Prova 2 de PAOO — *Streams* e imutabilidade

Mario Leston

7 de dezembro de 2019

1 Pontos fixos

Considere uma função $f : T \to T$ que leva objetos $x : T$ em objetos $f(x) : T$. Um *ponto fixo* de f é um objeto $z : T$ tal que $f(z) = z$. A computção de um ponto fixo pode, em alguns casos, ser realizada através da construção da sequência

$$x, f(x), f(f(x)), f(f(f(x))), \ldots.$$

É claro que uma tal sequência pode ser infinita. A ideia aqui é fornecer um arcabouço que permita calcular, caso exista, um objeto $z : T$ tal que existe $k \in \mathbb{N}$ tal que $z = f^k(x)$ e $z = f(z)$. Como de hábito, não queremos uma solução tradicional baseada em mutabilidade de variávies como a seguinte:

```
static <T> T fixPoint(T x, UnaryOperator<T> f) {
    while (!x.equals(f.apply(x))) {
        x = f.apply(x);
    }
    return x;
}
```

Você deverá usar *streams* para implementar uma versão livre de mutabilidade do algoritmo de ponto fixo acima. A `API` de *streams* do `Java` possui o método *iterate* que dado um objeto $x : T$ e uma função $f : T \to T$ produz o *stream*, ou a *sequência iterada*

$$x, f(x), f(f(x)), \ldots$$

possivelmente infinito. Como é bem sabido o método *filter* pode ser usado para produzir um *substream* de um *stream*. Assim, aplicando-se *filter* ao predicado $x = f(x)$ e, a seguir, a operação *findFirst* podemos, caso exista, extrair um ponto fixo:

```
public static <T> T naiveFixPoint(T x, UnaryOperator<T> f) {
    return Stream.iterate(x, f)
```

```
                 .filter(z → z.equals(f.apply(z)))
                 .findFirst()
                 .get();
}
```

Esta aternativa satisfaz os requisitos de imutabilidade. No entanto, há uma fonte de possível ineficiência. Quando um objeto `z` é gerado pelo *stream* ele primeiro é submetido a operação *filter* que produzirá `f.apply(z)`, e se `z` e `f.apply(z)` forem distintos, então o *stream* produzirá como próximo elemento o objeto `f.apply(z)`, aplicando-se assim duas vezes a operação unária `f`. A classe que vamos esboçar no que segue e que você implementará tem como objetivo garantir que a operação unária `f` é aplicada no máximo uma vez a cada objeto gerado pelo *stream*.

Os detalhes de como isso deve ser feito em `Java` seguem esboçados no esqueleto da classe `FP`:

```java
class FP<T> {
    public final T fst;
    public final T snd;
    private final UnaryOperator<T> f;

    private FP(T fst, UnaryOperator<T> f) {
        // TO DO
    }

    public static <T> FP<T> of(T fst, UnaryOperator<T> f) {
        // TO DO
    }

    public FP<T> next() {
        // TO DO
    }

    private boolean isFixedPoint() { // TO DO }

    public static <T> T fixPoint(T x, UnaryOperator<T> f) {
        // TO DO
    }

    @Override
    public String toString() {
        return "FP{" +
                "fst=" + fst +
                ", snd=" + snd +
                '}';
```

```
    }
}
```

Lembre-se que o propósito da interface `UnaryOperator<T>` é o de representar uma função $f : T \to T$.

Os atributos `fst`, `snd`, e `f` devem satisfazer `snd == f.apply(fst)` em qualquer instância da classe `FP`. O método estático `of` é um *factory* responsável pela construção dos objetos de tipo `FP`. O método `isFixedPoint` devolve `true` se e só se `FP` representa um ponto fixo de `f`. O método `next` devolve um objeto `FP` que representa o próximo par de objetos consecutivos na sequência. Assim, se o par atual é dado por

`fp, snd`

então o próximo par deve ser

`snd, f.apply(snd).`

O método `fixPoint`, dado `x` e uma operação unária `f`, deve devolver um ponto fixo de `f`, caso exista, através da construção da sequência iterada a partir de `x` e `f`, como descrito anteriormente. Para ver se você compreendeu a ideia escreva, usando `fixPoint`, uma função

```
static <T> int find(T key, List<T> ls)
```

que devolve ou o menor índice `i` em $[0,$ `ls.size()`$)$ tal que `ls.get(i).equals(key)` ou `ls.size()` caso um tal índice não exista.

