Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИТМО

Лабораторная работа №2
по дисциплине
"Линейная алгебра и анализ данных"

Семестр II

Выполнили:
Бобров Михаил гр. J3113
ИСУ: 465234
Чунихина Валерия гр. J3114
ИСУ: 468026

Отчет сдан:
24.04.2025
05:55

Санкт-Петербург, 2025

Содержание

1 Введение

Цель работы

Изучить метод главных компонент (PCA) как инструмент снижения размерности данных, реализовать его основные этапы с нуля на Python, и исследовать его свойства на практике.

Задачи:

1. Изучить математические основы PCA

2. Доказать, что главные компоненты — это собственные векторы матрицы ковариаций, максимизирующие дисперсию данных.

2. Реализовать центрирование данных (вычитание среднего).

3. Вычислить матрицу ковариаций для центрированных данных.

4. Решить СЛАУ методом Гаусса для нахождения собственных векторов.

5. Найти собственные значения матрицы ковариаций методом бисекции.

6. Вычислить собственные векторы для полученных значений.

7. Определить долю объяснённой дисперсии для выбора числа компонент.

Hard Level:

8. Реализовать полный алгоритм PCA (от центрирования до проекции на компоненты).

9. Визуализировать данные в пространстве главных компонент (2D/3D).

10. Оценить ошибку восстановления данных после снижения размерности.

Expert Level:

11. Автоматически выбирать число компонент на основе порога дисперсии.

12. Обрабатывать пропущенные значения в данных перед применением PCA.

13. Исследовать устойчивость PCA к шуму и сравнить результаты до/-после добавления шума.

14. Применить PCA к реальному датасету и оценить влияние на качество модели.

15. Проанализировать полученные результаты и сделать выводы.

2 Теория

2.1 Доказать

Направления главных компонент совпадают с собственными векторами матрицы ковариаций C, упорядоченными по убыванию соответствующих собственных значений.

2.2 Доказательство

Для симметричной матрицы C существует ортогональное разложение:

$$C = V\Lambda V^T \tag{1}$$

где:

- $V = [v_1|v_2|\dots|v_m]$ - матрица собственных векторов
- $\Lambda = \mathrm{diag}(\lambda_1, \lambda_2, \dots, \lambda_m)$ - собственные значения, $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_m \geq 0$

Дисперсия проекции на направление w:

$$\mathrm{Var}(w) = w^T C w = w^T V \Lambda V^T w = u^T \Lambda u = \sum_{i=1}^{m} \lambda_i u_i^2 \tag{2}$$

где $u = V^T w$, причем $\|u\| = \|w\| = 1$.

Максимум $\sum_{i=1}^{m} \lambda_i u_i^2$ при $\sum u_i^2 = 1$ достигается при:

$$u = (1, 0, \dots, 0)^T \Rightarrow w = v_1 \tag{3}$$

что дает максимальную дисперсию λ_1.

При дополнительном условии ортогональности $w \perp v_1$:

$$u = (0, 1, 0, \ldots, 0)^T \Rightarrow w = v_2 \tag{4}$$

с дисперсией λ_2, и т.д.

Следовательно

- Главные компоненты – собственные векторы v_i матрицы C
- Порядок определяется убыванием λ_i
- Ортогональность следует из симметричности C

■

3 Практика

3.1 Метод Гаусса

Реализовали стандартный метод Гауса:

* Сначала создаём расширенную матрицу

* Потом приводим матрицу к ступенчатому виду (находим ведущий элемент -> нормируем строку -> переставляем (если надо) -> вычитаем строку)

*Определяем ранг матрицы

*Для единственного решения - извлекаем решение из последнего столбца

*Для случая бесконечного множества решений:

