CML - Uma Linguagem para *Machine Learning*

Relatório 1 versão final - Trabalho de Conceitos de Linguagem de Programação

Caio Lopes, Leonardo Blanger, Marcelo Silvarolla

11 de abril de 2018

1 Prefácio

Este é o relatório das fases 1 e 2 modificado de acordo com as alterações da linguagem feitas nas fases 3, 4 e 5. O que foi acrescentado está em azul. O que foi removido está em vermelho.

2 Requisitos do Domínio

A linguagem a ser implementada tem como objetivo facilitar a utilização de técnicas tradicionais de Aprendizado de Máquina. Este domínio de aplicação vem recebendo crescente visibilidade nos últimos anos devido a diversos fatores. Podemos citar como principais motivos o aumento na produção de dados digitais, devido ao uso cada vez maior de soluções computacionais para diversas tarefas, tanto pessoais, como científicas e da indústria, além do surgimento e aprimoramento de algoritmos capazes de extrair informações úteis a partir destas grandes massas de dados, e o aumento da capacidade computacional, que permite a utilização destes algoritmos em uma escala prática.

Atualmente, projetos de software que realizem Aprendizado de Máquina geralmente utilizam linguagens de propósito geral, como *Python*, acrescidas

de bibliotecas que agilizem tarefas específicas relacionadas a manipulação e processamento de dados.

Nesta perspectiva, este trabalho tem como objetivo o projeto e implementação de uma linguagem com funcionalidades próprias para as tarefas mais fundamentais de Aprendizado de Máquina. Esta linguagem deve permitir a manipulação de dados e fornecer comandos diretos para o treinamento e uso de algoritmos tradicionais de aprendizado.

De forma mais específica, a linguagem deverá prover comandos para leitura, gravação e manipulação de dados, de forma a criar uma interface amigável para o uso de *datasets* armazenados em planilhas no formato `csv`. A linguagem deve permitir também um tratamento individual aos dados destes *datasets*, como por exemplo: separação de exemplos dentre conjunto de treinamento e de testes, separação dos atributos entre *features* e *labels*, conversão entre atributos categóricos e numéricos e normalização de valores.[1]

Além da manipulação de *datasets*, a linguagem deve prover também funções predefinidas associadas a implementações dos algoritmos mais tradicionais da área de Aprendizado de Máquina, tais como *Perceptron*, *Pocket-Perceptron*, Regressão Linear, Regressão Logística e Árvores de Decisão. Com estas funcionalidades, a linguagem busca tornar mais ágil e amigável o treinamento de modelos preditivos utilizando *datasets* do usuário.

Além do treinamento, a linguagem deve permitir o armazenamento e carregamento destes modelos previamente treinados, através de arquivos contendo informações sobre a estrutura dos modelos e os parâmetros resultantes do treinamento, para realização de inferência em dados inéditos de forma eficiente.

2.1 Justificativas

Dentre as motivações para o projeto desta linguagem, podemos citar a possibilidade de treinamento e utilização de algoritmos de aprendizado sem a necessidade de grande conhecimento técnico, devido ao encapsulamento do funcionamento interno destes algoritmos através das funções predefinidas da linguagem. Além disso, podemos citar também a busca pela facilidade e maior rapidez de implementação, além do risco menor de *bugs*.

Com estas vantagens, seria possível a realização mais ágil de testes rápidos

[1] As conversões e normalizações serão feitas de modo implícito pelos próprios algoritmos de aprendizado.

e esboços de sistemas de predição baseados nestes algoritmos, antes de uma implementação definitiva em linguagens e bibliotecas mais eficientes.

