ESCOLA
SUPERIOR
DE TECNOLOGIA
E GESTÃO

Trabalho Prático

Licenciatura em Engenharia Informática

2022/2023

Grupo:

Caricas77

Unidade Curricular:

Inteligência Artificial

Docente:

Davide Carneiro

Autores:

8200591, Nuno Castro

8200595, Gonçalo Oliveira

Índice

Índice de figuras

Índice de excertos de código

1. Introdução

1.1. Visão geral do problema e objetivo do trabalho

O principal objetivo do trabalho prático é o de implementar um processo de Meta-Learning em que se usa um Algoritmo Genético para guiar o processo de busca pelo melhor modelo de Machine Learning. O problema específico a resolver é o da classificação de dígitos manuscritos, na área da Visão por Computador. Nesse sentido, será usado o dataset MNIST.

O dataset MNIST foi proposto em 1998 para a identificação de números escritos à mão. É um dos mais conhecidos datasets que representa um problema que é tipicamente fácil para os humanos de resolver, mas difícil para a "máquina", sendo representado por uma base de dados de dígitos escritos à mão com um conjunto de treino composto por 60 000 exemplos (train.csv), e um conjunto de teste composto por 10 000 exemplos (test.csv).

Cada dígito é representado por uma matriz 28x28, totalizando 784 valores de pixels em escala de cinzento (features), que variam entre 0 (branco) e 255 (preto). O dataset contém ainda uma coluna (label) que contém um valor entre 0 e 9 que identifica o dígito representado. Trata-se, portanto, de um problema de ML supervisionado. A Figura 1 mostra alguns dos dígitos contidos no dataset. Como é possível notar, cada número pode ser escrito de formas muito diferentes, havendo ainda números que facilmente se confundem com outros, o que torna o problema complexo.

Figura 1 - Dataset MNIST

2. Algoritmo Genético e Problema de Otimização

2.1. Conceitos e aplicações

Um Algoritmo Genético consiste num procedimento iterativo que mantém uma população de estruturas candidatas a soluções, para domínios específicos. A cada incremento de tempo (geração), as estruturas da população atual são avaliadas na sua capacidade de serem soluções válidas para o domínio do problema e é formada uma nova população de soluções candidatas, baseada na sua avaliação, desenvolvida pela aplicação de operadores genéticos.

Este conceito não garante resultados ótimos, pelo que é mais indicado para problemas de melhoria de soluções (problemas de otimização) e em problemas cujo cálculo de soluções é difícil ou impossível.

Inicialização

- Após definir a estrutura de uma solução, é necessário implementar o processo pelo qual é possível inicializar soluções
- Cada solução deve ser inicializada aleatoriamente, de forma a cobrir da melhor forma o espaço de pesquisa
- É frequente a inicialização aleatória resultar em soluções inválidas, o que por vezes é desejável
- É inicializado um vetor de soluções (primeira geração da população)

Fitness

- Ao existir um conjunto de soluções, torna-se necessário avaliar quão boa cada uma delas é
- Será com base nesta avaliação que as melhores soluções serão escolhidas para darem origem à geração seguinte
- Para isto, é definida a função de fitness que aceita como parâmetro uma solução, e devolve um valor numérico que representa quão boa uma solução é
    - O valor não é geralmente absoluto, mas relativo visto que normalmente não se conhece o melhor/pior caso. Apenas permite comparar as soluções entre si, não sabemos quão longe estamos da melhor solução possível

Seleção

- Na fase de seleção, escolhem-se os indivíduos que darão origem à nova geração
- A Seleção de progenitores para reprodução deve garantir que os indivíduos com maior valor de aptidão tenham, proporcionalmente, mais descendentes
- Idealmente, cada indivíduo deverá ser representado por um número de descendentes equivalente ao rácio entre o seu valor de aptidão e o valor médio da população

- Existem diversas estratégias de Seleção:
    - Baseados no Fitness – Utilizar os melhores N elementos da população para gerar a próxima população
    - Baseado na idade – São “expulsas” as N soluções mais antigas

Reprodução

- A fase de reprodução tem como objetivo criar a nova geração seguinte
- Cada nova geração é criada através da aplicação de operadores genéticos à geração anterior
    - Mutação
    - Cruzamento
- Tipicamente, é definida na configuração a proporção de aplicação destes operadores
    - Qual a percentagem da nova geração que deve ser gerada por reprodução, por cruzamento, e qual deve ser mantida inalterada