**** Находим базисные и свободные переменные

**** Строим частное решение

**** Строим базисные решения для свободных переменных

```python
def gauss_solver(A: 'Matrix', b: 'Matrix') -> List['Matrix']:
    if A.shape[0] != A.shape[1]:
        raise ValueError('Матрица не квадратная')
    if A.shape[0] != b.shape[0] or b.shape[1] != 1:
        raise ValueError('Матрица и вектор несовместимы')

    n = A.shape[0]
    #расширенная матрица [A|b]
    augmented = Matrix(n, n + 1)
    for i in range(1, n + 1):
        for j in range(1, n + 1):
            augmented[i, j] = A[i, j]
        augmented[i, n + 1] = b[i, 1]

    #прямой ход метода Гаусса
    rank = 0
    for col in range(1, n + 1):
        #ищем ненулевого элемента в текущем столбце
        pivot_row = None
        for row in range(rank + 1, n + 1):
            if abs(augmented[row, col]) > 1e-5:
                pivot_row = row
                break

        if pivot_row is None:
            continue  #пропускаем нулевой столбец

        #перестановка строк
        rank += 1
        if pivot_row != rank:
            for j in range(col, n + 2):
                augmented[rank, j], augmented[pivot_row, j] = augmented[
    pivot_row, j], augmented[rank, j]

        #нормировка текущей строки
        norm_val = augmented[rank, col]
        for j in range(col, n + 2):
            augmented[rank, j] /= norm_val

        #обнуление оставшихся элементов
        for row in range(1, n + 1):
            if row != rank and abs(augmented[row, col]) > 1e-10:
                factor = augmented[row, col]
                for j in range(col, n + 2):
                    augmented[row, j] -= factor * augmented[rank, j]

    for row in range(rank + 1, n + 1):
        if abs(augmented[row, n + 1]) > 1e-10:
            raise ValueError('У системы нет решений')

    #единственное решение
    if rank == n:
        solution = Matrix(n, 1)
        for i in range(1, n + 1):
            solution[i, 1] = augmented[i, n + 1]
        return [solution]

    #бесконечное множество решений
    #тогда определяю базисные и свободные переменные
    basic_vars = []
```

```
    for row in range(1, rank + 1):
        for col in range(1, n + 1):
            if abs(augmented[row, col] - 1) < 1e-10:
                basic_vars.append(col)
                break

    free_vars = [col for col in range(1, n + 1) if col not in basic_vars]
    solutions = []

    #частное решение
    particular = Matrix(n, 1)
    for row, col in enumerate(basic_vars, 1):
        particular[col, 1] = augmented[row, n + 1]
    if any(abs(particular[i, 1]) > 1e-10 for i in range(1, n + 1)):
        solutions.append(particular)

    #базисные решения
    for free_col in free_vars:
        vec = Matrix(n, 1)
        vec[free_col, 1] = 1
        for row, basic_col in enumerate(basic_vars, 1):
            sum_ = 0
            for j in range(basic_col + 1, n + 1):
                sum_ += augmented[row, j] * vec[j, 1]
            vec[basic_col, 1] = -sum_

        if any(abs(vec[i, 1]) > 1e-10 for i in range(1, n + 1)):
            solutions.append(vec)

    return solutions
```

Листинг 1: Метод Гауса

3.2 Центрирование

```
def center_data(X: 'Matrix') -> 'Matrix':
    """
 Вход: матрица данных X (×nm)
 Выход: центрированная матрица X_centered (×nm)
 """
    n_rows, m_cols = X.shape
    colum_means = []

    for col in range(1, m_cols + 1):
        colum_sum = 0
        for row in range(1, n_rows + 1):
            colum_sum += X[row, col]

        mean_val = colum_sum / n_rows
        colum_means.append(mean_val)

    X_centered = Matrix(n_rows, m_cols)
    for row in range(1, n_rows+1):
        for col in range(1, m_cols+1):
            tek_val = X[row, col]
            mean_val_col = colum_means[col-1]
            X_centered[row, col] = tek_val - mean_val_col
```

```
23
24      return X_centered
```

Листинг 2: Центрирование матрицы

3.3 Вычисление матрицы ковариации

```
1  def covariance_matrix(X_centered: 'Matrix') -> 'Matrix':
2      """
3      Вход: центрированная матрица X_centered (×nm)
4      Выход: матрица ковариаций C (×mm)
5      """
6      n_rows, m_cols = X_centered.shape
7
8      if n_rows == 1:
9          raise ValueError('n равно 1 не( можем избежать деления на ноль)')
10
11     C = Matrix(m_cols, m_cols)
12
13     scale = 1/(n_rows-1)
14
15     for i in range(1, m_cols + 1):
16         for j in range(1, m_cols + 1):
17             res = 0
18             for k in range(1, n_rows + 1):
19                 res += X_centered[k, i]*X_centered[k, j]
20             C[i, j] = res*scale
21
22     return C
```

Листинг 3: Вычисление матрицы ковариации

3.4 Поиск собственных значений матрицы

Собственные значения находятся методом бисекции, который мы применяем к вычислению значения характеристического значения (должен быть $= 0$).