2.2 Exemplo

Abaixo é mostrado um esboço da utilização da linguagem, contendo os tipos `dataset` e `model`, que representam o conjunto de dados e o modelo preditivo a ser treinado sobre estes dados, respectivamente. Também são mostradas funções predefinidas para leitura e gravação de *datasets*, manipulação dos atributos destes *datasets*, treinamento de um modelo preditivo do tipo *Regressão Logística* e sua utilização para inferência em outros dados de entrada, além de leitura e gravação dos parâmetros deste modelo. Este exemplo corresponde ao treinamento de um modelo para predição de probabilidade de fraude em transações de cartões de crédito, utilizando um *dataset* público de transações[2].

```
dataset D = read_data(‘‘creditcard.csv’’);
D = categorical_to_binary(D, ‘‘Class’’, {‘‘0’’, ‘‘1’’})
dataset X = remove_columns(D, ‘‘Class’’);
dataset y = columns(D, ‘‘Class’’);
model M = logistic_regression(X, y, 10000);
...
dataset predictions = predict(M, x_test);
save_data(predictions, ‘‘predictions.csv’’);
...
save_model(M, ‘‘model_name’’);
M = read_model(‘‘model_name’’);
```

A sintaxe final da linguagem possui algumas divergências em relação à que foi apresentada no exemplo anterior. A função `read_data` se transformou em `load_data`, e recebe como parâmetros o nome do aquivo e o caractere separador do `csv`. Além disso, a função `categorical_to_binary` não existe na linguagem, pois a tarefa de converter atributos categóricos para numéricos e binários é realizada internamente no treinamento dos modelos e predição de dados.

Os dados do dataset estão armazenados na planilha `creditcard.csv`, que é lida e armazenada no *dataset* `D`. Todos os atributos são numéricos, exceto

[2] *Dataset* disponível em `https://www.kaggle.com/mlg-ulb/creditcardfraud`

a classe das transações, que assumem as *strings* ‘‘0’’ ou ‘‘1’’ quando a respectiva transação for legítima ou fraudulenta, respectivamente. A segunda linha converte os elementos do conjunto {‘‘0’’, ‘‘1’’} no atributo `Class`, para valores numéricos 0 ou 1. A terceira e quarta linhas separaram as *features* das *labels*, enquanto a quinta linha treina um modelo de regressão logística utilizando estes dados por 10000 iterações. Em seguida, é utilizado este modelo treinado para realizar predições em um dataset de dados inéditos `x_test`, que são salvas na planilha `predictions.csv`. Por fim, o modelo (sua estrutura e parâmetros) é salvo e lido do arquivo `model_name`.

2.3 Funções predefinidas

A linguagem fornecerá as seguintes funções predefinidas para o programador:

```
// lê arquivo csv especificado por path e devolve um dataset, com
as entradas no arquivo separadas pelo caractere separator
dataset load_data(string path, char separator);

// guarda o dataset no caminho especificado por path, com as entradas
no arquivo separadas pelo caractere separator
void save_data(dataset d, string path, char separator);

// devolve o mesmo dataset mas com apenas as colunas especificadas
dataset cols(dataset d, string[] attributes);

// devolve o mesmo dataset sem as colunas especificadas
dataset remove_cols(dataset d, string[] attributes);

// devolve o mesmo dataset com exemplos de índice em [begin,begin+num_samples)
dataset rows(dataset d, int begin, int end);

// devolve o número de linhas (exemplos) do dataset
int num_rows(dataset d);

// estas funções devolvem modelos baseados nos datasets dados
model perceptron(dataset X, dataset y, int max_iter);

model pocket_perceptron(dataset X, dataset y, int max_iter);
```

```
model linear_regression(dataset X, dataset y, real learning_rate,
int batch_size, int num_epochs, int max_iter);

model logistic_regression(dataset X, dataset y, string feature_of_interest,
real learning_rate, int batch_size, int num_epochs, int max_iter);

// calcula os as saídas y correspondentes a X e devolve y
dataset predict(dataset X, model M);

// carrrega modelo armazenado no arquivo especificado pelo caminho
path
model load_model(string path);

// armazena modelo M no caminho path
void save_model(model M, string path);
```

3 Levantamento dos Conceitos

3.1 Tipos

Os tipos primitivos da linguagem consistem em inteiro (`int`), valor real de ponto flutuante (`real`), caractere (`char`), booleano (`bool`), além do tipo que representa um modelo preditivo treinado (`model`). O tipo `model` está enquadrado como primitivo devido ao fato de não haver o interesse atual em permitir acesso aos parâmetros individuais do modelo no nível da linguagem. A palavra reservada `void` poderá ser usada apenas como tipo de retorno de funções, indicando que a função não retorna valor.