Mutação

- O operador genético Mutação pretende ser uma analogia com a mutação genética dos seres vivos
- A mutação é um operador genético unário: aplica-se a um indivíduo da população para gerar um novo indivíduo para a população seguinte

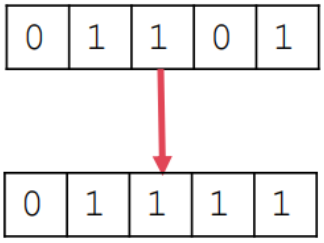

Figura 2 - Processo de Mutação

Cruzamento

- O operador genético Cruzamento é uma analogia com a reprodução sexuada dos seres vivos
- O objetivo do cruzamento é o de conseguir, nos descendentes, uma combinação do material genético dos progenitores, para isto é aplicado a dois indivíduos da população, produzindo outros dois indivíduos para a população da geração seguinte
- Existem diferentes formas de implementar o operador sendo a que a utilizada foi a seleção aleatória de x parâmetros, trocando-os entre os seus progenitores.

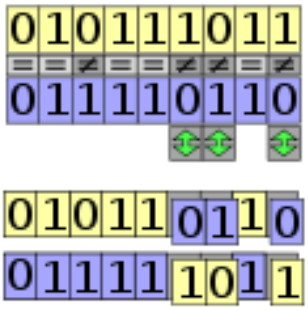

Figura 3 - Processo de cruzamento

2.2. Modelagem da solução e representação genética

Para representar as configurações dos diferentes algoritmos, foram criadas 3 classes tendo em conta o modelo em questão:

- ItemDeepLearning
- ItemRandomForest
- ItemGradientBoostingMachine

Cada classe "Item" contém um conjunto de configurações, alinhadas com o modelo em questão. No excerto de código abaixo, é possível visualizar o exemplo de uma classe "Item" simplificada, que contém associada os atributos utilizados para manipular o modelo, e os métodos para gerar novos valores para esses atributos.

De forma a simplificar os dados repetíveis foi criada uma classe Parent (Item) que contém esses mesmos dados.

```
class Item:
    def __init__(self, seed):
        self.seed = seed

    @staticmethod
    def generate_seed():
        return random.choice(
            rangeInt(
                1,
                int(sys.maxsize / (10**random.randint(0,
len(str(sys.maxsize))-1))),
                Config.get_step_seed()
            )
        )

    def __eq__(self, item):
        if not isinstance(item, type(self)):
            return False
        self_attributes = vars(self)
        item_attributes = vars(item)
        return self_attributes == item_attributes

    def __hash__(self):
        attributes = vars(self)
        attribute_values = []
        for value in attributes.values():
            if isinstance(value, list):
                attribute_values.append(tuple(value))
            else:
                attribute_values.append(value)
        return hash(tuple(attribute_values))

    def __str__(self) -> str:
        attributes = vars(self)
        attribute_strs = [f"{key} = {str(value)}" for key, value in
attributes.items()]
        return ", ".join(attribute_strs)
```

Excerto de código 1 - Classe Item

```
class ItemDeepLearning(Item):
    def __init__(self, hidden, epochs, mini_batch_size, rate,
activation, seed, adaptive_rate=False, stopping_tolerance =
Config.get_dl_stopping_tolerance(), stopping_rounds =
Config.get_dl_stopping_rounds()) -> None:
        self.hidden = hidden
        self.epochs = epochs
        self.mini_batch_size = mini_batch_size
        self.rate = rate
        self.activation = activation
        super().__init__(seed)
        self.adaptive_rate = adaptive_rate
        self.stopping_tolerance = stopping_tolerance
        self.stopping_rounds = stopping_rounds

    @staticmethod
    def generate_item():
        return ItemDeepLearning(
            hidden = ItemDeepLearning.generate_hidden(),
            epochs = ItemDeepLearning.generate_epochs(),
            mini_batch_size =
ItemDeepLearning.generate_mini_batch_size(),
            rate = ItemDeepLearning.generate_rate(),
            activation = ItemDeepLearning.generate_activation(),
            seed = ItemDeepLearning.generate_seed(),
        )
```