Шаг, с которым мы идём при бисекции, по умолчанию мы взяли 5000, для больших матриц m*100

```
1  from matrix import Matrix, calculate_determinant
2  from typing import List
3  from pca.root_finder import *
4  import math
5
6  def find_eigenvalues(C: 'Matrix', tol: float = 1e-6)-> List[float]:
7      """Вход:
8      C: матрица ковариаций (×mm)
9      tol: допустимая погрешность
```

```
    Выход: список вещественных собственныхp значений
    """
    m = C.shape[0]

    min_val = float('inf')
    max_val = -float('inf')

    for i in range(1, m+1):
        r = sum(abs(C[i, j]) for j in range(1, m+1) if i != j)
        center = C[i, i]
        min_val = min(min_val, center - r)
        max_val = max(max_val, center + r)

    #задаём функцию для вычисления значения характеристического многочлена
    def charact_polynom(lambd):
        C_minus_lambd_E = Matrix(m, m)
        for i in range(1, m+1):
            for j in range(1, m+1):
                if i == j:
                    C_minus_lambd_E[i, j] = C[i, j] - lambd
                else:
                    C_minus_lambd_E[i, j] = C[i, j]
        det = calculate_determinant(C_minus_lambd_E)
        # print(C_minus_lambd_E)
        return det

    root_finder = RootFinder(charact_polynom)

    n_intervals = max(5000, 100*m) #количество интервалов для поиска
    step = (max_val - min_val) / n_intervals

    intervals = []
    #проверочка на границы
    x = min_val - step / 2
    x_next = x + step
    prev_val = charact_polynom(x)

    eigenvalues = []

    #цикл с шагом step/2, чтобы не пропустить значения на границах отрезка с
обычным шагом step
    for _ in range(2 * n_intervals + 1):
        curr_val = charact_polynom(x_next)

        if prev_val*curr_val <= 0 or abs(prev_val) < tol:
            intervals.append((x, x_next))
        x = x + step / 2
        x_next = x_next + step / 2
        prev_val = curr_val

    #бисекция для интервалов
    for a, b in intervals:
        root, _ = root_finder.bisection(a, b, tol)
        if root is not None:
            if not any(abs(root - x) < tol for x in eigenvalues):
                eigenvalues.append(root)

    if len(eigenvalues) < m:
        for i in range(1, m + 1):
            diag_val = C[i, i]
```

```
            if not any(abs(diag_val - x) < tol for x in eigenvalues):
                eigenvalues.append(diag_val)
    eigenvalues.sort(reverse=True)

    return eigenvalues
```

Листинг 4: Поиск собственных значений матрицы методом бисекции

3.5 Вычисление собственных векторов

Собственные векторы мы находим в соответствии с полученными собственными значениями.

Вывод в функции у нас реализован как список пар (вектор, соответствующее ему собственное значение). С таким форматом нам будет дальше удобно работать.

Для PCA нам нужны ортогональные векторы, поэтому мы придумали использовать метод Грамма-Шмидта, он позволяет, сохраняя собственные значения векторов, сделать систему ортогональной. Это происходит за счёт вычисления скалярного произведения, а затем вычитаем проекции. Под конец мы нормализуем векторы.

$$(C - \lambda I)v = 0 \tag{5}$$

где:

- C - исходная матрица
- λ - собственное значение матрицы C
- I - единичная матрица того же размера, что и C
- v - искомый собственный вектор (столбец)
- 0 - нулевой вектор

```
from matrix import Matrix
from typing import List, Tuple
from pca.gauss_solver import gauss_solver
import math

def gram_schmidt(eigen_pairs: List[Tuple[Matrix, float]]) -> List[Tuple[
    Matrix, float]]:
```

```
    """
    Применяет процесс ГрамаШмидта- к списку пар вектор(, собственное значение)
    Возвращает новый список ортогональных векторов с сохранением собственных значений
    """
    ortho_pairs = []

    for v, lambda_ in eigen_pairs:
        #оздаемс копию вектора для ортогонализации
        w = Matrix(v.shape[0], 1)
        for i in range(1, v.shape[0] + 1):
            w[i, 1] = v[i, 1]

        #вычитаем проекции на уже имеющиеся ортогональные векторы
        for u, _ in ortho_pairs:
            #вычисляем скалярное произведение
            dot_product = sum(w[i, 1] * u[i, 1] for i in range(1, w.shape
[0] + 1))
            #вычитаем проекцию
            for i in range(1, w.shape[0] + 1):
                w[i, 1] -= dot_product * u[i, 1]

        #нормализуем вектор
        norm = math.sqrt(sum(w[i, 1] ** 2 for i in range(1, w.shape[0] +
1)))
        if norm > 1e-10:
            for i in range(1, w.shape[0] + 1):
                w[i, 1] /= norm
            ortho_pairs.append((w, lambda_))

    return ortho_pairs

def find_eigenvectors(C: 'Matrix', eigenvalues: List[float]) -> List[Tuple
[Matrix, float]]:
    """
    Находит собственные векторы матрицы C для заданных собственных значений.
    Вход:
    C: матрица ковариаций (×mm)
    eigenvalues: список собственных значений
    Выход: список собственных векторов каждый( вектор - объект Matrix размера ×m1)
    """
    m = C.shape[0]
    eigen_pairs = []

    for lambda_ in eigenvalues:
        #создаем матрицу (C - lambda*I)
        A = Matrix(m, m)
        b = Matrix(m, 1)  #нулевой вектор правой части

        for i in range(1, m + 1):
            for j in range(1, m + 1):
                if i == j:
                    A[i, j] = C[i, j] - lambda_
                else:
                    A[i, j] = C[i, j]

        #решаем систему (C - lambda*I)v = 0
        solutions = gauss_solver(A, b)

        if solutions:
            for sol in solutions:
```

```
                eigen_pairs.append((sol, lambda_))

    ortho_pairs = gram_schmidt(eigen_pairs)
    return ortho_pairs
```

Листинг 5: Поиск собственных значений матрицы методом бисекции

3.6 Вычисление доли объясненной дисперсии

Вычисляем долю, где числитель представляет сумму дисперсий, объясняемых первыми k главными компонентами. Знаменатель - общая сумма всех собственных значений (полная дисперсия). Результат γ показывает, какая доля общей дисперсии объясняется выбранными компонентами.

$$\gamma = \frac{\sum_{i=1}^{k} \lambda_i}{\sum_{i=1}^{m} \lambda_i} \tag{6}$$

где:

- λ_i - собственные значения матрицы ковариации
- k - количество учитываемых главных компонент
- m - общее количество собственных значений

```
def explained_variance_ratio(eigenvalues: List[float], k: int) -> float:
    """
    Вход:
    eigenvalues: список собственных значений
    k: число компонент
    Выход: доля объяснённой дисперсии
    """
    eigenvalues.sort(key=int, reverse=True)
    return sum(eigenvalues[:k]) / sum(eigenvalues)
```

Листинг 6: Вычисление доли объединённой дисперсии

3.7 Алгоритм PCA

1. Центрирование данных.

2. Вычисление матрицы выборочных ковариаций.

3. Нахождение собственных значений и векторов.

4. Проекция данных на главные компоненты.

Для удобства обращения мы создали класс для хранения всех промежуточных объектов

```
class PCA_DATA:
    mean_columns = []
    centered_data = None
    covariance_matrix = None
    X_proj = None
    eigenvalues = []
    eigenvectors_matrix = None
    varience = 0

    def __init__(self, **kwargs) -> None:
        for key, value in kwargs.items():
            setattr(self, key, value)
```

