

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ
АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ ИТМО»

ФАКУЛЬТЕТ ТЕХНОЛОГИЙ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА



Отчёт по лабораторной работе № 1
по дисциплине “Линейная алгебра и обработка данных”

Выполнили студенты:
Воробьев Андрей Павлович
гр. J3111
ИСУ 465440
Мавров Артём Николаевич
гр. J3113
ИСУ 466574
Отчет сдан: 23.04.2025

Санкт-Петербург
2025

Содержание

Введение	3
Теоретическая часть	4
.1 Математическое обоснование	4
Практическая часть	6
.1 Метод Гаусса для решения СЛАУ	6
.2 Центрирование данных	8
.3 Вычисление матрицы ковариаций	8
.4 Поиск собственных значений	9
.5 Поиск собственных векторов	11
.6 Вычисление доли объяснённой дисперсии	12
.7 Полный алгоритм PCA	12
.8 Визуализация проекций данных на первые две главные компоненты	13
.9 Вычисление среднеквадратичной ошибки восстановления данных	13
.10 Автоматический выбор числа главных компонент на основе порога объ- яснённой дисперсии	14
.11 Обработка пропущенных значений в данных	14
.12 Исследование влияния шума на PCA	15
.13 Применение PCA к реальному датасету	16
.14 Тестирование	17
Заключение	18

Введение

В данной лабораторной работе предстоит математически обосновать, что оптимальные направления PCA совпадают с собственными векторами матрицы ковариаций, реализовать полный цикл PCA без использования сторонних библиотек, дополнить функционал PCA, исследовать влияние шума и применить PCA к реальным данным.

Данная лабораторная работа была выполнена на языке Python из-за лёгкого синтаксиса, простой возможностью расширения, а также в случае необходимости ускорения работы программы отдельные сложные части когда могут быть перенесены на C++, C.

Реализация представлена в виде двух классов. Первый класс Matrix необходим для хранения и выполнения базовых операций с разреженными матрицами. Во втором классе PCA непосредственно представлен полный цикл PCA и дополнительные функции. Также были сделаны тесты.

Полный листинг кода можно найти по [ссылке](#).

Теоретическая часть

Математическое обоснование

Постановка задачи

Пусть $X \in \text{Mat}_{n \times m}(R)$ — матрица данных, центрированная по столбцам. Нас интересует вектор направления $w \in \mathbb{R}^m$, вдоль которого проекции наблюдений обладают наибольшей дисперсией.

Для отдельного наблюдения x_i проекция на направление w равна

$$z_i = x_i^\top w,$$

а вектор всех таких проекций записывается как

$$z = Xw.$$

Поскольку строки X уже центрированы, дисперсия проекций может быть выражена через евклидову норму:

$$\text{Var}(z) = \frac{1}{n} \sum_{i=1}^n z_i^2 = \frac{1}{n} \|Xw\|^2 = \frac{1}{n} w^\top X^\top X w.$$

Обозначив ковариационную матрицу как $\Sigma = \frac{1}{n} X^\top X$, получим:

$$\text{Var}(z) = w^\top \Sigma w.$$

Теперь задача сводится к максимизации квадратичной формы при условии нормированности вектора:

$$\max_{\|w\|=1} w^\top \Sigma w.$$

Свойства матрицы Σ

Пусть $X \in \mathbb{R}^n$ — случайный вектор с математическим ожиданием $\mathbb{E}[X]$. Тогда матрица ковариаций определяется как

$$\Sigma = \text{Cov}(X) = \mathbb{E}[(X - \mathbb{E}[X])(X - \mathbb{E}[X])^\top].$$

Рассмотрим квадратичную форму $x^\top \Sigma x$ для произвольного вектора $x \in \mathbb{R}^n$:

$$x^\top \Sigma x = x^\top \mathbb{E}[(X - \mathbb{E}[X])(X - \mathbb{E}[X])^\top] x.$$

Благодаря линейности математического ожидания можно записать:

$$x^\top \Sigma x = \mathbb{E}[x^\top (X - \mathbb{E}[X])(X - \mathbb{E}[X])^\top x].$$

Заметим, что

$$x^\top (X - \mathbb{E}[X])(X - \mathbb{E}[X])^\top x = ((X - \mathbb{E}[X])^\top x)^2,$$

то есть:

$$x^\top \Sigma x = \mathbb{E}[(x^\top (X - \mathbb{E}[X]))^2] \geq 0.$$

Тогда ковариационная матрица Σ симметрична и неотрицательно определённа, а значит, диагонализируема в ортонормальном базисе собственных векторов v_1, \dots, v_m с соответствующими собственными значениями $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_m \geq 0$.

