ChatGPT - Machine Learning

Ниже представлено **максимально детальное** объяснение каждой строки кода, включая, как работают функции библиотек, конструкции Python, а также математический смысл используемых формул. Структура объяснения:

1. **Общее описание**.
2. **Как работает Python** (глубокое объяснение синтаксиса, функций).
3. **Математический смысл** (если применимо).

---

1. Импорты библиотек

```python
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
```

1. `import matplotlib.pyplot as plt`:
   - Импортирует модуль `pyplot` из библиотеки `matplotlib`, который используется для построения графиков.
   - `as plt` — это псевдоним, позволяющий обращаться к модулю короче.
   - **Как работает**: Python ищет модуль `matplotlib`, загружает его в память, а затем предоставляет доступ к функциям рисования.
2. `import numpy as np`:
   - Импортирует библиотеку **NumPy**, предназначенную для работы с массивами и числовыми данными.
   - Псевдоним `np` позволяет писать `np.array`, `np.mean` и т.д.
3. `import pandas as pd`:
   - Импортирует библиотеку **Pandas**, которая используется для обработки данных в виде таблиц (DataFrame).
   - Псевдоним `pd` позволяет писать `pd.read_csv` и другие функции короче.

2. Функция `loadData`

python

```python
def loadData(filename):
    # Загрузка данных из файла
    data = pd.read_csv(filename, header=None)
    X = data.iloc[:, :-1].values  # Признаки (все столбцы, кроме последнего)
    y = data.iloc[:, -1].values  # Целевая переменная (последний столбец)
    return X, y
```

1. `def loadData(filename):`:
   - Объявление функции с аргументом `filename`.
2. `data = pd.read_csv(filename, header=None)`:
   - `pd.read_csv` — функция Pandas для чтения данных из CSV-файла.
   - `header=None` говорит, что файл не содержит заголовков, поэтому данные загружаются "как есть".
3. `X = data.iloc[:, :-1].values`:
   - `data.iloc[:, :-1]`:
     - Используется индексирование `iloc`, чтобы выбрать все строки (`:`) и все столбцы, кроме последнего (`:-1`).
     - Это делается для выбора **признаков** (features).
   - `.values`:
     - Преобразует DataFrame Pandas в массив NumPy (так как NumPy эффективнее для математических операций).
4. `y = data.iloc[:, -1].values`:
   - `data.iloc[:, -1]`:
     - Выбирает все строки (`:`) и последний столбец (`-1`).
     - Это целевая переменная (target).
5. `return X, y`:

- Возвращает два массива: `X` (признаки) и `y` (целевая переменная).

---

3. Функция `featureNormalize`

```python
def featureNormalize(X):
    # Нормализация признаков (стандартизация)
    mu = np.mean(X, axis=0)  # Среднее значение для каждого признака
    sigma = np.std(X, axis=0)  # Стандартное отклонение для каждого признака
    X_norm = (X - mu) / sigma  # Нормализация: (X - mu) / sigma
    return X_norm, mu, sigma
```

1. `def featureNormalize(X):`:
   - Определяет функцию для **стандартизации признаков**.
2. `mu = np.mean(X, axis=0)`:
   - `np.mean` вычисляет среднее значение.
   - `axis=0` говорит, что среднее считается по столбцам (для каждого признака отдельно).
3. `sigma = np.std(X, axis=0)`:
   - `np.std` вычисляет стандартное отклонение.
   - Опять же по столбцам (`axis=0`).
4. `X_norm = (X - mu) / sigma`:
   - Нормализация данных:
     $$X_{\text{norm}} = \frac{X - \mu}{\sigma}$$
     Где:
     - μ — среднее значение признака.
     - σ — стандартное отклонение признака.
   - Это приводит все признаки к **нулевому среднему** и **единичной дисперсии**.
5. `return X_norm, mu, sigma`:

- Возвращает нормализованные данные `X_norm`, а также параметры `mu` и `sigma`.