Escreva também, usando a função `fixPoint`, uma função

```
static int sum(List<Integer> xs)
```

que devolve a soma dos elementos da lista `xs`.

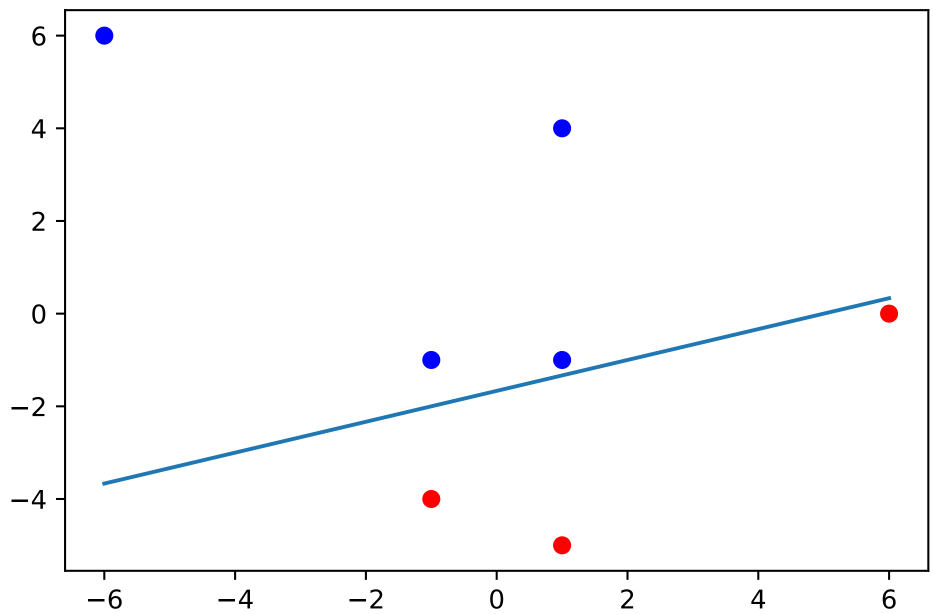
Agora, você deve escrever uma versão da função *foldLeft* usando também a função `fixPoint`:

```
static <T, U> U foldLeft(U acc, BiFunction<U, T, U> op, List<T> xs)
```

Perceptron

Durante esta seção vamos lidar com o conjunto $\mathbb{R}^d$ $d \in \mathbb{N}$. Cada habitante $\mathbf{v}$ de $\mathbb{R}^d$ é uma lista $v_1, \ldots, v_d$ de números reais. Lembre-se que o *produto interno* de vetores $\mathbf{v}, \mathbf{w} \in \mathbb{R}^d$, denotado $\langle \mathbf{v}, \mathbf{w} \rangle$, é o número $\sum_{i=1}^d v_i w_i$. A *norma* de um vetor $\mathbf{v}$, denotada $\|\mathbf{v}\|$, é o número $\sqrt{\langle \mathbf{v}, \mathbf{v} \rangle}$.

Sejam S^+ e S^- partes finitas de $\mathbb{R}^d$. Dizemos que S^+ e S^- são *separáveis* (veja a figura) se existem $\mathbf{w} \in \mathbb{R}^d$ e $b \in \mathbb{R}$ tais que



(i) $\langle \mathbf{w}, \mathbf{x} \rangle + b > 0$ para cada $\mathbf{x} \in S^+$, e

(ii) $\langle \mathbf{w}, \mathbf{x} \rangle + b < 0$ para cada $\mathbf{x} \in S^-$;

um tal par $\mathbf{w}, b$ é um *separador* de S^+ e S^-.