Разложение вектора w по базису собственных векторов

Любой вектор w можно представить в виде линейной комбинации собственных векторов:

$$w = \sum_{i=1}^m \alpha_i v_i, \quad \text{где} \quad \sum_{i=1}^m \alpha_i^2 = 1.$$

Подставляя это в выражение $w^\top \Sigma w$, получаем:

$$w^\top \Sigma w = \left(\sum_{i=1}^m \alpha_i v_i \right)^\top \Sigma \left(\sum_{j=1}^m \alpha_j v_j \right) = \sum_{i=1}^m \sum_{j=1}^m \alpha_i \alpha_j v_i^\top \Sigma v_j.$$

Так как v_j — собственный вектор: $\Sigma v_j = \lambda_j v_j$, и $v_i^\top v_j = \delta_{ij}$, то:

$$v_i^\top \Sigma v_j = \lambda_j \delta_{ij},$$

что даёт:

$$w^\top \Sigma w = \sum_{i=1}^m \alpha_i^2 \lambda_i.$$

Оптимальный выбор w

Так как $\sum \alpha_i^2 = 1$, выражение $w^\top \Sigma w$ — это выпуклая комбинация собственных значений. Она достигает максимума, когда весь вес сосредоточен на наибольшем собственном значении:

$$\alpha_1 = 1, \quad \alpha_2, \dots, \alpha_m = 0 \quad \Rightarrow \quad w = v_1.$$

Остальные компоненты главных направлений

Аналогично, чтобы найти второе направление с максимальной дисперсией, необходимо искать максимум того же функционала на подпространстве, ортогональном v_1 . Тогда оптимальным будет v_2 , и далее по порядку.

Вывод: Оптимальные направления PCA — это собственные векторы ковариационной матрицы Σ , упорядоченные по убыванию собственных значений.

Практическая часть

Метод Гаусса для решения СЛАУ

Метод Гаусса реализован классически с оптимизацией перестановки строк. На каждом шаге i выбирается вводящая строка с максимальным элементом в позиции $[i, i]$, для уменьшения погрешности. При необходимости строки меняются местами, только вместо явной перестановки строк, меняются лишь два числа в списке `order`, за счёт чего этот процесс значительно ускоряется. При дальнейшем обращении к матрице по индексу для индекса строки необходимо сначала обращаться к `order`, если порядок важен. После выбора ведущей строки и перестановки из каждой строки ниже i вычитается ведущая строка домноженная на `factor`. После приведения матрицы к верхнетреугольному виду в зависимости от количества нулевых строк даётся либо частное либо общее решение. В случае частного решения каждая неизвестная считается сверху вниз. В случае общего сначала определяются свободные переменные и аналогично с частным для каждого вектора с свободной переменной считается искомым вектор неизвестных.