Excerto de código 2 - Classe ItemDeepLearning

Para representar a qualidade de cada solução, foi criada a classe Solution que contém um "Item" e está associado a um fitness. Além disso, é possível obter o tipo do "Item", ou seja, o tipo do modelo ML em questão, como é possível ver no excerto de código abaixo.

```
class Solution:
    def __init__(self, item) -> None:
        if self._getType(item) == None:
            raise TypeError("O argumento 'item' deve ser uma instância
de 'Item'.")

        self.item = copy.deepcopy(item)

        self.fitness = None

    @staticmethod
    def _getType(item) -> SolutionType:
        switcher = {
            ItemDeepLearning: SolutionType.DEEP_LEARNING,
            ItemRandomForest: SolutionType.RANDOM_FOREST,
            ItemGradientBoostingMachine:
SolutionType.GRADIENT_BOOSTING_MACHINE,
        }
        return switcher.get(type(item), None)

    def getType(self) -> SolutionType:
        return self._getType(self.item)

    def __hash__(self):
        return hash((self.getType(), self.item))

    def __eq__(self, solution):
        if not isinstance(solution, Solution):
            return False

        return self.item == solution.item

    def __str__(self) -> str:
        return f'Type: {self.getType().value} | Configuration:
{self.item.__str__()} | Fitness: {str(self.fitness)}'
```

Excerto de código 3 - Classe Solution

3. Implementação do Algoritmo Genético

3.1. Lógica implementada

Inicialização da população

Para inicializar a população, foi criada uma função para dividir a mesma pelo número de modelos existentes. Por exemplo, caso o tamanho da população seja 100, a função irá retornar [34, 33, 33] (Como existem 3 modelos diferentes, foi divido por 3). Em seguida é criado um objeto que contém como key's o tipo do modelo e como valor um array que contém o conjunto das configurações geradas para essa modelo.

```
def init_pop(self):
    popArray = divide_number(Config.get_pop_size(), 3)

    popModels= {}
    popModels[SolutionType.DEEP_LEARNING]=[Solution(ItemDeepLearning.gen
erate_item()) for _ in range(popArray[0])]
    popModels[SolutionType.RANDOM_FOREST]=[Solution(ItemRandomForest.gen
erate_item()) for _ in range(popArray[1])]
    popModels[SolutionType.GRADIENT_BOOSTING_MACHINE]=[Solution(ItemGrad
ientBoostingMachine.generate_item()) for _ in range(popArray[2])]

    return popModels
```

Excerto de código 4 - Inicialização da população

Calcular o valor do fitness

Para calcular o fitness foram utilizadas as métricas de performance indicadas abaixo. Além disso, é possível configurar o peso que cada um tem ao calcular o fitness de uma determinada solução.

- Rmse
- Logloss
- Mean_per_class_error
- Accuracy

O valor final retornado pode estar entre 0 e 1, sendo que quanto mais perto de 1, melhor será o fitness.

```
def calculate_fitness(self, metrics: Metrics) -> float:
    # Apply the weights to each performance metric
    weighted_mse = Config.get_fitness_mse_percent() * metrics.mse
    weighted_rmse = Config.get_fitness_rmse_percent() * metrics.rmse
    weighted_logloss = Config.get_fitness_logloss_percent() *
metrics.logloss
    weighted_mean_per_class_error =
Config.get_fitness_mean_per_class_error_percent() *
metrics.mean_per_class_error
    # We want to minimize (1 - r2) to maximize r2
    weighted_r2 = Config.get_fitness_r2_percent() * (1 - metrics.r2)
    # We want to minimize (1 - accuracy) to maximize accuracy
    weighted_accuracy = Config.get_fitness_accuracy_percent() * (1 -
metrics.accuracy)

    # Calculate the fitness
    fitness = 1 / (1 + weighted_mse + weighted_rmse + weighted_logloss +
weighted_mean_per_class_error + weighted_r2 + weighted_accuracy)

    return fitness
```

Excerto de código 5 - Cálculo do fitness

Fitness

A função de fitness treina o modelo de uma determinada solução, tendo em consideração o seu tipo (ML, RF, GBM), em seguida obtém as métricas de performance do mesmo e associa o fitness à solução em questão

```
def fitness(self, sol: Solution):
    try:
        # Train model
        model = self.get_model(sol)

        metrics =
self.MachineLearning.get_model_performance(model=model)

        sol.fitness = self.calculate_fitness(metrics)
    except:
        sol.fitness = 0
```

Excerto de código 6 - Treino do modelo e obtenção de métricas para o fitness

Mutação

A mutação foi realizada, gerando novas configurações para um número aleatório de parâmetros a partir de uma determinada solução. Além disso, existe a possibilidade de configurar essa percentagem (número de parâmetros que podem ser alterados).

Por exemplo, tendo em conta a configuração Random Forest abaixo:

- ntrees=85,
- max_depth=14,
- min_rows=1.5,
- sample_rate=0.95,
- seed=91

É gerado um número aleatório entre 1 e o número de parâmetros da configuração em questão, neste caso 5.