---

4. Функция `computeCostMulti`

python

```
def computeCostMulti(X, y, theta):
    # Вычисление функции стоимости для многомерной линейной регрессии
    m = len(y)  # Количество примеров
    J = (1 / (2 * m)) * np.sum(np.square(X @ theta - y))  # Формула: J(θ) = (1/2m) * Σ((Xθ - y)^2)
    return J
```

1. `def computeCostMulti(X, y, theta):`:
   - Определение функции для расчета **функции стоимости**.
2. `m = len(y)`:
   - `len(y)` возвращает количество примеров (строк в данных).
3. `X @ theta - y`:
   - Вычисляет разницу между предсказанным значением и фактическим значением:
     $$X\theta - y$$
     Здесь:
     - X — матрица признаков.
     - θ — вектор параметров.
     - y — вектор реальных значений.
4. `np.square(...)`:
   - Возводит разницу в квадрат: $(X\theta - y)^2$.
5. `np.sum(...)`:
   - Складывает все квадраты разностей.
6. `(1 / (2 * m))`:

- Масштабирует сумму по количеству примеров:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (X\theta - y)^2$$

7. `return J`:
   - Возвращает значение функции стоимости J.

> **Продолжение объяснения функций и основного кода будет ниже (пост из-за ограничения символов).**

Продолжаем подробное объяснение кода. Следующая функция: `gradientDescentMulti`.

---

5. Функция `gradientDescentMulti`

```python
def gradientDescentMulti(X, y, theta, alpha, num_iters):
    # Градиентный спуск для многомерной линейной регрессии
    m = len(y)  # Количество примеров
    J_history = np.zeros(num_iters)  # История значений функции стоимости
    for i in range(num_iters):
        # Обновление параметров θ по формуле: θ = θ - (α / m) * X.T * (Xθ - y)
        theta = theta - (alpha / m) * (X.T @ (X @ theta - y))
        # Сохранение значения функции стоимости для текущей итерации
        J_history[i] = computeCostMulti(X, y, theta)
    return theta, J_history
```

1. `def gradientDescentMulti(X, y, theta, alpha, num_iters):`:
   - Определяет функцию для **градиентного спуска**.
   - **Аргументы**:
     - `X`: матрица признаков.
     - `y`: вектор целевой переменной.

- `theta`: начальный вектор параметров (нулевой или любой другой).
- `alpha`: скорость обучения.
- `num_iters`: количество итераций.

2. `m = len(y)`:
   - Считает количество примеров m в обучающей выборке.
3. `J_history = np.zeros(num_iters)`:
   - Создает массив для хранения значений функции стоимости на каждой итерации.
   - `np.zeros(num_iters)` создает массив из `num_iters` нулей.
4. **Цикл** `for i in range(num_iters):`:
   - Повторяет процесс градиентного спуска `num_iters` раз.
5. `X @ theta - y`:
   - Вычисляет разницу между предсказанными и фактическими значениями (градиент функции стоимости).
6. `X.T @ (X @ theta - y)`:
   - Умножает транспонированную матрицу X на вектор ошибок:

     $$\nabla J(\theta) = \frac{1}{m} X^T \cdot (X\theta - y)$$

     Здесь:
     - X^T — транспонированная матрица признаков.
     - $X\theta - y$ — вектор разностей.
7. `theta = theta - (alpha / m) * ...`:
   - Обновляет параметры θ с учетом градиента:

     $$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (X\theta - y) \cdot X_j$$

     Где α — скорость обучения.
8. `J_history[i] = computeCostMulti(X, y, theta)`:
   - Сохраняет значение функции стоимости $J(\theta)$ для текущей итерации.
9. `return theta, J_history`:

- Возвращает:
  - Обновленные параметры θ.
  - Историю значений функции стоимости.