На вход матрица `A` и вектор `b` из класса `Matrix`, где `A` - матрица коэффициентов, `b` - вектор неизвестных.

```
1 def gauss_solver(A: 'Matrix', b: 'Matrix') -> List['Matrix']:
2     order = list(range(A.num_rows))
3
4     for i in range(A.num_rows - 1):
5         max_value = 0.0
6         col = i - 1
7         while round(abs(max_value), A.accuracy) == 0.0 and col !=
8             A.num_rows:
9             col += 1
10        for j in range(i, A.num_rows):
11            value = A[order[j] + 1, col + 1]
12            if abs(value) > abs(max_value):
13                max_value = value
14                pivot_row = j
15
16        if round(max_value, A.accuracy) == 0.0:
17            continue
18
19        if order[pivot_row] != i:
20            order[i], order[pivot_row] = order[pivot_row], order[
21                i]
22
23        for j in range(i + 1, A.num_rows):
24            factor = A[order[j] + 1, col + 1] / max_value
25            if round(factor, A.accuracy) == 0.0:
26                continue
27
28            for k in range(col, A.num_rows):
29                A[order[j] + 1, k + 1] -= factor * A[order[i] +
30                    1, k + 1]
31
32            b[order[j] + 1, 1] -= factor * b[order[i] + 1, 1]
33
34    A.accuracy = 5
```

```

32     amt_null_rows = 0
33     mn_null_row = A.num_rows
34     for i in range(A.num_rows):
35         flag_null_row = True
36         for j in range(i, A.num_columns):
37             if round(A[order[i] + 1, j + 1], A.accuracy) != 0.0:
38                 flag_null_row = False
39                 break
40         if flag_null_row:
41             mn_null_row = min(i, mn_null_row)
42             amt_null_rows += 1
43
44     if amt_null_rows == 0:
45         Ans = Matrix(f"{A.num_rows} 1")
46         Ans[A.num_rows, 1] = b[order[A.num_rows - 1] + 1, 1] / A[
47             order[A.num_rows - 1] + 1, A.num_columns]
48
49         for i in range(A.num_rows - 2, -1, -1):
50             value = 0
51
52             for j in range(A.num_columns, i + 1, -1):
53                 value += A[order[i] + 1, j] * Ans[j, 1]
54
55             Ans[i + 1, 1] = (b[order[i] + 1, 1] - value) / A[
56                 order[i] + 1, i + 1]
57
58         basis = []
59         basis.append(Ans)
60
61     else:
62         null_on_diag = []
63         last_ind = A.num_rows
64         for i in range(mn_null_row - 1, -1, -1):
65             for j in range(i, last_ind):
66                 if round(A[order[i] + 1, j + 1], A.accuracy) !=
67                     0.0:
68                     for k in range(j + 2, last_ind + 1):
69                         null_on_diag.append(k - 1)
70                         last_ind = j
71                         break
72
73         basis = []
74         for i in range(amt_null_rows):
75             basis.append(Matrix(f"{A.num_rows} 1"))
76
77         j = 0
78         for i in null_on_diag:
79             basis[j][i + 1, 1] = 1
80             j += 1
81
82         for k in range(amt_null_rows):
83             for i in range(null_on_diag[k] - 1, -1, -1):
84                 if i in null_on_diag:

```

```

82         continue
83
84     first_not_null = A.num_columns
85     for j in range(i, A.num_columns):
86         if round(A[order[i] + 1, j + 1]) != 0.0:
87             first_not_null = j
88             break
89
90     value = 0
91     for j in range(A.num_columns - 1, first_not_null,
92                    -1):
93         value += A[order[i] + 1, j + 1] * basis[k][j
94                + 1, 1]
95
96     if value != 0:
97         basis[k][i + 1, 1] = (-value) / A[order[i] +
98                1, first_not_null + 1]
99
100     return basis

```

Центрирование данных

Центрирование данных происходит по формуле $X_{centered} = X - mean(X)$. Реализация разбита на две функции: `mean_vector` которая вычисляет вектор средних значений для каждой колонки матрицы `X` и `center_data`, которая возвращает центрированную матрицу.

```

1 def mean_vector(X: 'Matrix') -> 'Matrix':
2     mean_vector = Matrix(f"{X.num_columns} 1")
3     for i in range(1, X.num_columns + 1):
4         mean_vector[i, 1] = sum(X[j, i] for j in range(1, X.
5                num_rows + 1)) / X.num_rows
6
7     return mean_vector
8
9 def center_data(X: 'Matrix') -> 'Matrix':
10    X_vector = PCA.mean_vector(X)
11    X_centered = Matrix(f"{X.num_rows} {X.num_columns}")
12    for i in range(1, X.num_rows + 1):
13        for j in range(1, X.num_columns + 1):
14            X_centered[i, j] = X[i, j] - mean_X_vector[j, 1]
15
16    return X_centered

```

Вычисление матрицы ковариаций

Вычисления происходят по формуле: $C = \frac{1}{n-1} X^T X$.

```

1 def covariance_matrix(X_centered: 'Matrix') -> 'Matrix':
2     X_centered_transposed = X_centered.transpose()
3     covariance_matrix = X_centered_transposed * X_centered

```



```

4         covariance_matrix /= X_centered.num_rows - 1
5
6     return covariance_matrix

```

Поиск собственных значений

Поиск собственных значений осуществляется посредством применения формулы:

$$\det(C - \lambda I) = 0$$

Алгоритм нахождения собственных значений основан на нескольких идеях. Так как по ТЗ требуется находить значения методом бисекции, необходимо сначала определять интервал, на котором будут исследоваться отрезки.

