AMS 595 Project 5: Page Rank Algorithm, PCA, Linear Regression, and Gradient Descent

Jane Condon

November 14, 2025

1 Introduction

There are 4 sections to this project. First, we implement a page rank algorithm, often used by search engines, to rank websites based on the number of websites pointing to them. In the second part, we implement principal component analysis (PCA) on a dataset consisting of height and weight measurements for 100 people. In the third part, we implement linear regression via least squares to predict house prices. In the last part, we implement gradient descent to minimize a loss function.

2 Page Rank Algorithm

In this section, we implement a PageRank algorithm, which is usen by search engines, to rank websites based on the number of websites pointing to them. We have the following matrix to represent a small web network:

$$M = \begin{bmatrix} 0 & 0 & \frac{1}{2} & 0 \\ \frac{1}{3} & 0 & 0 & \frac{1}{2} \\ \frac{1}{3} & \frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{3} & \frac{1}{2} & \frac{1}{2} & 0 \end{bmatrix}$$

where each entry M[i,j] is the probability that a user on page j will click a link to page i. We can accomplish this using the following steps:

2.1 Computing the Dominant Eigenvector of M:

Finding Index of the Dominant Eigenvalue (Closest to 1)

[8]:
```
# Computing eigenvalues and eigenvectors
eigenval, eigenvec = eig(M)
```

[10]:
```
# Finding index of eigenvalue closest to 1
index = np.argmax(np.real(eigenval))
pagerank = np.real(eigenvec[:, index])
```

[12]:
```
# Normalization
pagerank = pagerank / np.sum(pagerank)
print(pagerank) # displaying the dominant eigenvector
```

```
[0.15789474 0.21052632 0.31578947 0.31578947]
```

Figure 1: Code used to compute the dominant eigenvector of M.

2.2 Constructing an initial rank vector and iterating until convergence:

We do this by iterating until the difference between consecutive rank vectors is small (in this case, less than the epsilon value of 10e-6). We normalize the resulting eigenvector so that the sum of all ranks is 1.

Creating Intial Rank Vector v and Iterating Until Convergence

[15]:
```
# Starting with initial rank vector v (of ones)
v = np.ones(4) / 4
epsilon = 1e-6
```

[17]:
```
# Iterating until convergence
while True:
    v_n = M @ v
    if np.linalg.norm(v_n - v) < epsilon: # if difference between consecutive rank vectors is small
        break
    v = v_n # setting current vector to the next consecutive vector

pagerank_it = v_n / np.sum(v_n) # normalizing
print(pagerank_it) # displaying the dominant eigenvector
```

```
[0.15789468 0.21052625 0.31578954 0.31578954]
```

Figure 2: Iterating until convergence.

The resulting eigenvector shown in the figure above represents the final PageRank scores. The values of the eigenvector represent page importance. Each value represents the long term probability of being on page i (after several random clicks). For example, a higher value indicates that a user is more likely to be on that page in the steady state. Based on the above vector representing the final PageRank scores, page 3 and page 4 are tied for the highest rank (both with a score of approximately 0.316). This may have to do with the symmetry involved in the patterns of page 3 and page 4 (each page has an 0.5 probability of linking to each other and to the same other pages).

3 Dimensionality Reduction via PCA

In this section, we are given a dataset containing the (standardized) height and weight measurements of 100 people:

$$\text{Data} = \begin{bmatrix} h_1 & w_1 \\ h_2 & w_2 \\ \vdots & \vdots \\ h_{100} & w_{100} \end{bmatrix}$$

In order to compress data while still preserving the most important patterns, we will implement principal component analysis (PCA) on the dataset. We can accomplish this using the following steps:

3.1 Computing the Covariance Matrix:

```
# Computing covariance matrix
cov_mat = np.cov(data, rowvar=False)
print(cov_mat)
```

```
[[1.02608749 0.11769063]
 [0.11769063 1.08134929]]
```

Figure 3: Code used to compute the covariance matrix of the data.

3.2 Performing Eigenvalue Decomposition on the Covariance Matrix:

```
[28]: eigenval, eigenvec = eigh(cov_mat)
```

Figure 4: Code used to perform eigenvalue decomposition.

3.3 Identifying Principal Components:

```
# Sorting eigenvalues and eigenvectors in descending order
eigenval = eigenval[::-1]
eigenvec = eigenvec[:, ::-1]
print("Eigenvalues (variance explained):", eigenval)
print("Eigenvectors (principal components):\n", eigenvec)
```

```
Eigenvalues (variance explained): [1.17460905 0.93282774]
Eigenvectors (principal components):
 [[ 0.62106317 -0.78376051]
 [ 0.78376051  0.62106317]]
```

Figure 5: Sorting the eigenvalues and eigenvectors to identify the principal components.