Além dos tipos primitivos, existem três tipos compostos na linguagem. Os vetores, que são sequências de qualquer outro tipo, indexadas a partir de zero, o tipo `string`, que representa sequências de caracteres, e o tipo `dataset`, que representa um conjunto de dados. valores do tipo `dataset` são constituídas de uma sequência de exemplos, sendo que os exemplos, por sua vez, são subdivididos em atributos. Será possível acessar intervalos de exemplos dentro do *dataset*, e haverá funções predefinidas na linguagem para acessar atributos específicos de todos os exemplos.

Não será possível para o programador criar novos tipos na linguagem.

3.2 Expressões e Comandos

A linguagem possuirá as expressões aritméticas e lógicas fundamentais, como qualquer linguagem de propósito geral. Também terá expressões da forma `d[2:10]` para especificar `datasets` menores, no caso apenas com os exemplos de índice entre 2 e 10.

Operações do tipo `d[2:10]`, como no exemplo acima, não existem na versão final da linguagem. Sub-datasets podem ser obtidos através da função predefinida `rows`.

Além disto, a linguagem possuirá comandos para atribuição, desvio condicional (`if/else`) e laço de repetição (`while`).

Em detalhes, teremos as expressões e comandos a seguir, onde, quando dizemos que algo tem um certo tipo, estamos dizendo que esse algo é uma expressão cujo valor calculado tem o tal tipo.

Expressões:

- Literais
  - 5 tem tipo `int`
  - 5.0 tem tipo `real`
  - 'a' tem tipo `char`
  - "Caio" tem tipo `string`
  - true tem tipo `bool`
- Variáveis
  - x tem seu tipo determinado pela declaração visível segundo o escopo, conforme definido na seção seguinte
- Agregações
  - $\{e_1, e_2, \ldots, e_n\}$ tem tipo vetorDe(t)" se cada e_i tem tipo t. Por exemplo, {"salario", "idade", "credito"} tem tipo "vetor de `string`'s", pois "salario", "idade" e "credito" têm tipo `string`. Por outro lado, {12, 5} têm tipo "vetor de `int`'s". Assim, se o programador declarar `int[] v = {12, 5};`, o teste `v[1] == 5` resultará no valor `true`.
- Aplicações de funções

- `f`$(e_1, e_2, \ldots, e_n)$ tem tipo `t` se `f` é uma função de $s_1 \times s_2 \times \ldots \times s_n$ para t e cada e_i tem tipo s_i. Note que `f` precisa estar definida no programa para que a aplicação seja válida.
- Analogamente para as funcões predefinidas.

- Operações aritméticas
  - e_1 `+` e_2 tem tipo `int` se cada e_i tem tipo `int` e o "`+`" está na raiz da *parse tree* da expressão
  - e_1 `+` e_2 tem tipo `real` se cada e_i tem tipo `int` ou `real`, pelo menos um dos e_i tem tipo `real` e o "`+`" está na raiz da *parse tree* da expressão
  - Analogamente para `-`, `*` e `/`

- Operações lógicas
  - e_1 `<` e_2 tem tipo `bool` se cada e_i tem tipo `int` ou `float` e o "`<`" está na raiz da *parse tree* da expressão
  - Analogamente para `>`, `<=`, `>=`, `==`, `!=`
  - e_1 `&&` e_2 tem tipo `bool` se cada e_i tem tipo `bool` e o "`&&`" está na raiz da *parse tree* da expressão
  - Analogamente para `||`
  - `!`e_1 tem tipo `bool` se e_1 tem tipo `bool`

- Operações com vetores
  - `v[`e_1`]` tem tipo t se v tem tipo "vetor de t's" e e_1 tem tipo `int`

- Operações com *datasets*
  - `d[`e_1`:`e_2`]` tem tipo `dataset` se d tem tipo `dataset` e os e_i têm tipo `int`