Границы интервала получаются из информации о матрице. Известно, что у матрицы ковариаций вещественные и неотрицательные корни. Из этого делается вывод, что нижняя граница — 0, а верхняя — след матрицы (это берётся из теоремы Виета для характеристического многочлена).

Из-за сложностей с нахождением собственных значений было решено дополнительно разбивать интервал с помощью уже найденных собственных значений. Для этого в начале реализована проверка: значение 1 добавляется в список, если оно является корнем характеристического многочлена. Также отдельно проверяется 0, так как это частое значение, которое тяжело заметить с помощью обычной бисекции. Это особенно актуально, так как у ковариационных матриц часто наблюдаются нулевые собственные значения из-за линейной зависимости признаков.

Большие интервалы разбиваются на маленькие отрезки в количестве `num_intervals`. Далее для каждого такого отрезка считается значение характеристического многочлена в начале и в конце. Если значения имеют противоположные знаки, значит на этом отрезке гарантированно есть корень, и запускается алгоритм бисекции. Метод постепенно сужает отрезок, пока длина интервала не станет меньше `tolerance`. После этого значение округляется и добавляется в список найденных собственных значений.

Если после прохода найдено меньше собственных значений, чем число строк матрицы (т.е. возможное число независимых признаков), количество миниинтервалов увеличивается в 10 раз. Это делается для того, чтобы попытаться слишком близкие друг к другу собственные значения, которые могли быть пропущены при грубом разбиении.

Когда достигается максимальное количество миниинтервалов, алгоритм останавливается. Это нужно так как могут быть кратные собственные значения.

```

1 def characteristic_polynomial_value(covariance_matrix: 'Matrix',
   value: float) -> float:
2     matrix = deepcopy(covariance_matrix)
3     for row in range(1, matrix.num_rows + 1):
4         matrix[row, row] -= value
5
6     return matrix.get_determinant()
7
8 def find_eigenvalues(covariance_matrix: 'Matrix',
9                     tolerance: Optional[float] = 1e-10,
10                    initial_num_intervals: Optional[int] = 10,
11                    max_num_intervals: Optional[int] = 1000) -> List
   [float]:
12     eigenvalues = []
13     matrix = deepcopy(covariance_matrix)

```

```

14
15 zero_eigenvalue = \
16     (PCA.characteristic_polynomial_value(matrix, -tolerance)
17      * PCA.characteristic_polynomial_value(matrix, tolerance) <=
18          0
19      or PCA.characteristic_polynomial_value(matrix, 0) < 1e-6)
20 one_eigenvalue = abs(PCA.characteristic_polynomial_value(matrix,
21     1)) < tolerance
22 eigenvalues.append(1)
23
24 num_intervals = intial_num_intervals
25
26 while len(eigenvalues) - 1 != matrix.num_rows - zero_eigenvalue
27     and num_intervals <= max_num_intervals:
28     for interval_index in range(len(eigenvalues) + 1):
29         if interval_index == 0:
30             lower_bound = tolerance
31             higher_bound = eigenvalues[0] - tolerance
32         elif interval_index == len(eigenvalues):
33             lower_bound = eigenvalues[-1] + tolerance
34             higher_bound = matrix.get_tracer() + tolerance
35         else:
36             lower_bound = eigenvalues[interval_index - 1] +
37                 tolerance
38             higher_bound = eigenvalues[interval_index] -
39                 tolerance
40         if higher_bound - lower_bound < tolerance:
41             continue
42
43         interval_length = (higher_bound - lower_bound) /
44             num_intervals
45         current_num_intervals = num_intervals
46         if interval_length < tolerance:
47             current_num_intervals = int((higher_bound -
48                 lower_bound) / tolerance) + 1
49             interval_length = (higher_bound - lower_bound) /
50                 num_intervals
51
52         for interval in range(current_num_intervals):
53             if len(eigenvalues) - 1 == matrix.num_rows -
54                 zero_eigenvalue:
55                 break
56             low = lower_bound + interval * interval_length
57             high = lower_bound + (interval + 1) * interval_length
58             characteristic_value_low = PCA.
59                 characteristic_polynomial_value(matrix, low)
60             characteristic_value_high = PCA.
61                 characteristic_polynomial_value(matrix, high)
62
63             if characteristic_value_low == 0:
64                 eigenvalues.append(round(low, int(math.log10(1 /
65                     tolerance))))
66                 continue