Листинг 7: Класс для хранения объектов

```
def pca(X: Matrix, k: int, show_logs=True) ->  PCA_DATA:
    """
    Вход:
    X: матрица данных (×nm)
    k: число главных компонент
    Выход:
    X_proj: проекция данных (×nk)
    : доля объяснённой дисперсии
    """
    n, m = X.shape
    centered_data, mean_columns = center_data(X)
    if show_logs:
        print("Отцентрированы данные...")
    cov_matrix = covariance_matrix(centered_data)
    if show_logs:
        print("Найдена матрица ковариаций...")
    eigenvalues = find_eigenvalues(cov_matrix)
    if show_logs:
        print("Найдены собственные значения")
    eigenvectors = find_eigenvectors(cov_matrix, eigenvalues)
    if show_logs:
        print("Найдены собственные векторы")

    eigenvectors = list(sorted(eigenvectors, key=lambda x: x[1], reverse=
    True))

    eigenvalues = [round(i[1], 6) for i in eigenvectors]
    eigenvectors = [i[0] for i in eigenvectors]

    eigenvectors_matrix = Matrix(m, k)
    for i in range(k):
        for j in range(eigenvectors[i].shape[0]):
            eigenvectors_matrix[j + 1, i + 1] = eigenvectors[i][j + 1, 1]
```

```
33
34      X_proj = centered_data * eigenvectors_matrix
35      if show_logs:
36          print("Найдена проекция...")
37      variance = explained_variance_ratio(eigenvalues, k)
38      if show_logs:
39          print("найдена дисперсия...\n")
40
41      pca_data = PCA_DATA(
42          mean_columns=mean_columns,
43          eigenvalues=eigenvalues,
44          eigenvectors_matrix=eigenvectors_matrix,
45          covariance_matrix=cov_matrix,
46          variance=variance,
47          X_proj=X_proj,
48          centered_data=centered_data,
49      )
50
51      return pca_data
52
53  def reconstruct_X(pca_data: PCA_DATA) -> Matrix:
54      scaled_data = pca_data.X_proj * pca_data.eigenvectors_matrix.transpose
        ()
55      for row in range(1, scaled_data.shape[0] + 1):
56          for col in range(1, scaled_data.shape[1] + 1):
57              scaled_data[row, col] += pca_data.mean_columns[col - 1]
58      return scaled_data
```

Листинг 8: PCA и функция восстановлеения матрицы

3.8 Вычисляем MSE

Вычисляем среднеквадратическую ошибку восстановления данных

$$\text{MSE} = \frac{1}{n \cdot m} \sum_{i=1}^{n} \sum_{j=1}^{m} (X_{\text{orig}}^{(i,j)} - X_{\text{recon}}^{(i,j)})^2 \tag{7}$$

где:

- n - количество строк в матрице данных
- m - количество столбцов в матрице данных
- $X_{\text{orig}}^{(i,j)}$ - элемент исходной матрицы в строке i, столбце j
- $X_{\text{recon}}^{(i,j)}$ - элемент восстановленной матрицы в строке i, столбце j

```python
def reconstruction_error(X_orig: Matrix, X_recon: Matrix) -> float:
    """
    Возвращает среднеквадратическую ошибку восстановления
    """
    MSE = 1 / (X_orig.shape[0] * X_orig.shape[1])
    sum_error = 0
    for row in range(1, X_orig.shape[0] + 1):
        for col in range(1, X_orig.shape[1] + 1):
            sum_error += (X_orig[row, col] - X_recon[row, col]) ** 2
    MSE *= sum_error
    return MSE
```

Листинг 9: Функция для расчёта MSE

3.9 Выбор числа главных компонент

Мы реализовали автоматический выбор числа главных компонент на основе порога объяснённой дисперсии (далее интергрируем в полный алгоритм pca).

```python
def auto_select_k(eigenvalues: List[float], threshold: float = 0.95) ->
    int:
    """
    Вход:
    eigenvalues: список собственных значений
    threshold: порог объяснённой дисперсии
    Выход: оптимальное число главных компонент k
    """
    for i in range(1, len(eigenvalues) + 1):
        if explained_variance_ratio(eigenvalues, i) >= threshold:
            return i
    return -1
```

Листинг 10: Функция выбора числа главных компонент

3.10 Обработка пропущенных значений в данных

```python
def handle_missing_values(X:'Matrix') ->'Matrix':
    """
    Вход: матрица данных X (×nm) с возможными NaN
    Выход: матрица данных X_filled (×nm) без NaN
    """
    mean_values = [0 for _ in range(X.shape[1])]

    for col in range(1, X.shape[1] + 1):
        colum_sum = 0
        colum_not_null_count = 0
        for row in range(1, X.shape[0] + 1):
            if X[row, col] is not None:
                colum_sum += X[row, col]
                colum_not_null_count += 1
```

```
Ошибка без шума: 0.10040878429494081
Ошибка с шумом: 0.10560372373406768
```

