e-02  1.22305947e-02  4.08513674e-02
   8.71584075e-02  1.28013472e-01  3.07475182e-02  9.02280137e-03
   1.93622868e-02  2.16723966e-02  3.46799236e-02  2.18585173e-01]
 [ 5.61062576e-02  1.37325515e+00  4.71693446e-01  1.74827745e-03
   2.23378157e-02  6.20186380e-04  6.33773727e-03 -1.84740199e-02
   2.69246969e-02  8.20561094e-03  7.42241951e-02  1.52908999e-02
  -1.25861512e-03  2.37399838e-02  1.65513135e-02  3.03472164e-01]
 [-4.72994869e-03  9.35782164e-01  2.75185166e-01  1.28690466e-02
   6.20186380e-04  1.90154492e-02  1.26522814e-02  2.95462886e-02
   6.91002798e-02  9.85968342e-02  1.72533453e-02  7.48717010e-03
   1.60145714e-02  1.56464772e-02  2.89401210e-02  1.72462215e-01]
 [ 4.16729300e-02  1.25733127e+00  3.97839318e-01  1.22305947e-02
   6.33773727e-03  1.26522814e-02  1.36888494e-02  2.77050047e-02
   7.64787616e-02  1.04072632e-01  3.65530672e-02  1.09449934e-02
   1.56066599e-02  2.04484844e-02  3.40470944e-02  2.44008885e-01]
 [ 1.01239391e-02  6.01597737e+00  3.15502088e+00  4.08513674e-02
  -1.84740199e-02  2.95462886e-02  2.77050047e-02  5.56805874e-01
   9.88746464e-01  1.54402533e+00  1.93807086e-01  9.57591270e-02
   2.03078270e-01  1.99453051e-01  3.31471103e-01  1.71755288e+00]
 [ 8.86981801e-01  2.55236158e+01  1.41250890e+01  8.71584075e-02
   2.69246969e-02  6.91002798e-02  7.64787616e-02  9.88746464e-01
   3.35511380e+00  4.34063519e+00  1.85688560e+00  3.93703171e-01
   4.99585563e-01  1.02942476e+00  1.02416653e+00  9.23902873e+00]
 [ 8.94119628e-01  3.15150074e+01  1.72626283e+01  1.28013472e-01
   8.20561094e-03  9.85968342e-02  1.04072632e-01  1.54402533e+00
   4.34063519e+00  5.88239122e+00  2.04930217e+00  4.89711707e-01
   7.01791728e-01  1.23030311e+00  1.35551664e+00  1.09529038e+01]
 [ 1.44764552e+00  2.23148457e+01  1.38910673e+01  3.07475182e-02
   7.42241951e-02  1.72533453e-02  3.65530672e-02  1.93807086e-01
   1.85688560e+00  2.04930217e+00  3.50966529e+00  4.88068289e-01
   1.47474190e-01  1.25770982e+00  7.17858621e-01  9.93905674e+00]
 [ 2.15613021e-01  4.83374646e+00  2.96969089e+00  9.02280137e-03
   1.52908999e-02  7.48717010e-03  1.09449934e-02  9.57591270e-02
   3.93703171e-01  4.89711707e-01  4.88068289e-01  1.52182651e-01
   5.76076813e-02  2.05136737e-01  1.93387996e-01  1.79836473e+00]
 [ 3.69723097e-02  3.63266873e+00  1.88999804e+00  1.93622868e-02
  -1.25861512e-03  1.60145714e-02  1.56066599e-02  2.03078270e-01
```