```

```

55         if characteristic_value_high == 0:
56             eigenvalues.append(round(high, int(math.log10(1 /
57                 tolerance))))
58             continue
59
60         if characteristic_value_low *
61             characteristic_value_high < 0:
62             if characteristic_value_low > 0:
63                 low, high = high, low
64                 eigenvalue = (low + high) / 2
65                 while True:
66                     characteristic_value = PCA.
67                         characteristic_polynomial_value(matrix,
68                             eigenvalue)
69                     if abs(high - low) < tolerance / 10:
70                         eigenvalues.append(round(eigenvalue, int(
71                             math.log10(1 / tolerance))))
72                         eigenvalues.sort()
73                         break
74                     if characteristic_value > 0:
75                         high = eigenvalue
76                     else:
77                         low = eigenvalue
78                         eigenvalue = (low + high) / 2
79
80         num_intervals *= 10
81
82     if zero_eigenvalue:
83         eigenvalues.append(0)
84     if not one_eigenvalue:
85         eigenvalues = [i for i in eigenvalues if i != 1]
86
87     return sorted(eigenvalues)

```

Поиск собственных векторов

Собственные вектора вычисляются по формуле: $(C - \lambda I)v = 0$, где λ - собственное число, C - матрица ковариаций, I - единичная матрица.

```

1 def find_eigenvectors(C: 'Matrix', eigenvalues: List[float]) -> Dict[
2     float, List['Matrix']]:
3     eigenvectors = {}
4     b = Matrix(f"{C.num_rows} 1")
5     I = Matrix.eye(C.num_rows)
6
7     for eigenvalue in eigenvalues:
8         A = C - (float(eigenvalue) * I)
9         eigenvectors[eigenvalue] = PCA.gauss_solver(A, b)
10
11     return eigenvectors

```

Вычисление доли объяснённой дисперсии

Доля объяснённой дисперсии считается по формуле:

$$\gamma = \frac{\sum_{i=0}^k \lambda_i}{\sum_{i=0}^m \lambda_i}$$

```
1 def explained_variance_ratio(eigenvalues: List[float], k: int) ->
    float:
2     return sum(eigenvalues[-k:]) / sum(eigenvalues)
```

Полный алгоритм PCA

Полный алгоритм PCA состоит из:

1. Центрирования данных.
2. Вычисления матрицы выборочных ковариаций.
3. Нахождения собственных значений и векторов.
4. Проекции данных на главные компоненты.

```
1 def get_components(eigenvectors: Dict[float, List['Matrix']], k: int)
    -> 'Matrix':
2     components = Matrix(f"{eigenvectors[next(iter(eigenvectors))
        ][0].num_rows} {k}")
3     count = 0
4     for eigenvalue in sorted(eigenvectors.keys(), reverse=True):
5         if count == k:
6             break
7         for vector in eigenvectors[eigenvalue]:
8             for element in range(1, vector.num_rows + 1):
9                 components[element, count + 1] = vector[element,
10                     1]
11             count += 1
12             if count == k:
13                 break
14     return components
15
16 def PCA(X: 'Matrix', k: int) -> Tuple['Matrix', 'Matrix', float]:
17     centered_X = PCA.center_data(X)
18     covariance_matrix = PCA.covariance_matrix(centered_X)
19     eigenvalues = PCA.find_eigenvalues(covariance_matrix)
20     eigenvectors = PCA.find_eigenvectors(covariance_matrix,
21         eigenvalues)
22     for eigenvalue in eigenvectors.keys():
23         eigenvectors[eigenvalue] = [vector.norm_vector() for
24             vector in eigenvectors[eigenvalue]]
25     components = PCA.get_components(eigenvectors, k)
26     projection = centered_X * components
27     variance = PCA.explained_variance_ratio(eigenvalues, k)
28
29     return projection, components, variance
```