---

6. Функция `normalEqn`

```python
def normalEqn(X, y):
    # Нормальное уравнение для решения задачи линейной регрессии
    # Формула: θ = (X^T * X)^(-1) * X^T * y
    theta = np.linalg.pinv(X.T @ X) @ X.T @ y  # Pseudoinverse для вычисления θ
    return theta
```

1. `def normalEqn(X, y):`:
   - Определяет функцию для **аналитического решения линейной регрессии** с использованием нормального уравнения.
2. `X.T @ X`:
   - Умножает транспонированную матрицу X^T на матрицу X:

     $$X^T X$$

     Это дает квадратную матрицу, которая используется для аналитического решения.
3. `np.linalg.pinv(...)`:
   - Вычисляет **псевдообратную матрицу** (Moore-Penrose inverse) для $X^T X$.
   - Используется вместо **обратной матрицы** для повышения стабильности при вырожденных матрицах.
4. `X.T @ y`:
   - Умножает X^T на y:

     $$X^T y$$
5. `theta = ...`:

- Вычисляет параметры θ по формуле:

$$\theta = (X^T X)^{-1} X^T y$$

6. `return theta`:
   - Возвращает рассчитанные параметры θ.

---

7. Основной код

Загрузка и обработка данных

```python
X, y = loadData('../data/data.txt')
m = len(y)  # Количество примеров
```

- Загружает данные из файла.
- Считает количество строк в целевом массиве y.

Нормализация признаков

```python
X_norm, mu, sigma = featureNormalize(X)
```

- Нормализует признаки и сохраняет средние значения (`mu`) и стандартные отклонения (`sigma`) для дальнейшего использования.

Добавление единичного столбца

```python
```

```
X_norm = np.hstack((np.ones((m, 1)), X_norm))
```

- `np.hstack` добавляет столбец единиц к матрице признаков:

$$X = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \end{bmatrix}$$

  Это необходимо для учета **свободного члена** θ_0 в уравнении линейной регрессии.

Градиентный спуск

```python
alpha = 0.05  # Скорость обучения
num_iters = 100  # Количество итераций
theta = np.zeros(X_norm.shape[1])  # Инициализация параметров θ нулями
theta_gd, J_history = gradientDescentMulti(X_norm, y, theta, alpha, num_iters)
```

- `alpha` — скорость обучения.
- `num_iters` — количество итераций.
- `theta` — начальное значение параметров (вектор нулей).

Построение графика функции стоимости

```python
plt.figure()
plt.plot(range(1, num_iters + 1), J_history, '-b', linewidth=2)
plt.xlabel('Итерации')
plt.ylabel('Стоимость J')
plt.title('Сходимость градиентного спуска')
```

```
plt.savefig('../plots/gradient_descent_cost.png')
plt.close()
```

- Строит график функции стоимости $J(\theta)$ по итерациям.
- Сохраняет график в файл.

> Продолжение объяснения основной части кода будет в следующем сообщении (размер текста ограничен).

Продолжение объяснения основной части кода

Обучение модели с использованием нормального уравнения

```python
X_with_ones = np.hstack((np.ones((m, 1)), X))  # Добавление единичного столбца
theta_normal = normalEqn(X_with_ones, y)
print("Модель обучена с использованием нормального уравнения.")
```

1. `np.hstack((np.ones((m, 1)), X))`:
   - Добавляет единичный столбец к матрице X, чтобы включить свободный член θ_0:

$$X = \begin{bmatrix} 1 & x_1 & x_2 \\ 1 & x_1 & x_2 \\ \vdots & \vdots & \vdots \end{bmatrix}$$

   - `np.ones((m, 1))` создает столбец из единиц размером $m \times 1$.
2. `theta_normal = normalEqn(X_with_ones, y)`:
   - Вызывает функцию `normalEqn` для вычисления параметров θ с использованием аналитического метода нормального уравнения.
3. `print("Модель обучена с использованием нормального уравнения.")`:

- Сообщает пользователю, что обучение завершено.

Ввод данных пользователем

```python
print("\n--- Предсказание стоимости трактора ---")
engine_speed_input = float(input("Введите скорость двигателя: "))
num_gears_input = float(input("Введите количество передач: "))
```

1. `input`:
   - Запрашивает у пользователя ввод данных для **скорости двигателя** и **количества передач**.
2. `float()`:
   - Преобразует введенное значение из строки в число с плавающей точкой.

