Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Лабораторная работа №1
по дисциплине
”Линейная алгебра и обработка данных”

Семестр II

Выполнили: студенты
Денисов Илья Дмитриевич
гр. J3111
ИСУ 465741
Воробьев Игорь Алексеевич
гр. J3110
ИСУ 465442

Отчет сдан: 24.04.2025

Санкт-Петербург
2025

Содержание

1 Начало работы

Для выполнения поставленных задач разработан класс MMatrix, который предоставляет функционал для работы с матрицами и поддерживает основные операции над ними.

```
class MMatrix:
    def __init__(self, n, m):
        self.n = n
        self.m = m
        self.V = [[0] * m for _ in range(n)]

    def __str__(self):
        return "\n".join([" ".join(map(str, row)) for row in self.V])

    def __getitem__(self, idx):
        return self.V[idx]

    def __setitem__(self, idx, value):
        self.V[idx] = value

    def swap_rows(self, i, j):
        self.V[i], self.V[j] = self.V[j], self.V[i]

    def element_add(self, i, j, value):
        self.V[i - 1][j - 1] = value

    def get(self, i, j):
        return self.V[i - 1][j - 1]

    def set(self, i, j, value):
        self.V[i - 1][j - 1] = value

    def transpose(self):
        result = MMatrix(self.m, self.n)
        for i in range(self.n):
            for j in range(self.m):
                result.set(j + 1, i + 1, self.get(i + 1, j + 1))
        return result

    def multiply(self, other):
        if self.m != other.n:
            raise ValueError("The number of columns in the first matrix
                must equal the number of rows in the second matrix.")
        result = MMatrix(self.n, other.m)
        for i in range(self.n):
            for j in range(other.m):
                total = 0
                for k in range(self.m):
                    total += self.get(i + 1, k + 1) * other.get(k + 1,
                        j + 1)
                result.set(i + 1, j + 1, total)
        return result
```

2 Задания Easy

Первым этапом работы является создание структуры данных для представления матрицы. Для этого разработан специализированный класс, включающий методы для выполнения базовых операций.

Далее выполняется предварительная обработка данных:

1. **Центрирование данных** – приведение признаков к нулевому среднему.
2. **Вычисление ковариационной матрицы** – определение степени линейной зависимости между признаками.

Затем система линейных уравнений решается методом Гаусса с выбором ведущего элемента, который состоит из двух ключевых этапов:

- **Прямой ход** – приведение матрицы к ступенчатому виду.
- **Обратный ход** – последовательное нахождение неизвестных.

Такой подход обеспечивает численную устойчивость и точность вычислений.