Визуализация проекций данных на первые две главные компоненты

Для визуализации используется библиотека Matplotlib.

```
1 def plot_pca_projection(X_proj: 'Matrix') -> Figure:
2     x = [X_proj[i, 1] for i in range(1, X_proj.num_rows + 1)]
3     y = [X_proj[i, 2] for i in range(1, X_proj.num_rows + 1)]
4
5     fig, ax = plt.subplots(figsize=(8, 6))
6     ax.scatter(x, y, s=30, color='steelblue', alpha=0.7,
7               edgecolors='k', linewidths=0.5)
8     for i in range(len(x)):
9         ax.text(x[i] + 5, y[i], str(i + 1), fontsize=9, color='
10                darkred')
11     ax.set_title("Projection onto the first 2 principal
12                components", fontsize=14)
13     ax.set_xlabel("Main component 1", fontsize=12)
14     ax.set_ylabel("Main component 2", fontsize=12)
15     ax.grid(True)
16     ax.set_aspect('equal', adjustable='box')
17
18     return fig
```



Рис. 1: *
Пример визуализации

Вычисление среднеквадратичной ошибки восстановления данных

Среднеквадратичная ошибка вычисляется по формуле:

$$MSE = \frac{1}{nm} \sum_{i=0}^n \sum_{j=0}^m (X_{orig}^{ij} - X_{recon}^{ij})^2$$

Реализация разбита на две функции reconstruction восстанавливает исходную матрицу из проекции, матрицы компонентв, вектора средних значений. reconstruction_error считает среднеквадратичную ошибку по вышеприведённой формуле.

```

1 def reconstruction(projection: 'Matrix', components: 'Matrix',
2   mean_vector: 'Matrix') -> 'Matrix':
3     X_recon = projection * components.transpose()
4     for i in range(1, X_recon.num_rows + 1):
5         for j in range(1, X_recon.num_columns + 1):
6             X_recon[i, j] += mean_vector[j, 1]
7
8     return X_recon
9
10 def reconstruction_error(X_orig: 'Matrix', X_recon: 'Matrix') ->
11   float:
12     error = 0
13     for i in range(1, X_orig.num_rows + 1):
14         for j in range(1, X_orig.num_columns + 1):
15             error += (X_orig[i, j] - X_recon[i, j]) ** 2
16
17     return error / (X_orig.num_rows * X_orig.num_columns)

```

Автоматический выбор числа главных компонент на основе порога объяснённой дисперсии

Автоматический выбор числа главных компонент происходит по формуле:

$$k = \left\{ k : \frac{\sum_{i=1}^k \lambda_i}{\sum_{i=1}^m \lambda_i} \geq threshold \right\}$$

```

1 def auto_select_k(eigenvalues: List[float], threshold: float = 0.95)
2   -> int:
3     l = 0
4     r = len(eigenvalues)
5     while l < r:
6         m = (l + r) // 2
7         if PCA.explained_variance_ratio(eigenvalues, m + 1) >=
8             threshold:
9             r = m
10        else:
11            l = m + 1
12
13    return l + 1

```

Обработка пропущенных значений в данных

Пропущенные значения обрабатывались по следующему принципу:

$$X_{filled} = \begin{cases} X^{ij} & , \text{если } X^{ij} \neq NaN \\ mean(X) & , \text{иначе.} \end{cases}$$

```

1 def handle_missing_values(dataset: str) -> 'Matrix':
2     X = Matrix()
3     input_string = dataset.split('\n')
4

```

```

5     size_string = input_string[0].split()
6     X.num_rows = int(size_string[0])
7     X.num_columns = int(size_string[1])
8     X.square = X.num_rows == X.num_columns
9
10    X.values = []
11    X.row_sizes = [0]
12    X.column_indices = []
13    X.accuracy = int(input_string[1])
14
15    missing_values = []
16    columns_sums = [0] * X.num_columns
17    columns_sizes = [0] * X.num_columns
18
19    for row in range(X.num_rows):
20        row_string = input_string[row + 2].split()
21        X.row_sizes.append(X.row_sizes[-1])
22        for column in range(X.num_columns):
23            element = row_string[column]
24            if element == 'nan':
25                missing_values.append((row, column))
26            else:
27                element = round(float(element), X.accuracy)
28                if element != 0:
29                    X.values.append(element)
30                    X.column_indices.append(column)
31                    X.row_sizes[-1] += 1
32                    columns_sums[column] += element
33                    columns_sizes[column] += 1
34
35    columns_means = []
36    for sum, size in zip(columns_sums, columns_sizes):
37        if size != 0:
38            columns_means.append(sum / size)
39        else:
40            columns_means.append(0)
41    for row, column in missing_values:
42        X[row + 1, column + 1] = columns_means[column]
43
44    return X