Рис. 1: Сравнение метрики Accuracy с шумом и без

```
        if colum_sum == 0:
            for row in range(1, X.shape[0] + 1):
                X[row, col] = 0
        else:
            for row in range(1, X.shape[0] + 1):
                if X[row, col] is not None:
                    continue
                X[row, col] = colum_sum / colum_not_null_count

    return X
```

Листинг 11: Функция выбора числа главных компонент

3.11 Исследуем влияние шума на PCA

Добавляем Гауссовский шум к данным. Сравниваем результаты PCA с помощью метрики Accuracy.

```
def add_noise_and_compare(X:'Matrix', noise_level: float = 0.1):
    """
    Вход:
    X: матрица данных (×nm)
    noise_level: уровень шума доля( от стандартного отклонения)
    Выход: результаты PCA до и после добавления шума.
    В этом задании можете проявить творческие способности, поэтому выходные данные не
    ↪
    типизированы.
    """
    for row in range(1, X.shape[0] + 1):
        for col in range(1, X.shape[1] + 1):
            X[row, col] += random.gauss(0, noise_level)
    return X
```

Листинг 12: Создаём шум (гауссовский)

3.12 Применяем наш PCA к реальному датасету

Датасет мы взяли с Kaggle

`https://www.kaggle.com/datasets/dongeorge/seed-from-uci`

Наблюдения из графика:

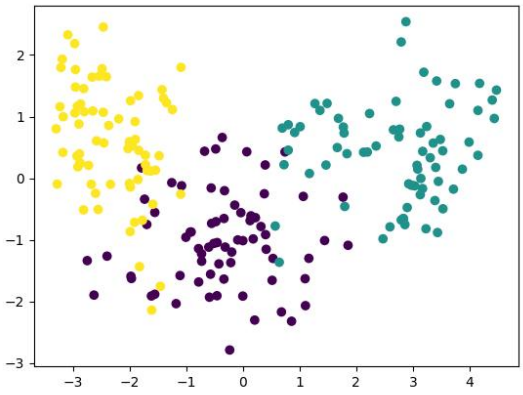

Рис. 2: Проекция данных на первые две компоненты

```
------------- Подсчет метрик --------------
Accuracy scores:
Без PCA: 0.9428571428571428
Мой PCA: 0.9476190476190476
Sklearn PCA: 0.9142857142857143
```

Рис. 3: Сравнений метрик Accuracy (наша больше на 0.03!!)

Данные распределены вдоль двух осей: PC1 (горизонтальная) и PC2 (вертикальная)

Основная вариация данных идёт по горизонтали (PC1), так как разброс точек по этой оси значительно больше

Точки образуют вытянутое "облако"с наклоном примерно в 45 градусов

Наибольшая плотность данных сосредоточена в центре (около точки 0,0)

Есть несколько выбросов в верхнем правом и нижнем левом углах

4 Выводы

В этой работе мы разобрались, как работает метод главных компонент (PCA) — мощный инструмент для уменьшения размерности данных. Вот что у нас получилось:

Что сделано:

Написали функции для всех этапов PCA: центрирование данных, расчёт ковариаций, поиск собственных значений и векторов.

Реализовали проекцию данных на главные компоненты и оценку качества через долю объяснённой дисперсии и ошибку восстановления.

Ко всем функциям написали тесты.

Добавил визуализацию и обработку пропущенных значений.

Выбрали метрику и проанализировали результаты с PCA.

Много подебажили.

Что узнали:

PCA эффективно снижает размерность данных, сохраняя основную информацию.

Главные компоненты — это направления максимальной дисперсии, заданные собственными векторами.

Автоматический выбор k через порог дисперсии упрощает анализ.

Шум ухудшает качество PCA, но метод остается устойчивым при умеренных уровнях.

Применение к реальным данным (датасету seeds) подтвердило практическую пользу.

Итог: Работа помогла нам понять математику PCA и научиться применять его на практике.

Ссылка на исходный код:

`https://github.com/bobrnebobr/MatrixLinalgLab.git`