O problema em que estamos interessados nesta seção consiste em dados conjuntos S^+ e S^- de pontos separáveis em $\mathbb{R}^d$, encontrar um separador para S^+ e S^-. Ora, não é difícil verificar que S^+ e S^- são separáveis se, e só se, existe $\mathbf{u} \in \mathbb{R}^{d+1}$ tal que $\langle \mathbf{u}, (1 :: \mathbf{x}) \rangle > 0$ para cada $\mathbf{x} \in S^+$ e $\langle \mathbf{u}, (1 :: \mathbf{x}) \rangle < 0$ para cada $\mathbf{x} \in S^-$.

Vamos reformular, por conveniência, o enunciado do problema. Dado um conjunto $S = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\}$, onde $y_i \in \{-1, +1\}$ para $i = 1, \ldots, n$, seja

- S^+ é o conjunto dos pontos $\mathbf{x}_i$ tais que $y_i = +1$, e
- S^- é o conjunto dos pontos $\mathbf{x}_i$ tais que $y_i = -1$.

Vamos dizer que S é *separável* se S^+ e S^- são separáveis.

A ideia do algoritmo que vamos implementar está destacada abaixo:

- Suponha que $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\}$ é separável.

- Tome $\mathbf{w} \in \mathbb{R}^d$.
- Se $\mathbf{w}$ não é um separador, então existe i tal que $y_i\langle \mathbf{w}, \mathbf{x}_i\rangle \leqslant 0$.
- Neste caso, o algoritmo tenta como novo separador o ponto $\mathbf{w} + y_i\mathbf{x}_i$.
- Note que
$$y_i\langle \mathbf{w} + y_i\mathbf{x}_i, \mathbf{x}_i\rangle = y_i\langle \mathbf{w}, \mathbf{x}_i\rangle + \|\mathbf{x}_i\|^2 > y_i\langle \mathbf{w}, \mathbf{x}_i\rangle,$$
pois $\|\mathbf{x}_i\|^2 > 0$.
- Portanto, num certo sentido $\mathbf{w} + y_i\mathbf{x}_i$ é "melhor" que $\mathbf{w}$.

Eis um esboço da classe cujos detalhes vocês deverão escrever:

```java
public class Perceptron {
    private final List<List<Double>> xss;
    private final List<Double> ys;

    private Perceptron(List<List<Double>> xss, List<Double> ys) {
        this.xss = xss;
        this.ys = ys;
    }

    private Optional<Integer> wrong(List<Double> ws) {
        // TO DO
    }

    private List<Double> step(List<Double> ws) {
        // TO DO
    }

    private List<Double> getSeparator() {
        // TO DO
    }

    public static Function<List<Double>, Double>
    predictor(List<List<Double>> xss, List<Double> ys) {
        // TO DO
    }
}
```

É importante salientar mais uma vez que sua implementação deve usar *streams* e deve ser livre de mutabilidade. Você deverá também usar a classe `FP` da seção anterior. Ademais, vamos nos referir a certas funções que já vimos em aula, tais como `inner`, `scalarProd`, etc.

A variável de instância `xss` corresponde ao conjunto dos pontos separáveis e `ys` corresponde aos rótulos de tais pontos. Assim, se `ts.get(i) == 1` então `xss.get(i)` é um ponto de S^+, caso contrário, `ts.get(i) == -1`, donde `xss.get(i)` é um ponto de S^-.

O método `wrong` deve dado um candidato a separador `ws` deve devolver ou um índice `i` em $[0,$ `xss.size()`$)$ tal que `ys.get(i) * inner(ws, xss.get(i))` $\leqslant 0$ ou `OptionaInt.empty()` caso um tal `i` não exista.

O método `step` dado um candidato a separador `ws` deve devolver o próprio `ws` caso `wrong(ws)` devolva `OptionalInt.empty()`, ou um novo candidado a separador, como ilustrado acima, em caso contrário.

O método `getSeparator()` deve devolver um separador. Use o método `fixPoint` aplicado a um candidato a separador e ao operador unário `step`.

Finalmente, escreva o método `predictor` que devolve uma função que recebe um ponto e devolve a sua respectiva classificação em relação ao separador. Imite o estilo que fizemos em aula quando implementamos o *kmeans*.