- Atribuições
  - `x =` e_1 tem tipo t se `x` é uma variável de tipo t, e_1 tem tipo t e o "`=`" está na raiz da *parse tree* da expressão

  - `v[`e_1`] =` e_2 tem tipo t se e_1 tem tipo `int`, e_2 tem tipo t, e `v` tem tipo "vetor de `t`'s"

Comandos:

- Comando-expressão
  - e_1`;` é comando se e_1 é expressão
  - `;` é comando
- Desvio condicional
  - `if(`e_1`)` c_1 é comando se e_1 tem tipo `bool` e c_1 é comando composto
  - `if(`e_1`)` c_1 `else` c_2 é comando se e_1 tem tipo `bool` e c_1 e c_2 são comandos compostos
- Laço
  - `while(`e_1`)` c_1 é comando se e_1 tem tipo `bool` e c_1 é comando composto
- Comando composto
  - $\{c_1\ \ c_2\ \ \ldots\ \ c_n\}$ é comando se os c_i forem comandos ou declarações (onde $n \geq 1$)

3.3 Vinculação e Escopo

A vinculação das variáveis será feita de modo explícito. Toda variável deve ser declarada no código precedida do um identificador do tipo, como por exemplo:

```
dataset D = read_data('data.csv');
```

Quanto ao tempo de vinculação, as variáveis serão vinculadas de forma estática. Ou seja, antes do tempo de execução do programa, permancendo inalteradas durante a execução.

Como não é permitida a criação de novos tipos na linguagem, a vinculação de tipos a suas palavras reservadas já está embutida no processador da linguagem, e portanto, também ocorre de forma estática.

O escopo será estático: uma variável estará visível num bloco se ela for local àquele bloco ou a algum bloco que o contenha.[3] A exceção ocorre quando duas variáveis de mesmo nome estão no bloco ou em algum bloco que o contém: neste caso, a mais interior será visível e a mais exterior, não.

3.4 Sistema de Tipos

O sistema de tipos será monomórfico: variáveis, constantes e parâmetros de funções serão sempre declaradas com um único tipo. Haverá sobrecarga dos operadores de soma, subtração, multiplicação e divisão, que poderão operar sobre `int`s e `real`s. A sobrecarga será independente de contexto: olhando-se apenas para os operandos, se deduzirá o operador a ser utilizado.

3.5 Verificação de Tipos

Na verificação de tipos, será feita a coerção de `int` para `real`, tanto em expressões aritméticas como em declarações da forma `real x = 1;`. Como o programador não definirá seus próprios tipos, a equivalência de tipos se reduzirá à equivalência nominal: dois tipos são equivalentes se, e somente se, são iguais. Finalmente, sendo os tipos explicitados no código, não será feita qualquer inferência de tipos.

3.6 Avaliação de Expressões

Parênteses definirão parcialmente a ordem de avaliação de expressões. Caso ainda haja ambiguidade, a precedência de operadores a seguir será utilizada:

1. `!`
2. `*, /`
3. `+, -`
4. `<, <=, >, >=`
5. `==, !=`

[3]Note que a contenção é uma relação transitiva: se o bloco *b1* contém o bloco *b2*, e o bloco *b2* contém o bloco *b3*, então *b1* contém *b3*.

6. `&&`
7. `||`
8. `=`

   No caso de operadores com empate de precedência, a execução das operações será realizada da esquerda para a direita, com exceção da atribuição ('=') e da negação unária ('!'), que terão associatividade à direita.

3.7 Abstrações

A linguagem não terá abstrações de tipo, já que o programador não poderá definir seus próprios tipos.

Haverá abstração de funções e procedimentos sob o rótulo "função", assim como em *C*. Isto é, procedimentos serão declarados como funções que retornam `void`.

A passagem de parãmetros será exclusivamente pelo mecanismo de cópia. Há aqui uma sutileza: em *C*, vetores decaem para ponteiros ao serem passados como argumento e, portanto, são passados por referência; em *CML* não há ponteiros e os vetores são copiados.

Parametrização de tipos não será possível na linguagem e os argumentos das funções serão avaliados *a priori*.