Нормализация пользовательских данных для градиентного спуска

```python
engine_speed_norm = (engine_speed_input - mu[0]) / sigma[0]
num_gears_norm = (num_gears_input - mu[1]) / sigma[1]
user_features_gd = np.array([[1, engine_speed_norm, num_gears_norm]])
```

1. `(engine_speed_input - mu[0]) / sigma[0]`:
   - Нормализует **введенное пользователем значение** скорости двигателя с использованием заранее рассчитанных μ (средних значений) и σ (стандартных отклонений).
2. `np.array([[1, engine_speed_norm, num_gears_norm]])`:

- Создает массив NumPy, представляющий вектор признаков с добавленным **свободным членом** θ_0 (единица в первом элементе).

Предсказание с использованием градиентного спуска

```python
predicted_price_gd = user_features_gd @ theta_gd
print(f"Предсказанная стоимость (Градиентный спуск): {predicted_price_gd[0]:.2f}")
```

1. `user_features_gd @ theta_gd`:
   - Использует **матричное умножение** между вектором признаков пользователя и вектором параметров θ, обученным градиентным спуском:

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2$$

2. `print(f"...")`:
   - Выводит предсказанную стоимость в формате с двумя знаками после запятой.

Предсказание с использованием нормального уравнения

```python
user_features_normal = np.array([[1, engine_speed_input, num_gears_input]])
predicted_price_normal = user_features_normal @ theta_normal
print(f"Предсказанная стоимость (Нормальное уравнение):
{predicted_price_normal[0]:.2f}")
```

1. `np.array([[1, engine_speed_input, num_gears_input]])`:

- Создает массив признаков пользователя **без нормализации**, поскольку нормальное уравнение работает с оригинальными данными.

2. `user_features_normal @ theta_normal`:
   - Выполняет матричное умножение для предсказания стоимости:
   $$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2$$
3. `print`:
   - Выводит результат предсказания.

Построение 3D-графика для сравнения реальных данных и предсказаний

```python
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Реальные данные
ax.scatter(X[:, 0], X[:, 1], y, color='blue', label='Реальные значения')

# Предсказание градиентным спуском
ax.scatter(engine_speed_input, num_gears_input, predicted_price_gd[0],
           color='red', s=100, label='Предсказание (Градиентный спуск)')

# Предсказание нормальным уравнением
ax.scatter(engine_speed_input, num_gears_input, predicted_price_normal[0],
           color='green', s=100, label='Предсказание (Нормальное уравнение)')

# Метки и легенда
ax.set_xlabel('Скорость двигателя')
ax.set_ylabel('Количество передач')
ax.set_zlabel('Стоимость')
ax.set_title('Сравнение реальных значений и предсказаний')
ax.legend()

# Сохранение графика
```

```
plt.savefig('../plots/models_comparison.png')
plt.close()
```

1. `fig = plt.figure()`:
   - Создает новый объект **фигуры** для построения графика.
2. `ax = fig.add_subplot(111, projection='3d')`:
   - Добавляет **трехмерную ось** (3D-проекция).
3. `ax.scatter(...)`:
   - Строит точки на графике:
     - Реальные данные (`color='blue'`).
     - Предсказание градиентным спуском (`color='red'`).
     - Предсказание нормальным уравнением (`color='green'`).
4. `ax.set_xlabel`, `set_ylabel`, `set_zlabel`:
   - Добавляет подписи к осям: **скорость двигателя**, **количество передач** и **стоимость**.
5. `plt.savefig('../plots/models_comparison.png')`:
   - Сохраняет график в файл.
6. `plt.close()`:
   - Закрывает текущую фигуру, освобождая память.

Заключение

В коде используется два подхода для линейной регрессии:

1. **Градиентный спуск** — итерационный метод с контролируемой сходимостью.
2. **Нормальное уравнение** — аналитический метод решения.

Дополнительно:

- **Нормализация признаков** ускоряет сходимость градиентного спуска.
- Визуализация результатов помогает оценить качество модели.

Этот код структурирован так, что можно вводить новые данные и предсказывать результаты, а также анализировать сходимость и точность моделей.