```
  4.99585563e-01  7.01791728e-01  1.47474190e-01  5.76076813e-02
  1.70745227e-01  1.24614037e-01  1.78554370e-01  1.12998739e+00]
 [ 2.62764381e-01  1.01134043e+01  6.42824195e+00  2.16723966e-02
  2.37399838e-02  1.56464772e-02  2.04484844e-02  1.99453051e-01
  1.02942476e+00  1.23030311e+00  1.25770982e+00  2.05136737e-01
  1.24614037e-01  6.33318065e-01  4.14133100e-01  4.74804661e+00]
 [ 2.71389906e-01  1.18807356e+01  6.08436714e+00  3.46799236e-02
  1.65513135e-02  2.89401210e-02  3.40470944e-02  3.31471103e-01
  1.02416653e+00  1.35551664e+00  7.17858621e-01  1.93387996e-01
  1.78554370e-01  4.14133100e-01  6.19707053e-01  3.44054450e+00]
 [ 3.39933101e+00  1.02654385e+02  6.00340555e+01  2.18585173e-01
  3.03472164e-01  1.72462215e-01  2.44008885e-01  1.71755288e+00
  9.23902873e+00  1.09529038e+01  9.93905674e+00  1.79836473e+00
  1.12998739e+00  4.74804661e+00  3.44054450e+00  4.61232588e+01]]

Correlation Matrix:
 [[ 1.          0.17211872  0.16346114  0.04141041  0.08884915 -0.00811831
  0.08430103  0.00321115  0.11461023  0.08725312  0.18289073  0.13081415
  0.02117701  0.078148    0.08159486  0.11846656]
 [ 0.17211872  1.          0.70144866  0.34860039  0.35981463  0.26574763
  0.42083726  0.3157201   0.54567808  0.50884838  0.46645417  0.48523178
  0.34427021  0.49766187  0.59101438  0.59192332]
 [ 0.16346114  0.70144866  1.          0.29344337  0.31856699  0.20143411
  0.34323021  0.42678743  0.77839287  0.71844123  0.74845216  0.7684046
  0.46168775  0.81534715  0.78015963  0.89227551]
 [ 0.04141041  0.34860039  0.29344337  1.          0.10319089  0.82327409
  0.92217951  0.48295433  0.41976577  0.46561835  0.14478667  0.20403759
  0.41336503  0.24024113  0.38862996  0.28393052]
 [ 0.08884915  0.35981463  0.31856699  0.10319089  1.          0.03009183
  0.36243523 -0.16564913  0.09835056  0.02263673  0.26508936  0.26225883
 -0.02037972  0.19959476  0.14067564  0.29897751]
 [-0.00811831  0.26574763  0.20143411  0.82327409  0.03009183  1.
  0.78420946  0.28714267  0.27357278  0.29480356  0.06678627  0.1391816
  0.28105269  0.14257794  0.26659612  0.18415379]
 [ 0.08430103  0.42083726  0.34323021  0.92217951  0.36243523  0.78420946
  1.          0.31733856  0.35686483  0.36675506  0.166766    0.23980016
  0.32281389  0.21961738  0.36966077  0.30708753]
 [ 0.00321115  0.3157201   0.42678743  0.48295433 -0.16564913  0.28714267
  0.31733856  1.          0.72340151  0.85315033  0.13863883  0.32896198
  0.65862357  0.33587513  0.56428766  0.3389208 ]
 [ 0.11461023  0.54567808  0.77839287  0.41976577  0.09835056  0.27357278
  0.35686483  0.72340151  1.          0.97706331  0.54112634  0.55097604
  0.6600578   0.70620379  0.71027043  0.74269876]
 [ 0.08725312  0.50884838  0.71844123  0.46561835  0.02263673  0.29480356
  0.36675506  0.85315033  0.97706331  1.          0.45102041  0.51758414
  0.70025627  0.63741787  0.70996131  0.66495511]
 [ 0.18289073  0.46645417  0.74845216  0.14478667  0.26508936  0.06678627
  0.166766    0.13863883  0.54112634  0.45102041  1.          0.66782884
  0.1905061   0.84360018  0.48675772  0.78118218]
 [ 0.13081415  0.48523178  0.7684046   0.20403759  0.26225883  0.1391816
  0.23980016  0.32896198  0.55097604  0.51758414  0.66782884  1.
  0.35737448  0.66076942  0.62972904  0.67878965]
 [ 0.02117701  0.34427021  0.46168775  0.41336503 -0.02037972  0.28105269
  0.32281389  0.65862357  0.6600578   0.70025627  0.1905061   0.35737448
  1.          0.37894985  0.54891232  0.40266077]
 [ 0.078148    0.49766187  0.81534715  0.24024113  0.19959476  0.14257794
```