2.1 Реализация метода Гаусса для решения СЛАУ

Задание: Реализовать метод Гаусса для решения СЛАУ

Функция решает систему линейных уравнений методом Гаусса. Сначала проверяем, квадратная ли матрица и подходят ли размерности. Потом объединяем коэффициенты и свободные члены в расширенную матрицу. Делаем прямой ход - приводим матрицу к ступенчатому виду, выбирая ведущие элементы и преобразуя строки. Затем проверяем ранги матриц, чтобы понять, есть ли решение. Если система совместна, обратным ходом находим ответ подстановкой. На выходе получаем либо решение, либо сообщение, что решений нет или их бесконечно много, плюс информацию о ранге матрицы.

```
def gauss_solver(A: MMatrix, b: MMatrix) -> List[MMatrix]:
    n = A.n
    m = A.m
    if n != m:
        raise ValueError("Matrix A must be square (n × n) to solve the
            system.")

    augmented_matrix = [A[i][:] + [b[i][0]] for i in range(n)]

    for i in range(n):
        max_row = max(range(i, n), key=lambda r: abs(augmented_matrix[r
            ][i]))
        if abs(augmented_matrix[max_row][i]) < 1e-12:
            continue

        augmented_matrix[i], augmented_matrix[max_row] =
            augmented_matrix[max_row], augmented_matrix[i]

        pivot = augmented_matrix[i][i]
        if pivot == 0:
            continue
        for j in range(i + 1, n):
```

```
            factor = augmented_matrix[j][i] / pivot
            for k in range(i, m + 1):
                augmented_matrix[j][k] -= factor * augmented_matrix[i][
                    k]

    solutions = []
    tol = 1e-6

    rank = 0
    for i in range(n):
        if any(abs(augmented_matrix[i][j]) > tol for j in range(m)):
            rank += 1
    lead_vars = [-1] * m
    for i in range(rank):
        for j in range(m):
            if abs(augmented_matrix[i][j]) > tol:
                lead_vars[j] = i
                break

    free_vars = [j for j in range(m) if lead_vars[j] == -1]

    if not free_vars:
        solution = MMatrix(m, 1)
        for i in range(rank):
            for j in range(m):
                if abs(augmented_matrix[i][j]) > tol:
                    sum_val = augmented_matrix[i][m]
                    for k in range(j + 1, m):
                        sum_val -= augmented_matrix[i][k] * solution.
                            get(k + 1, 1)
                    val = sum_val / augmented_matrix[i][j]
                    solution.set(j + 1, 1, val)
                    break
            solutions.append(solution)
    else:
        for free in free_vars:
            vec = MMatrix(m, 1)
            vec.set(free + 1, 1, 1.0)
            for i in range(rank - 1, -1, -1):
                lead_col = -1
                for j in range(m):
                    if abs(augmented_matrix[i][j]) > tol:
                        lead_col = j
                        break
                if lead_col == -1:
                    continue
                sum_ax = 0.0
                for k in range(lead_col + 1, m):
                    sum_ax += augmented_matrix[i][k] * vec.get(k + 1,
                        1)
                val = (augmented_matrix[i][m] - sum_ax) /
                    augmented_matrix[i][lead_col]
                vec.set(lead_col + 1, 1, val)
```

```
69
70                 solutions.append(vec)
71
72         return solutions
```

2.2 Реализация функции центрирования данных

Задание: Реализовать функцию центрирования данных

В рамках задания Easy 2 реализована процедура центрирования данных. Для каждого столбца исходной матрицы вычисляется среднее значение, после чего все элементы столбца корректируются путем вычитания этого среднего. В результате получаем новую матрицу, где сумма значений по каждому столбцу строго равна нулю, что соответствует статистическому центрированию данных относительно среднего.

$$X_{\text{centered}} = X - \text{mean}(X)$$

```
    # Easy 2
def center_data(X: MMatrix) -> MMatrix:
    n = X.n
    m = X.m
    result = MMatrix(n, m)
    means = []
    for j in range(m):
        total = 0.0
        for i in range(n):
            total += X.get(i + 1, j + 1)
        means.append(total / n)
    for i in range(n):
        for j in range(m):
            result.set(i + 1, j + 1, X.get(i + 1, j + 1) - means[j])
    return result
```

2.3 Вычисление матрицы ковариаций

Задание: Вычислить матрицу ковариаций

В данном задании реализован расчет ковариационной матрицы. Алгоритм выполняет следующие шаги:

1. Берется центрированная матрица из предыдущего этапа
2. Вычисляется ее транспонированная версия
3. Ковариационная матрица находится как матричное произведение транспонированной матрицы на исходную центрированную
4. Каждый элемент результата нормируется на (n-1), где n - количество наблюдений

Это дает классическую несмещенную оценку ковариаций между признаками.

$$C = \frac{1}{n-1} X^T X$$

```
    # Easy 3
def covariance_matrix(X_centered: MMatrix) -> MMatrix:
    n = X_centered.n
    m = X_centered.m
    if n <= 1:
        raise ValueError("More than one row is required to compute the
            covariance matrix.")
    X_T = X_centered.transpose()
    C = X_T.multiply(X_centered)
    for i in range(C.n):
        for j in range(C.m):
            C.set(i + 1, j + 1, C.get(i + 1, j + 1) / (n - 1))
    return C
```