The first principal component is the eigenvector [0.621, 0.784]. The second principal component is [0.784,0.621]. They are orthogonal to each other. The eigenvalues represent the amount of variance explained by each principal component. After dividing each value by the total variance, we can say that the first principal component explains about 56% of the variance, while the second principal component explains about 44% of the variance.

3.4 Reducing the Dataset to 1D:

To reduce the dataset to 1D, we project it onto the principal component that captures the most variance. In this case, that would be the first principal component.

```
[35]: # Taking first principal component
      pc1 = eigenvec[:, 0]
      # Projecting data onto pc1
      z = data @ pc1
```

Figure 6: Code used to reduce the dataset to 1D.

We can also plot the original data vs. the projection:

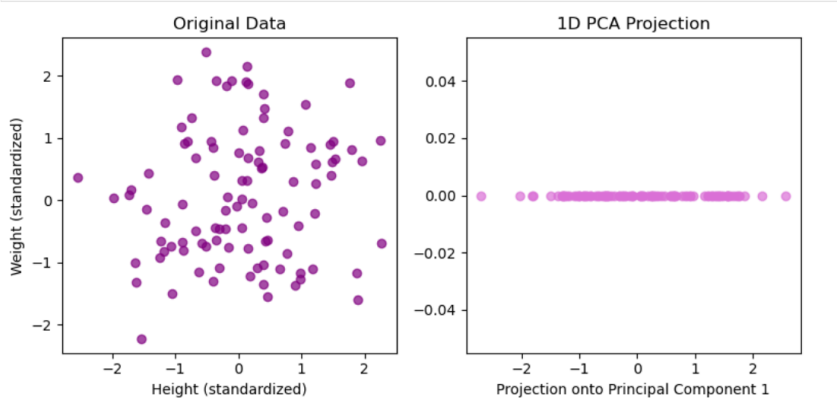

Figure 7: Plot of original data vs 1D data.

As shown in the figures above, there appears to be a positive correlation between height and weight.

4 Linear Regression via Least Squares

Linear regression can be used to predict house prices based on a variety of factors such as square footage, number of bedrooms, and age of the house. We have the following dataset:

$$X = \begin{bmatrix} 2100 & 3 & 20 \\ 2500 & 4 & 15 \\ 1800 & 2 & 30 \\ 2200 & 3 & 25 \end{bmatrix}$$

where the columns represent square footage, number of bedrooms, and age, respectively. We also have a column vector representing house price:

$$y = \begin{bmatrix} 460 \\ 540 \\ 330 \\ 400 \end{bmatrix}$$

We can predict house prices using the following steps:

4.1 Solving Least Squares Problem $X\beta = y$

First, we set up and solve the system as a least squares problem $X\beta = y$, where β represents the coefficients (weights) for square footage, bedrooms, and age.

Solving Least-Squares Problem Xβ = y

```
beta, residuals, rank, s = np.linalg.lstsq(X, y, rcond = None)
```

```
print("Coefficients (β):", beta)
```

```
Coefficients (β): [ 5.61935591e+02  4.77822046e-01 -2.23911023e+02 -2.15564409e+01]
```

Figure 8: Setting up and solving the system as a least squares problem.

As we can tell by the coefficients above, square footage has a positive impact on house price, while number of bedrooms and age has a negative impact on house price. However, the interpretation of these coefficients might be inaccurate due to scaling issues (specifically, the number of bedrooms variable) and small sample size.

4.2 Predicting House Prices using the Resulting Model

Using our least squares regression model, we can also predict the price of other houses not in the dataset. Say that we have a house that is 2400 square feet, has 3 bedrooms, and is 20 years old:

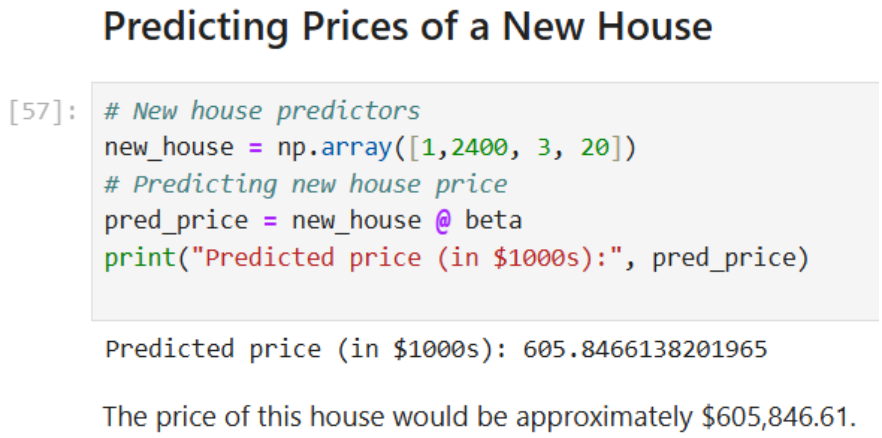

Figure 9: Predicting the price of a house with the parameters specified above.