```
  0.21961738  0.33587513  0.70620379  0.63741787  0.84360018  0.66076942
  0.37894985  1.          0.66105225  0.87850461]
[ 0.08159486  0.59101438  0.78015963  0.38862996  0.14067564  0.26659612
  0.36966077  0.56428766  0.71027043  0.70996131  0.48675772  0.62972904
  0.54891232  0.66105225  1.          0.64353766]
[ 0.11846656  0.59192332  0.89227551  0.28393052  0.29897751  0.18415379
  0.30708753  0.3389208   0.74269876  0.66495511  0.78118218  0.67878965
  0.40266077  0.87850461  0.64353766  1.        ]]
```

3. Perform principal component analysis on your data matrix.

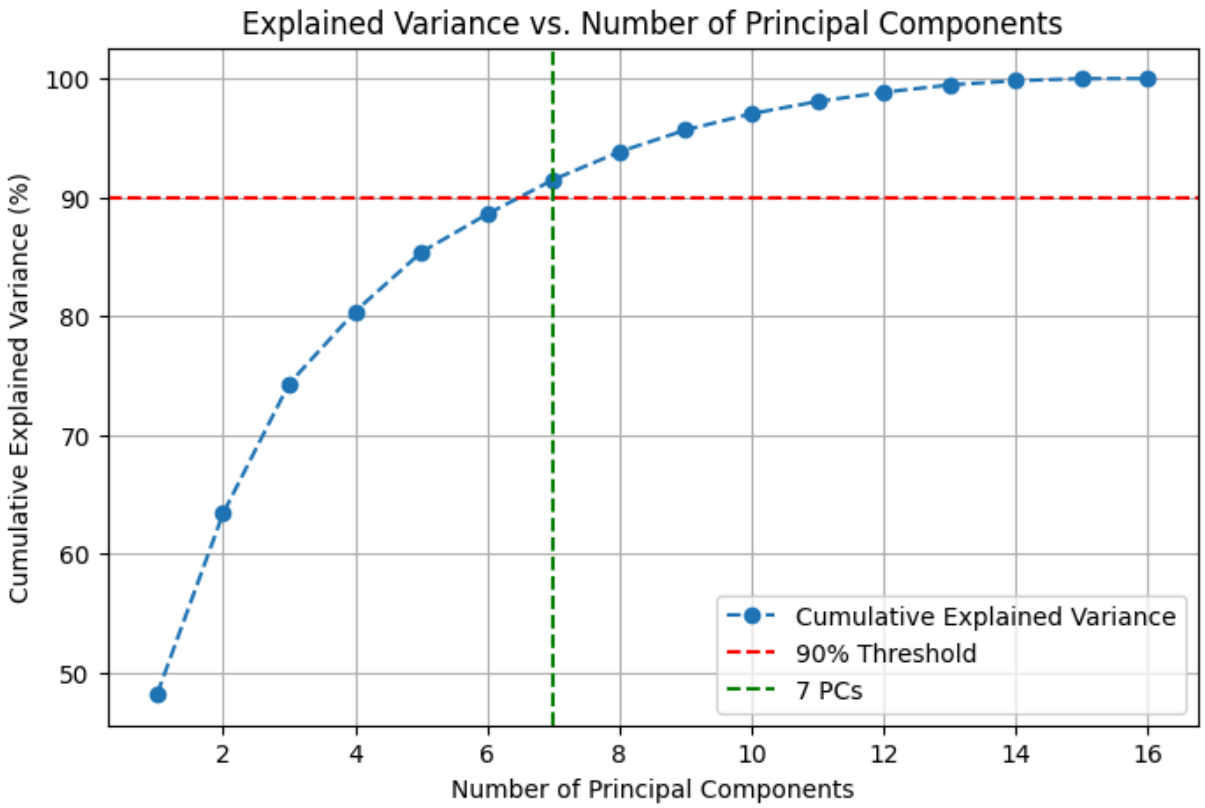
- List the first three principal components and the associated principal values
- How do these vectors and scalars relate to your matrix in part 2?
- Discuss your findings/interpret your principal components
- Can you use PCA for your dataset to perform dimension reduction? What is the resulting dimension of your data?

In [6]:
```
# applying pca using sklearn's pca
# more efficient
# centres your data for you prior to pca automatically
# has built-in methods to analyze explained variance and principal component