3 Задания Normal

3.1 Нахождение собственных значений матрицы C методом бисекции

Задание: Найти собственные значения матрицы C методом бисекции

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

Алгоритм вычисляет определитель через приведение матрицы к треугольному виду с помощью:

1. Частичного выбора ведущего элемента (масштабирование строк)
2. Элементарных преобразований с сохранением знака перестановок
3. Перемножения диагональных элементов с учётом знака

Метод обеспечивает численную стабильность и точность даже для плохо обусловленных матриц.

```
    # Addition to Normal 1 1
def gauss_determinant(M):
    n = M.n
    A = [[M.get(i + 1, j + 1) for j in range(n)] for i in range(n)]
    det_sign = 1
    for i in range(n):
        max_row = max(range(i, n), key=lambda r: abs(A[r][i]))
        if abs(A[max_row][i]) < 1e-12:
            return 0
        if i != max_row:
            A[i], A[max_row] = A[max_row], A[i]
            det_sign *= -1
        for j in range(i + 1, n):
            factor = A[j][i] / A[i][i]
            for k in range(i, n):
                A[j][k] -= factor * A[i][k]
    det = det_sign
    for i in range(n):
        det *= A[i][i]
    return det
```

```
    # Addition to Normal 1 1 1
def gershgorin_bounds(matrix: MMatrix) -> tuple:
    n = matrix.n
    lower = float('inf')
    upper = -float('inf')
    for i in range(n):
        radius = 0.0
        for j in range(n):
            if j != i:
                radius += abs(matrix.get(i + 1, j + 1))
        center = matrix.get(i + 1, i + 1)
        current_lower = center - radius
        current_upper = center + radius
        lower = min(lower, current_lower)
        upper = max(upper, current_upper)
    return lower, upper
```

Дополнение к Norrmal 1 1 1 (Addition to Normal 1 1 1) - границы по Гершгорину, здесь функция вычисляет границы по теореме Гершгорина для матрицы, neорема Гершгорина дает область, в которой находятся все собственные значения матрицы

```
    # Addition to Normal 1 2
def determinant_at_lambda(matrix: MMatrix, lambda_val: float) -> float:
    n = matrix.n
    C_minus_lambda = MMatrix(n, n)
    for i in range(n):
        for j in range(n):
            value = matrix.get(i + 1, j + 1) - (lambda_val if i == j
                else 0)
            C_minus_lambda.set(i + 1, j + 1, value)
    return gauss_determinant(C_minus_lambda)
```

Дополнение к Normal 1 2 (Addition to Normal 1 2) - определитель матрицы с вычитанием λ на диагонали, здесь функция вычисляет определитель матрицы $C - \lambda I$, где λ — собственное значение. это требуется для нахождения характеристического многочлена матрицы. формируем матрицу $C - \lambda I$, где на диагонали из всех элементов матрицы C вычитается значение λ, а затем вычисляется определитель полученной матрицы

```
    #  Normal 1
def find_eigenvalues(C: MMatrix, tol: float = 1e-6) -> List[float]:
    lower, upper = gershgorin_bounds(C)
    search_step = max((upper - lower) / 300, tol)
    eigenvalues = []
    current_lambda = lower
    prev_det = determinant_at_lambda(C, current_lambda)
    while current_lambda <= upper:
        current_lambda += search_step
        current_det = determinant_at_lambda(C, current_lambda)
        if prev_det * current_det < 0 or abs(current_det) < tol:
            a = current_lambda - search_step
            b = current_lambda
            while (b - a) > tol:
                mid = (a + b) / 2
                mid_det = determinant_at_lambda(C, mid)
                if mid_det * determinant_at_lambda(C, a) < 0:
                    b = mid
                else:
                    a = mid
            eigenvalues.append((a + b) / 2)
            prev_det = current_det

        decimal_places = int(-math.log10(tol))
        unique_eigs = {round(eig, decimal_places) for eig in
           eigenvalues}
    return sorted(unique_eigs, reverse=True)
```