Supondo o número aleatório é 2, então serão trocados 2 parâmetros de forma aleatória, como por exemplo:

- ntrees=80,
- max_depth=10,

```
def mutate(self, sol: Solution):
        new_sol = copy.deepcopy(sol)
        new_sol.fitness = None

        # Select the variables that are changing
        variables = self.get_variables(sol1)
        num_parameters = len(variables)
        num_parameters_changing = random.randint(1, (int(num_parameters
* Config.get_mutation_prob_replace_params_values()) + 1))
        variables = list(vars(sol.item).keys())
        variables_changing = random.sample(variables,
num_parameters_changing)

        # Generate the new values for the variables selected
        for variable in variables_changing:
            setattr(new_sol.item, variable, getattr(new_sol.item,
f"generate_{variable}")())
```

Excerto de código 7 - Mutação

Cruzamento

O cruzamento foi realizado, trocando configurações de um número aleatório de parâmetros a partir de 2 soluções diferentes. Além disso, existe a possibilidade de configurar essa percentagem (número de parâmetros que podem ser trocados).

Por exemplo, tendo em conta as duas configurações Random Forest abaixo:

• ntrees=85, • max_depth=14, • min_rows=1.5, • sample_rate=0.95, • seed=91	• ntrees=50, • max_depth=20, • min_rows=1.0, • sample_rate=1.0, • seed=63507853396

É gerado um número aleatório entre 1 e o número de parâmetros da configuração em questão, neste caso 5.

Supondo que o número aleatório é 3, então serão escolhidos 3 parâmetros de forma aleatória e posteriormente trocados, gerando 2 novas configurações, como exemplificado abaixo:

• ntrees=50, • max_depth=20,	• ntrees=85, • max_depth=14,

• min_rows=1.5, • sample_rate=1.0, • seed=91	• min_rows=1.0, • sample_rate=0.95, • seed=63507853396

```
def crossover(self, sol1: Solution, sol2: Solution):
    new_sol1 = copy.deepcopy(sol1)
    new_sol1.fitness = None

    new_sol2 = copy.deepcopy(sol2)
    new_sol2.fitness = None

    variables = self.get_variables(sol1)
    num_parameters = len(variables)
    num_parameters_changing = random.choice(rangeInt(1,
int(num_parameters *
Config.get_max_prec_crossover_replace_params_values())))
    variables_changing = random.sample(variables,
num_parameters_changing)

    for variable in variables_changing:
        setattr(new_sol2.item, variable, getattr(sol1.item, variable))
        setattr(new_sol1.item, variable, getattr(sol2.item, variable))

    return new_sol1, new_sol2
```

Excerto de código 8 - Cruzamento

3.2. Estratégias de seleção/reprodução

Seleção

A função de seleção foi realizada, selecionando a cada subpopulação metade das suas soluções. Dentro dessa seleção, uma percentagem configurável corresponde às melhores soluções, enquanto o restante é escolhido aleatoriamente na lista. Além disso, a função evita selecionar dados repetidos. Caso não haja dados suficientes para atingir o tamanho pretendido da nova população, são geradas novas soluções.

```
def select(self, pop):
    selected = {}
    for solution_type in SolutionType:
        selected[solution_type] = self.selectType(pop[solution_type])
    return selected

def selectType(self, pop):
    # Calculate the selection size based on the population size
    selection_size = math.ceil(len(pop) / 2)
    # Calculate the size of the random selection as % of the selection
size
    random_selection_size = int(round(selection_size *
Config.get_select_random_percent()))
    random_size = 0

    # Sort the population based on fitness in descending order
    sorted_population = list(set(pop))
    sorted_population.sort(key=lambda solution: solution.fitness,
reverse=True)

    # Adjust the selection sizes if the sorted population is smaller
    if len(sorted_population) < selection_size:
        random_selection_size = 0
        # Calculate the number of random individuals needed to fill the
selection
        random_size = selection_size - len(sorted_population)
        selection_size = len(sorted_population)
```

Excerto de código 9 - Seleção

```
    # Select the top individuals based on fitness
    new_population = sorted_population[:selection_size -
random_selection_size]

    # Select the remaining individuals randomly
    random_positions = random.sample(range(selection_size -
random_selection_size, len(sorted_population)), k=random_selection_size)
    new_population.extend(sorted_population[i] for i in
random_positions)
```

Excerto de código 10 - Continuação da seleção

Reprodução

A função de reprodução foi executada para gerar a nova população, logo após a seleção. Essa função é responsável por criar soluções adicionais e compor a próxima população. Durante a reprodução, uma percentagem configurável da nova população é destinada à mutação, outra percentagem ao crossover, e o restante é preenchido com soluções aleatórias.