# standardize dataset
data_stand = StandardScaler().fit_transform(data)

# apply principal component analysis to the standardized data
pca = PCA(n_components=16)  # number of principal components (feautures/vari
X_pca = pca.fit_transform(data_stand)

# organize pca results
explained_variance = pca.explained_variance_ratio_ * 100
cumulative_variance = np.cumsum(pca.explained_variance_ratio_) * 100
eigenvalues = pca.explained_variance_

# printing data frame for results
pca_results_df = pd.DataFrame({
    'Principal Component': [f'PC{i+1}' for i in range(16)],
    'Explained Variance (%)': explained_variance,
    'Cumulative Variance (%)': cumulative_variance,
    'Eigenvalue/Principal Value': eigenvalues})
print("\nPCA:")
display(pca_results_df)

# finding number of components that explains 90% of the variance
num_components = np.argmax(cumulative_variance >= 90) + 1

# plot cumulative variance cumulative variance
plt.figure(figsize=(8, 5))
plt.plot(range(1, 17), cumulative_variance, marker='o', linestyle='--', labe
plt.axhline(y=90, color='r', linestyle='--', label="90% Threshold")
plt.axvline(x=num_components, color='g', linestyle='--', label=f"{num_compor
plt.xlabel('Number of Principal Components')
plt.ylabel('Cumulative Explained Variance (%)')
```

```
plt.title('Explained Variance vs. Number of Principal Components')
plt.legend()
plt.grid(True)
plt.show()

# print pca transformed data
pca_df = pd.DataFrame(X_pca, columns=[f'PC{i+1}' for i in range(16)])
print("\nFirst 5 Rows of PCA Transformed Data:") # only first 5 rows/players
display(pca_df.head())