Разберем задание Normal 1 - нахождение собственных значений. Функция находит собственные значения матрицы C методом, который применяет границы Гершгорина и метод бисекции для нахождения собственных значений, происходит расчет границ по Гершгорину, в пределах этих границ выполняется поиск с шагом, если определители матрицы при различных значениях λ изменяются по знаку или близки к нулю, то происходит нахождение корня через метод бисекции, а затем наши найденные собственные значения сортируются и возвращаются.

3.2 Нахождение собственных векторов матрицы C

Задание: Найти собственные векторы матрицы C

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

```
    # Addition to Normal 2 1
def normalize(vector: MMatrix) -> int:
    norm = 0.0
    for i in range(vector.n):
        norm += vector.get(i + 1, 1) ** 2
    return math.sqrt(norm)
```

Разберем дополнение к заданию Normal 2 1 (Affition to Normal 2 1) - нормализация вектора. Функция нормализует вектор, происходит все это путем вычисления нормы вектора и деления каждого элемента на эту норму. Рассчитывается норма вектора, затем каждый элемент вектора делится на эту норму, в конце концов функция возвращает нормированный вектор.

```python
    # Normal 2
def find_eigenvectors(C: 'MMatrix', eigenvalues: List[float]) -> List['MMatrix']:
    n = C.n
    eigenvectors = []
    unique_vectors = []

    for eigenvalue in eigenvalues:
        modified_matrix = MMatrix(n, n)
        for i in range(n):
            for j in range(n):
                value = C.get(i + 1, j + 1)
                if i == j:
                    value -= eigenvalue
                modified_matrix.set(i + 1, j + 1, value)

        result_vector = MMatrix(n, 1)
        solutions = gauss_solver(modified_matrix, result_vector)
        for vec in solutions:
            norm = 0.0
            for i in range(n):
                norm += vec.get(i + 1, 1) ** 2
            norm = math.sqrt(norm)
            if norm < 1e-10:
                continue
            vec_tuple = tuple(round(vec.get(i + 1, 1), 6) for i in range(n))
            if vec_tuple not in unique_vectors:
                unique_vectors.append(vec_tuple)
                normalized_vec = MMatrix(n, 1)
                for i in range(n):
                    normalized_vec.set(i + 1, 1, vec.get(i + 1, 1) / norm)
                eigenvectors.append(normalized_vec)

    return eigenvectors
```

Разберем задание Normal 2 - нахождение собственных векторов. функция находит собственные векторы для каждого из найденных собственных значений. сначала для каждого собственного значения строится матрица $C - \lambda I$, происходит решение системы линейных уравнений для получения собственных векторов, затем векторы нормализуются и добавляются в список уникальных собственных векторов

3.3 Вычисление доли объяснённой дисперсии

Задание: Вычислить долю объяснённой дисперсии

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

```
    # Normal 3
def explained_variance_ratio(eigenvalues: List[float], k: int) -> float
    :
    if k > len(eigenvalues) or k <= 0:
        raise ValueError("k must be within the range from 1 to the
           number of eigenvalues")
    total_sum = 0.0
    top_sum = 0.0
    for i in range(len(eigenvalues)):
        total_sum += eigenvalues[i]
        if i < k:
            top_sum += eigenvalues[i]
    if total_sum == 0:
        return 0.0
    return top_sum / total_sum
```

Разберем задание Normal 3 - доля дисперсии. Функция вычисляет долю дисперсии, объясненную первыми k собственными значениями, вычисляется сумма всех собственных значений, вычисляется сумма первых k собственных значений, возвращается отношение этих двух сумм, которое и представляет собой долю дисперсии.