De forma a garantir que o crossover ocorra sem problemas, é verificado se a quantidade de dados é par. Caso contrário, a quantidade de soluções por crossover é ajustada de forma aleatória, sendo decrementada ou incrementada uma solução, e, por consequência, a quantidade de soluções por mutação ou novas soluções aleatórias também é ajustada de forma aleatória.

```
def reproduce(self, pop):
    reproduced = {}
    for solution_type in SolutionType:
        reproduced[solution_type] =
self.reproduceType(pop[solution_type])
    return reproduced
```

Excerto de código 11 - Reprodução

```
def reproduceType(self, pop):
    mutateList,crossoverList, randomList  = divide_list(pop,
Config.get_mutation_rate(), Config.get_crossover_rate())

    new_pop = []
    new_pop.extend(pop)

    for sol in mutateList:
        new_pop.append(self.mutate(sol))

    for i in range(0,len(crossoverList),2):
        c1, c2 = self.crossover(crossoverList[i], crossoverList[i+1])
        new_pop.append(c1)
        new_pop.append(c2)

    for sol in randomList:
        new_pop.append(self.random(sol.getType()))

    return new_pop
```

Excerto de código 12 – Reprodução

3.3. Problemas encontrados e estratégicas implementadas

Durante o desenvolvimento da solução, foram encontrados vários problemas que precisavam de ser resolvidos. Os problemas identificados foram os seguintes:

1. **Elevado tempo de execução do RF e GBM**: O tempo necessário para executar o algoritmo era muito longo. Para resolver esse problema, foram realizadas algumas modificações. No Gradient Boosting Machine (GBM) e no Random Forest (RF), definiu-se o valor de Nfolds a 0, visto que o dataset era constituído com um grande volume de dados de dados, não havendo necessidade de realizar cross-validation e executar várias vezes o mesmo processo para melhorar o modelo.
2. **Critérios de convergência**: Foram adicionados critérios de convergência aos três modelos, como stopping_rounds e stopping_tolerance, para interromper automaticamente o modelo caso a evolução fique estagnada. Para identificar rapidamente os critérios de convergência mais eficazes, realizamos uma pesquisa rápida para identificar os três ou quatro valores mais frequentes. Em seguida, desenvolveu-se um código simples que testou todas as possibilidades, a fim de encontrar os melhores valores para cada algoritmo. Isso permitiu otimizar o processo de forma eficiente.
3. **Elevado volume de dados**: Devido ao elevado volume de dados é possível selecionar apenas uma percentagem dos dados do dataset original, no algoritmo genético para tornar o processo mais rápido. Quando reduzido para metade, o processo era bem mais rápido e os valores de fitness não divergiam muito quando comparados com a utilização do dataset completo.
4. **Grande quantidade de dados repetidos na seleção**: Durante o processo de seleção, foi observado que havia uma grande quantidade de dados repetidos. Para resolver esse problema, os dados repetidos foram removidos antes da seleção e eram geradas aleatoriamente soluções para serem adicionadas à população.
5. **Quantidade ímpar de dados para realizar o crossover**: Houve um problema em relação à quantidade ímpar de dados para realizar o crossover. Para resolver isso, foi verificado se a quantidade de soluções era par. Se não fosse, a quantidade de soluções foi ajustada aleatoriamente, decrementando ou incrementando uma solução. Consequentemente, a quantidade de soluções para mutação e novas soluções aleatórias também foi ajustada aleatoriamente.
6. **Memoria cheia**: Durante a execução, enfrentamos o problema de memória cheia, que ocorria após algumas gerações. No entanto, conseguimos resolver esse problema, ao limpar os modelos da memória a cada geração. Para preservar o histórico das gerações, os dados são armazenados em um arquivo de histórico, contendo os valores de todas as gerações.

A fim de obter o mesmo valor que tínhamos na execução anterior, introduzimos o parâmetro "seed". Dessa forma, podemos garantir que o mesmo modelo seja obtido em outras execuções.