# print pca vectors / eigenvectors (rows = PCs, columns = original feature n
pca_vectors = pd.DataFrame(pca.components_, index=[f'PC{i+1}' for i in range
print("\nPrincipal Component Vectors:")
display(pca_vectors)
```

PCA:

	Principal Component	Explained Variance (%)	Cumulative Variance (%)	Eigenvalue/Principal Value
0	PC1	48.170109	48.170109	7.720715
1	PC2	15.229227	63.399335	2.440944
2	PC3	10.922021	74.321356	1.750584
3	PC4	6.084492	80.405849	0.975224
4	PC5	4.984537	85.390386	0.798923
5	PC6	3.208806	88.599192	0.514308
6	PC7	2.902970	91.502161	0.465289
7	PC8	2.339008	93.841169	0.374897
8	PC9	1.833724	95.674893	0.293910
9	PC10	1.373592	97.048486	0.220160
10	PC11	1.026841	98.075327	0.164582
11	PC12	0.776257	98.851584	0.124419
12	PC13	0.619614	99.471197	0.099312
13	PC14	0.348238	99.819436	0.055816
14	PC15	0.179032	99.998468	0.028695
15	PC16	0.001532	100.000000	0.000246



First 5 Rows of PCA Transformed Data:

	PC1	PC2	PC3	PC4	PC5	PC6	PC7	PC
0	2.076403	-1.258096	1.793435	0.051427	1.079515	-0.352869	-0.401997	-0.31680
1	5.396389	0.535067	1.636788	0.048488	0.439007	0.943997	-0.677518	-0.64137
2	-0.141558	-0.048921	0.155520	-0.407637	0.892979	-1.162661	-0.137436	0.56410
3	1.742859	-0.034753	0.764125	-0.477279	0.874243	0.303255	-0.209123	0.45795
4	0.420986	0.392652	-1.216721	-0.212223	0.721916	-1.077058	0.214520	0.59412

Principal Component Vectors:

	Age	G	MP	FG%	3P%	2P%	eFG%	
PC1	0.054216	0.254287	0.327467	0.202046	0.092669	0.149255	0.196105	(
PC2	0.079928	0.039406	0.186699	-0.476170	0.107665	-0.468223	-0.419316	-(
PC3	-0.148609	-0.191239	-0.062522	-0.179768	-0.517000	-0.225570	-0.358731	
PC4	0.961685	0.091085	-0.020325	-0.015803	-0.057790	-0.081523	-0.009625	
PC5	-0.139940	0.319872	0.023741	-0.167331	0.638576	-0.320552	0.009097	
PC6	0.071837	-0.637487	-0.130557	0.071389	0.409402	-0.109564	0.091507	
PC7	0.055285	-0.421681	0.000183	-0.004157	0.077089	0.024249	0.034973	-(
PC8	0.017636	0.164669	-0.004347	-0.061957	-0.121501	0.111242	-0.061191	-(
PC9	0.075833	-0.337683	0.044895	0.018706	0.075462	-0.075094	0.061851	-
PC10	0.039518	-0.164594	0.250457	-0.321214	0.047348	0.496475	-0.175836	-(
PC11	0.007001	0.048914	-0.202139	-0.266279	0.245452	0.507910	-0.303107	(
PC12	0.084166	0.052595	-0.067947	-0.066836	0.153130	0.231884	-0.227849	
PC13	0.033990	0.160718	-0.517483	0.090129	-0.001382	-0.085124	0.002525	-
PC14	-0.000478	0.054748	-0.632028	0.114569	0.055297	0.037000	-0.199933	-(
PC15	0.006058	-0.013000	0.241826	0.680265	0.135669	-0.027508	-0.651916	-
PC16	-0.000179	-0.000850	0.003079	-0.003423	0.000224	0.000839	0.002226	-(

In [7]:
```
# print(pca_results_df.head(3).to_markdown())
# print(pca_vectors.head(3).T.to_markdown())
```

Below are the first three principal components, which account for about 74% of the variation, as well as the associated principal values.

	PC1	PC2	PC3
Age	0.0542161	0.0799279	-0.148609
G	0.254287	0.0394062	-0.191239
MP	0.327467	0.186699	-0.0625222
FG%	0.202046	-0.47617	-0.179768
3P%	0.0926692	0.107665	-0.517
2P%	0.149255	-0.468223	-0.22557
eFG%	0.196105	-0.419316	-0.358731
ORB	0.232064	-0.220805	0.420215
DRB	0.32266	-0.00233425	0.219192

	PC1	PC2	PC3
TRB	0.31493	-0.0695217	0.294717
AST	0.249771	0.3139	-0.16707
STL	0.264632	0.205088	-0.0746307
BLK	0.231912	-0.166096	0.312837
TOV	0.294906	0.239421	-0.0245564
PF	0.299195	0.0239195	0.0882447
PTS	0.308528	0.210104	-0.0838843

Principal Component	Explained Variance (%)	Cumulative Variance (%)	Eigenvalue/Principal Value
PC1	48.1701	48.1701	7.72072
PC2	15.2292	63.3993	2.44094
PC3	10.922	74.3214	1.75058

How do these vectors and scalars relate to your matrix in part 2?

The orignal data set is made up of players (rows) and NBA statistics (columns). Many of these features are correlated some how, meaning that there is redundancy in the data. For example, a player who scores more points per game on average likely has a higher field goal percentage. What principal component analysis (PCA) does is it transforms the original (standardized) data into a new sort of coordinate system, where the principal components (PCs) are like the axes -- which capture the most variance in the data with as few dimensions as possible. The eigenvalues (λ_i) represent how much variance each PC accounts for or explains. A larger λ_i indicates more importance in capturing the dataset's structure/variation. The total variation is given by the sum of the eigenvalues, or:

$$\text{Total Variance} = \sum_{i=1}^{16} \lambda_i$$

Then, this means that the proportion of variation explained by each PC is given by:

$$\frac{\lambda_i}{\sum \lambda} \times 100$$

The first three principal components account for about 74% of the total variation, meaning they capture most of the meaningful structure and insights in the data.

PC1: Overall Player Activity (~48% of variation):

- PC1, which accounts for 48% of the variation, is heavily influenced by all features, except Age, 3P% 2P%, and eFG%.
- The highest loading is minutes played (MP), which makes sense, as players who play more minutes tend to have more opportunities for scoring, rebounding, etc.
- All loadings are positive, meaning features increase together, or are proportional to one another. For example:
  - Players who play more minutes tend to score more points
  - Players who get more rebounds (TRB -- total rebounds) generally do so on both, offensive and defensive ends.
  - These kinds of arguments and relationships can generally be made for most features.
- Shooting percentages do not contribute much to PC1 because they vary less among NBA players.
  - The lack of variation does not surprise me, as if you are good enough to be in the NBA, your shooting percentages have to be relatively high (around 40-50%).
  - Shooting percentages don't have large variation from one player to another, but what does is average number of points (or shots made).
  - If you think about it, two players could have the exact same 2P%, but Player 1 could have only made 4/10 shots, where as Player 2 could have made 40/100 shots.
  - Clearly, Player 2 plays and shoots a lot more than Player 1, meaning he's probably a better player, even though their 2P% is the same/similar.

PC2: Playmakers and Guards (~15% of variation):

- Field goal percentage, two point percentage, and effective field goal percentage (FG%, 2P%, and eFG%, respectively) have strong negative loadings. Therefore, this would indicate that players with a larger contribution of PC2 have lower shooting effiiency.
- Assists, steals, and turnovers (AST, STL, and TOV) have positive loadings, indicating that players with higher PC2 values can be seen as "playmakers", who handle the ball a lot.
- I think PC2 separates efficient scorers from ball handlers. What I mean by that is players with low PC2 values may excel in shooting efficiency, but also may not generate many plays (via assists or steals). On the contrary, players with high PC2 values are likely to be pointguards, as they are seen as playmakers, who assist more, but also turn the ball over more frequently, as they typically take risks that sometimes don't pay off.

PC3: Centers vs. Perimeter Shooters (~11% of variation):

- Three point percentage (3P%) has the strongest (negative) loading. Offensive rebounds, blocks, and total rebounds (ORB, BLK, and TRB) have positive loadings, meaning that players who are near the basket often (eg. centers) likely have a high PC3 value.
- From the difference in large, negative 3P% vs. large positve ORB/BLK/TRB, I think that PC3 distinguishes centers or "big men" from perimeter shooters.
    - Player with high PC3 values are strong rebounders and shot blockers, but don't shoot three-pointers often. The opposite argument can be made for low PC3 values.

Conclusion

Although the first three principal components only account for 74% of the total variation, I believe they are the only ones that provide clear meaning and insights. To capture 90% or more of the variation, we would need to consider the first seven principal components. However, I find it difficult to point out significant insights for PCs 4-7.

That being said, since the first seven principal components explain over 90% of the total variation, we can use PCA to reduce the dataset’s dimension from 16 to 7. This reduction is allowed because the remaining nine components contribute less than 10% of the dataset's variation, meaning they likely capture redundant information. By keeping only seven dimensions, we make the dataset smaller, improving computational efficiency while keeping the integrity and accuracy of the original data.