4 Задания Hard

4.1 Реализация полного алгоритма PCA

Задание: Реализовать полный алгоритм PCA:

1. Центрирование данных.
2. Вычисление матрицы выборочных ковариаций.
3. Нахождение собственных значений и векторов.
4. Проекция данных на главные компоненты.

```
    # Hard 1
def pca(X: 'Matrix', k: int) -> tuple['Matrix', float]:
    X = center_data(X)
    X = covariance_matrix(X)
    eigen_values = find_eigenvalues(X)
    eigen_vectors = find_eigenvectors(X, eigen_values)
    main_comp = eigen_vectors[:k]
    Vk = MMatrix(X.n, k)
    for i in range(X.n):
        for j in range(k):
            Vk.element_add(i + 1, j + 1, main_comp[j].get(i + 1, 1))
    return X.multiply(Vk), explained_variance_ratio(eigen_values, k)
```

Разберем задание Hard 1 - метод главных компонент (PCA). Функция выполняет анализ главных компонент (PCA) для матрицы данных X и возвращает проекцию исходных данных на первые k главных компонент и долю дисперсии, объясненную этими первыми k главными компонентами. Особенностью является доля объяснений дисперсии, которая показывает, насколько эффективно главные компоненты описывают исходные данные.

4.2 Визуализация проекции данных на первые две главные компоненты

Задание: Визуализировать проекцию данных на первые две главные компоненты

```
    # Hard 2
def plot_pca_projection(X_proj: MMatrix) -> Figure:
    x = [X_proj.get(i + 1, 1) for i in range(X_proj.n)]
    y = [X_proj.get(i + 1, 2) for i in range(X_proj.n)]

    fig, ax = plt.subplots(figsize=(10, 7))

    ax.scatter(x, y, s=50, linewidth=0.8)

    for i in range(len(x)):
        ax.text(x[i] + 0.1, y[i] + 0.1, str(i + 1), color='darkred')

    ax.set_title("Projection of the data onto the first two principal
       components")
    ax.set_xlabel("PC1 (Principal Component 1)")
    ax.set_ylabel("PC2 (Principal Component 2)")
    ax.grid(True, linestyle=':', alpha=0.5)
    ax.axhline(0, color='grey', linewidth=0.5)
    ax.axvline(0, color='grey', linewidth=0.5)

    return fig
```

Разберем задание Hard 2 - визуализация данных на первые две главные компоненты. Функция создает график, который показывает данные после их проекции на первые две главные компоненты. визуализация показывает, как распределяются данные после снижения размерности. Сначала извлекаются значения первой главной компоненты для каждой строки в матрице, извлекаются значения второй главной компоненты для каждой строки, создается объект для рисования графика, отображаются точки на графике, подписываются точки на графике, и происходит последующее оформление.

4.3 Вычисление среднеквадратической ошибки восстановления данных

Задание: Вычислить среднеквадратическую ошибку восстановления данных

$$\text{MSE} = \frac{1}{m \cdot n} \sum_{ij} (X_{orig} - X_{recon})^2$$

```
    # Hard 3
def reconstruction_error(X_orig: 'Matrix', X_recon: 'Matrix') -> float:
    s = 0
    for i in range(X_recon.n):
        for j in range(X_recon.m):
            s += (X_orig.get(i + 1, j + 1) - X_recon.get(i + 1, j + 1))
                ** 2
    return s / (X_orig.n * X_orig.m)
```

Разберем задание Hard 3 - ошибка. Функция вычисляет ошибку реконструкции между исходной матрицей данных и восстановленной матрицей данных. Ошибка реконструкции измеряет, насколько хорошо восстановленная матрица сохраняет информацию из исходной матрицы. задается переменная для хранения суммы квадратов ошибок, создается цикл по всем элементам матрицы, который для каждого элемента вычисляет разницу между элементами исходной матрицы и восстановленной матрицы, а разница возводится в квадрат и добавляется к сумме ошибок. В результате выходит средняя квадратичная ошибка, которая делится на общее количество элементов в матрице