7. **Problema de visualização de dados entre execuções**: Para solucionar o desafio de visualizar os dados entre diferentes execuções do algoritmo, foi implementada a criação de ficheiros de histórico(logs) e guardados os gráficos de evolução. Essa abordagem permite uma visualização persistente e facilita a análise dos dados, tornando mais fácil acompanhar o progresso do algoritmo em diferentes momentos. Com os ficheiros de histórico, é possível visualizar os dados de forma contínua, facilitando a identificação de padrões para aprimorar o desempenho do algoritmo em execuções futuras.
8. **Elevado tempo de execução do algoritmo genético**: A fim de resolver o problema de elevado tempo de execução do algoritmo genético foram criados servidores na plataforma Digital Ocean de forma gratuita usando uma distribuição de servidor Debian. Essa distribuição é considerada mais "limpa" do que um sistema comum pois não contem interfaces e é especializada para servidores, o que permitiu obter melhor desempenho computacional. Além disso, executar os processos nos servidores dedicados possibilitou que executassem por um longo período, sem problemas que poderiam ocorrer em computadores pessoais. Para além das modificações mencionadas anteriormente, o algoritmo permite configurar e aumentar a quantidade de RAM e de número de threads do H2O e assim ter um melhor desempenho.
9. **Treino para submissão:** De forma a garantir uma melhor performance do modelo ao gerá-lo para submissão foram utilizados 100% dos dados para train(não foi dividido em train, test, validation), assim conseguiu-se obter mais dados para o treino do modelo o que permitiu obter uma melhor performance para realizar o processo de identificação dos números.

4. Configuração do Algoritmo Genético

4.1. Configurações customizáveis do algoritmo genético

O algoritmo pode ser customizado por meio do ficheiro ".env". Nesse ficheiro, várias variáveis estão presentes, sendo que aquelas iniciadas com "dl" são específicas para o algoritmo de Deep Learning, as iniciadas com "rf" são relacionadas ao algoritmo Random Forest, as variáveis com prefixo "gbm" estão associadas ao algoritmo Gradient Boosting Machine, as variáveis com "fitness" são utilizadas para a função de avaliação de desempenho, e as variáveis restantes são globais para o algoritmo genético. Além disso, as variáveis com a descrição "min", "max" e "step" serão utilizadas nas funções de geração de valores aleatórios, onde "min" representa o valor mínimo, "max" representa o valor máximo e "step" representa o intervalo entre cada valor dentro do intervalo definido por "min" e "max".

```
#Valores inteiros positivos
rf_min_ntrees = 30
rf_max_ntrees = 150
rf_step_ntrees = 5

#Valores inteiros positivos
rf_min_max_depth = 10
rf_max_max_depth = 31
rf_step_max_depth = 2

#Valores numérico positivos
rf_min_min_rows = 1
rf_max_min_rows = 11
rf_step_min_rows = 0.5

#Valores inteiros positivos
rf_min_sample_rate = 40
rf_max_sample_rate = 100
rf_step_sample_rate = 5

#Valores numérico positivos
rf_stopping_rounds = 15
rf_stopping_tolerance = 0.01
```

Excerto de código 13 – Configurações do Algoritmo Genético

```
#Valores inteiros positivos
dl_min_size_hidden = 1
dl_max_size_hidden = 3
dl_step_size_hidden = 1

#Valores inteiros positivos
dl_min_hidden = 50
dl_max_hidden = 100
dl_step_hidden = 5

#Valores inteiros positivos
dl_min_epochs = 5
dl_max_epochs = 50
dl_step_epochs = 2

#Valores inteiros positivos
dl_min_mini_batch_size=1
dl_max_mini_batch_size=500
dl_step_mini_batch_size=10

#Valores numérico positivos
dl_min_rate=0.001
dl_max_rate=0.01
dl_step_rate=0.001

#Valores numérico positivos
dl_stopping_rounds = 3
dl_stopping_tolerance = 0.01

#Valores inteiros positivos
gbm_min_ntrees = 30
gbm_max_ntrees = 150
gbm_step_ntrees = 5

#Valores inteiros positivos
gbm_min_max_depth = 1
gbm_max_max_depth = 8
gbm_step_max_depth = 1
```

Excerto de código 14 – Configurações do Algoritmo Genético

```
#Valores numérico positivos
gbm_min_min_rows = 1
gbm_max_min_rows = 11
gbm_step_min_rows = 0.5

#Valores inteiros positivos
gbm_min_sample_rate = 40
gbm_max_sample_rate = 100
gbm_step_sample_rate = 5

#Valores inteiros positivos
gbm_min_col_sample_rate = 40
gbm_max_col_sample_rate = 100
gbm_step_col_sample_rate = 5

#Valores numérico positivos
gbm_min_learn_rate=0.01
gbm_max_learn_rate=0.5
gbm_step_learn_rate=0.01

#Valores numérico positivos
gbm_stopping_rounds = 5
gbm_stopping_tolerance = 0.01

#Variavel global aos algoritmos que contem o intervalo entre os valores
da seed
step_seed=5

#Tamnho da população que depois será divida em 3 partes iguais (DL, RF,
GBM)
pop_size = 90
#Número máximo de gerações que o algoritmo pode realizar
max_gen = 30
#Número entre 0 e 1 que representa a percentagem de dados da reprodução
que vão ser para mutação
mutation_rate = 0.5
#Número entre 0 e 1 que representa a percentagem de dados da reprodução
que vão ser para crossover
crossover_rate = 0.35
```