```

Исследование влияния шума на РСА

Шум добавляется посредством генерации матрицы случайных чисел, которые зависят от среднего вектора и введённого параметра `noise_level`. Далее эта матрица прибавляется к матрице к которой мы хотим добавить шум. Применяется алгоритм РСА и вычисляется среднеквадратичная ошибка.

```

1 def handle_missing_values(dataset: str) -> 'Matrix':
2     X = Matrix()
3     input_string = dataset.split('\n')
4
5     size_string = input_string[0].split()

```

```

6      X.num_rows = int(size_string[0])
7      X.num_columns = int(size_string[1])
8      X.square = X.num_rows == X.num_columns
9
10     X.values = []
11     X.row_sizes = [0]
12     X.column_indices = []
13     X.accuracy = int(input_string[1])
14
15     missing_values = []
16     columns_sums = [0] * X.num_columns
17     columns_sizes = [0] * X.num_columns
18
19     for row in range(X.num_rows):
20         row_string = input_string[row + 2].split()
21         X.row_sizes.append(X.row_sizes[-1])
22         for column in range(X.num_columns):
23             element = row_string[column]
24             if element == 'nan':
25                 missing_values.append((row, column))
26             else:
27                 element = round(float(element), X.accuracy)
28                 if element != 0:
29                     X.values.append(element)
30                     X.column_indices.append(column)
31                     X.row_sizes[-1] += 1
32                     columns_sums[column] += element
33                     columns_sizes[column] += 1
34
35     columns_means = []
36     for sum, size in zip(columns_sums, columns_sizes):
37         if size != 0:
38             columns_means.append(sum / size)
39         else:
40             columns_means.append(0)
41     for row, column in missing_values:
42         X[row + 1, column + 1] = columns_means[column]
43
44     return X

```

При небольшом шуме ($\text{noise_level} = 0.1$) РСА сохраняет высокую точность вплоть до тысячных. Также замечена отрицательная корреляция между количеством компонент и погрешностью: чем больше количество компонент, тем выше устойчивость к шуму. Это может говорить о том, что шум как-бы расползается по осям и большее их количество компенсирует его.

Применение РСА к реальному датасету

Реализованный алгоритм РСА был применён к датасету "pokemon.csv". Время выполнения значительно возросло с большим размером матрицы.

```

1 def apply_pca_to_dataset(dataset_name: str, k: int) -> Tuple['Matrix', float]:
2     file = open(f"tests/{dataset_name}.txt", "r")

```



```
3     ds = file.read()
4     file.close()
5     X = PCA.handle_missing_values(ds)
6
7     projection, components, variance = PCA.RSA(X, k)
8     mean_vector = PCA.mean_vector(X)
9
10    reconstructed_X = PCA.reconstruction(projection, components,
11                                         mean_vector)
12    error = PCA.reconstruction_error(X, reconstructed_X)
13
14    return projection, error
```

Тестирование

Для тестирования работы PCA были сгенерированы матрицы. Эти матрицы спроецированы на компоненты и реконструированы обратно. Далее проверялось лежит ли средняя квадратичная ошибка в пределах погрешности.

Тесты можно найти в репозитории в разделе tests.

Заключение

В ходе лабораторной работы был реализован полный цикл PCA, математически обоснован выбор собственных векторов матрицы ковариаций, проведено тестирование программы, исследовано влияние шумов на работу PCA.

Из недочётов реализованного PCA, можно выделить:

- Низкая скорость работы на больших матрицах. Так как реализованный PCA использовал класс разреженных матриц многие операции такие как вызов элемента по индексу, поиск определителя и т.д. более сложные по сравнению с плотными.

Для улучшения скорости можно: использовать плотные матрицы пожертвовав памятью, переписать сложные блоки кода на C++ или C.

- Погрешность при вычислениях. Хотя и погрешность не велика, она может негативно влиять на желаемый результат.

Для улучшения погрешности можно использовать библиотеку Python Decimal, или реализовать улучшение дробных чисел самостоятельно, но это негативно скажется на алгоритмической и пространственной сложности алгоритма.

В процессе выполнения лабораторной работы были получены и закреплены знания теории линейных операторов, получен опыт работы с применением теории линейной алгебры на практике.