Метод градиентного спуска — численный метод нахождения локального минимума или максимума функции. На каждом шаге вычисляется градиент функции — вектор, указывающий направление наискорейшего роста данной величины.

Вычисление нового значения x происходит по следующей формуле:
$x^{new} = x^{cur} - \lambda * grad(f(x^{cur}))$
λ может быть выбрана 3 способами:
1. константное число (постоянный шаг)
2. выбирается на каждом шаге (в зависимости от локальной выпуклости функции?)
3. метод наискорейшего спуска

Модифицированный метод градиентного спуска используется в алгоритме обратного распространения ошибки, применяемого при обучении нейронных сетей типа перцептрон. Алгоритм ОРО изменяет весовые коэффициенты сети так, чтобы минимизировать среднюю ошибку на выходе нейронной сети при подаче на вход последовательности обучающих входных данных. (именно этот вариант был описан в презентации, данной в качестве примера)

Этапы работы над заданием.
1. Реализовать элементарный пример метода градиентного спуска для конкретной функции на языке Java в ООП парадигме.
2. Прикрутить к проекту Maven, научиться собирать им проекты.
3. Создать репозиторий и закоммитить.
4. Переписать функции программы для работы с функциями от n переменных. Коммит.
5. Освоить связку Spark-Java-Maven на элементарном примере.
6. Градиентный спуск — итеративный алгоритм, но можно распараллелить вычисление производных.
7. Как технически реализовать распараллеливание с учётом специфики Spark (RDD, функциональная парадигма). Update: никак. Красивого решения реализующего функциональную парадигму нет.

Алгоритм градиентного спуска.
eps — точность
lambda — величина шага
condition — условие, занчение которого сравнивается с eps

```
while (condition > eps) do {
     for i=1 to n do {
          compute(f'(xi));
          xi = xi - lambda*f'(xi);    //т.к. grad(f(x)) — вектор
частных производных
     }
}
```

Можно распараллелить внутренний цикл for, т.к его итерации выполняются независимо друг от друга.

Возможность и степень параллелизации зависит от 3 параметров:
1. доступное количество вычислительных единиц (узлов/ядер)
2. размерность задачи (в данном случае — кол-во аргументов функции)
3. возможность/удобство параллелизации задачи

В идеальном случае, массивы с вектором аргументов и вектором частных производных должны быть представлены в виде RDD, в этом случае можно использовать специальные операторы Spark для работы с RDD. Это даст возможность программе выполняться максимально эффективно на данной архитектуре и при заданной размерности задачи, оставляя возможность их изменения без изменения самого алгоритма распараллеливания.
Я не смогла найти подходящего решения, как реализовать на Spark вышеуказанный алгоритм (в том виде, как я его там описала). Если функция многих переменных задана аналитически, то для вычисления производной необходимо подставлять значение lambda в произвольный элемент массива, что не представляется возможным, если использовать коллекции RDD (насколько я знаю, в RDD последовательный доступ к элементам и нет итераторов). Для двух переменных вероятно ещё можно организовать такой доступ (через первый и последний элемент), но ускорение при распараллеливании двух элементов будет практически незаметным, если вообще будет.
Для реализации на Spark у меня возникло несколько других идей, одну из которых я попыталась реализовать. Наверно, это нельзя назвать алгоритмом градиентного спуска. Алгоритм ищет минимум функции двух переменных на указанном промежутке с заданным шагом. На данный момент функция задана в программе аналитически и значения высчитываются в процессе работы программы, но для алгоритма вполне подойдет функция заданная дискретно матрицей значений. Для указанного промежутка строится сетка значений, разделяется на блоки и для каждого блока параллельно ищется минимум функции. Это (теоретически) позволяет получить достаточно высокий уровень распараллеливания задачи.

Идея для программы на Spark.
Программа для функции 2х переменных.
Область поиска минимума задается: x: [a, b], y: [c, d].
Задается количество блоков, на которое будет разбита матрица (общее для каждой стороны) и шаг изменения переменных.
С помощью класса MBlock создается блочная матрица. Для параллельной обработки в RDD хранятся пары — координаты блоков. Цикл foreach (должен выполняться параллельно) для каждой пары вызывает функцию compute класса Mblock. После выполнения функции для каждого блока матрицы известен минимум и его координаты в блоке. После этого просматриваем минимумы в каждом блоке и выбираем минимальный из них, определяем абсолютные координаты x и y.
Что нужно исправить:
0. на момент отправки отчёта пограмма не компилируется.
1. Матрица со значениями некоторой дискретной функции должна читаться из входного файла + описание промежутка для расчётов (начальные значения промежутка, шаг и проверка их на соответствие размерам матрицы)
2. Второй выбор минимума тоже можно распараллелить.
3. Для входных данных должен быть алгоритм определения того, как разбить матрицу на блоки (блоки должны быть равного размера)