Excerto de código 15 - Configurações do Algoritmo Genético

```
# Percentagem máxima de parâmetros que podem ser alterados na mutação
max_perc_mutation_replace_params_values = 0.5
# Percentagem máxima de parâmetros que podem ser cruzados no crossover
max_perc_crossover_replace_params_values = 0.5
#Valor que poder ser verdadeiro pu Falso e caso seja verdadeiro serão escritos mais dados na consola
verbose = False
#Quantidade de dados random's a serem selecionada na função de seleção
select_random_percent=0.2

#Nota test_ratio+train_ratio<=1 e se test_ratio+train_ratio<1 o restante dos dados vai ser para validação
#Percemtagem de dados para teste
test_ratio=0.15
#Percemtagem de dados para treino
train_ratio=0.7

#Os seguintes são valores entre 0 e 1 sendo que a soma deles tem de ser 1, cada valore é a percentagem de importância no cálculo do fitness
fitness_mse_percent = 0.15
fitness_rmse_percent = 0.15
fitness_logloss_percent = 0.15
fitness_mean_per_class_error_percent = 0.20
fitness_r2_percent = 0.15
fitness_accuracy_percent = 0.20

#Número entre 0 e 1 e representa a percentagem de dados a ser usada no algoritmo genético
data_percentage_use_in_ga=0.5

#Número entre 0 e 1 e representa a percentagem de threads a usar no h2o
n_threads_percentage=1
#Número entre 0 e 1 e representa a percentagem de ram disponivel a usar no h2o
ram_percentage=1
```

Excerto de código 16 - Configurações do Algoritmo Genético

Por fim, o algoritmo faz o download dos datasets, caso seja necessário. No entanto, para realizar essa operação, é preciso configurar o Kaggle.

Para configurar, basta instalar o Kaggle no Python usando o comando "pip install kaggle". Em seguida, é necessário ir ao Kaggle, aceder as " Settings" e clicar em "Create New Token". Por fim, basta criar a pasta ".kaggle" na pasta do utilizador e adicionar o ficheiro com o token à pasta.

5. Resultados e Avaliação

5.1. Melhores modelos encontrados

Para treinar os modelos, dividiu-se o dataset em 70% para treino, 15% para teste e outros 15% para validação.

Para calcular o fitness dos modelos, foram utilizados os pesos abaixo:

```
fitness_mse_percent = 0
fitness_rmse_percent = 0.20
fitness_logloss_percent = 0.20
fitness_mean_per_class_error_percent = 0.30
fitness_r2_percent = 0
fitness_accuracy_percent = 0.30
```

Excerto de código 17 - Pesos do fitness

O **melhor modelo** gerado, foi encontrado na geração 22, com um tamanho de população de 30 e retornou a seguinte configuração do H2OGradientBoostingEstimator:

```
ItemGradientBoostingMachine(
        ntrees=450,
        max_depth=7,
        min_rows=10,
        sample_rate=1.0,
        col_sample_rate=1.0,
        learn_rate=0.06999999999999999,
        seed=6220331565186546
)
```

Excerto de código 18 - Configuração do melhor modelo GBM

Ao utilizar as configurações acima, foi obtido um fitness de 93.73%, que foi calculado utilizando para cada métrica de performance os pesos abaixo:

- **RMSE**: 0.14240027431160754,
- **Log Loss**: 0.12327794988393383,
- **Mean Per Class Error**: 0.02298576667172355,
- **Accuracy**: 0.977128899744643