4.3 Discussing the Role of the Least Squares Method in this Prediction Task

In general, using scipy's 'lstsq' is a safer/more flexible option than directly solving with scipy's 'solve.' lstsq provides a best linear approximation to the data, and may be used even if the predictor matrix is not square or if it is rank deficient. Using the 'solve' function requires a square, full-rank matrix, which is rarely found in real life datasets. lstsq generalizes 'solve' when we have a non-square predictor matrix ("long" or "wide" datasets) and gives the best-fit solution.

5 Gradient Descent for Minimizing Loss Function

Given a matrix $X \in \mathbb{R}^{100\times 50}$, we can implement gradient descent to minimize the loss function:

$$f(X) = \frac{1}{2}\sum_{i,j}(X_{i,j} - A_{i,j})^2$$

A is a known matrix that is the same size as X. This is effectively a mean-squared error (MSE) loss function, with the goal of making X as close as possible to a given matrix A.

5.1 Defining the Loss Function and Gradient

First, we define the loss function, shown above. We also define the gradient of the loss function as:

$$\nabla f(X) = X - A$$

[67]:
```
# Importing libraries
from scipy.optimize import minimize
```

[68]:
```
# Loss function
def loss(X_flat, A_flat):
    return 0.5 * np.sum((X_flat - A_flat)**2)
```

[71]:
```
# Gradient
def gradient(X_flat, A_flat):
    return X_flat - A_flat
```

Figure 10: Defining the loss function and its gradient.

Implementing Gradient Descent to Minimize the Loss Function

After we have defined these two functions, we implement gradient descent using scipy.optimize.minimize to minimize the loss function.

Running Gradient Descent Using Minimize Function

[74]:
```
# Flattening the initial X and A
X0 = X_initial.flatten()
A_flat = A.flatten()
```

[76]:
```
# Tracking loss
loss_list = []
```

[78]:
```
# Creating function to store loss at each iteration
def callback(X_flat):
    current_loss = loss(X_flat, A_flat)
    loss_list.append(current_loss)
```

[80]:
```
# Running optimization
res = minimize(
    fun=loss,
    x0=X0,
    jac=gradient,
    args=(A_flat,),
    method='BFGS',
    callback=callback,
    options={'gtol':1e-6, 'maxiter':1000, 'disp':True}
)
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 3
         Function evaluations: 6
         Gradient evaluations: 6
```

Figure 11: Implementing gradient descent to minimize the loss function.

The algorithm stops when the difference between the loss values in two consecutive iterations is small (in this case, smaller than our threshold of 10e-6), or after 1000 iterations. As we can see, the algorithm stops after 3 iterations. We make sure to track the loss values over each iteration to keep track.

5.2 Visualizing Loss vs. Number of Iterations

After running the gradient descent, we reshape the result back into a matrix. Then, we can plot the value of the loss function vs. the number of iterations:

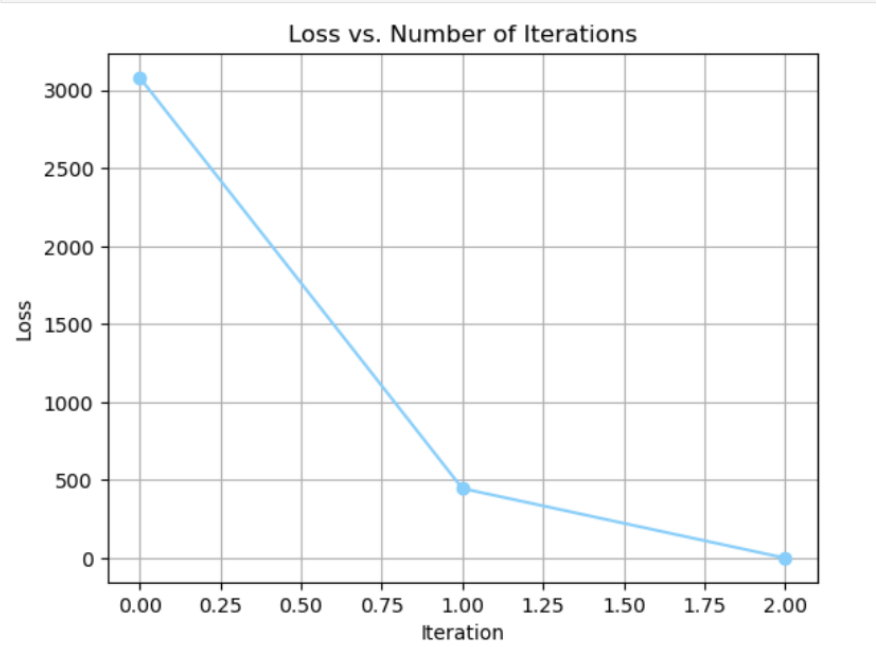

Figure 12: Plot of loss vs. number of iterations.

In the plot above, the loss function decreases monotonically until it eventually levels off near zero.