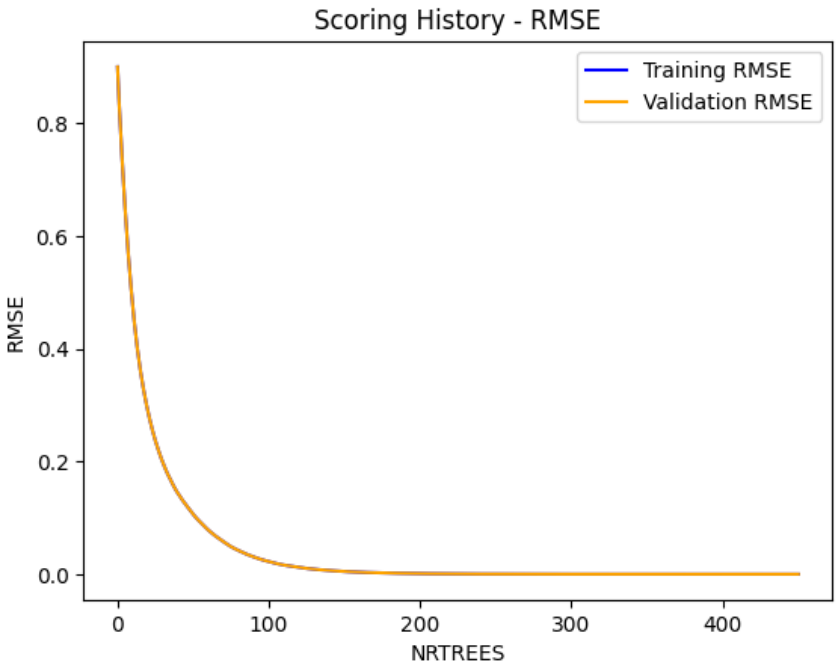

Figura 4- Scoring history RMSE: Modelo GradientBoosting Machine

A partir do gráfico anterior é possível visualizar, que o modelo gerado aparenta ser um bom modelo visto que, o RMSE diminui à medida que o modelo é treinado e a diferença para os dados de validação é pequena.

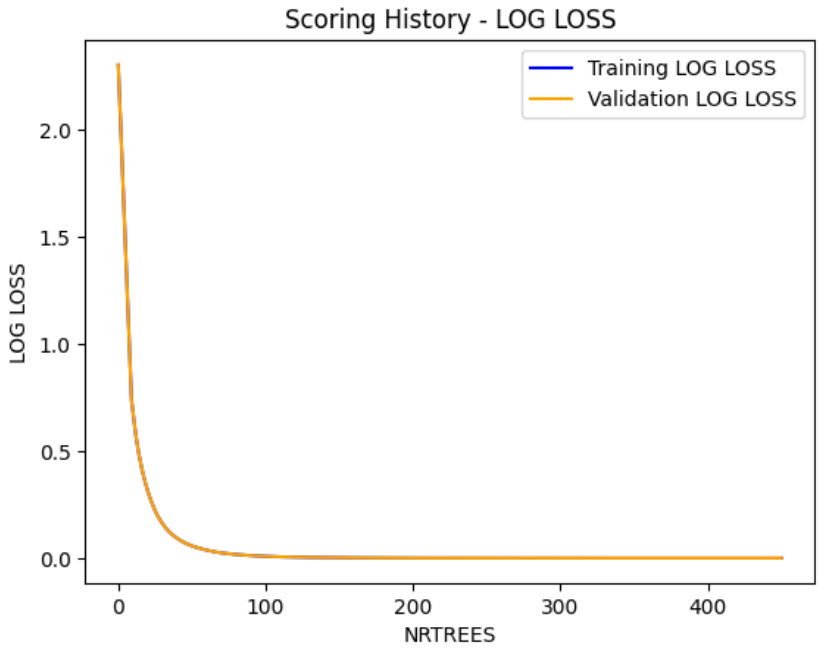

Figura 5 - Scoring history Losslog: Modelo GradientBoosting Machine

A partir do gráfico anterior é possível visualizar, que o modelo gerado aparenta ser um bom modelo visto que, as linhas de treino e validação são próximas, descem inicialmente, e estagnam perto do fim.

O **segundo melhor modelo** gerado, foi encontrado na geração 19, com um tamanho de população de 30 e retornou a seguinte configuração do H2ODeepLearningEstimator:

```
ItemDeepLearning(
        hidden=[95, 95, 70],
        epochs=39,
        mini_batch_size=371,
        rate= 0.002,
        seed=61824101,
        activation="rectifier"
)
```

Excerto de código 19 - Configuração melhor do modelo Deeplearning

Ao utilizar as configurações acima, foi obtido um fitness de 91.41%, que foi calculado tendo em conta as métricas de performance abaixo:

- **RMSE**: 0.1577500551324804,
- **Log Loss**: 0.2124635912515343,
- **Mean Per Class Error**: 0.028129212073550198,
- **Accuracy**: 0.9719704598858677

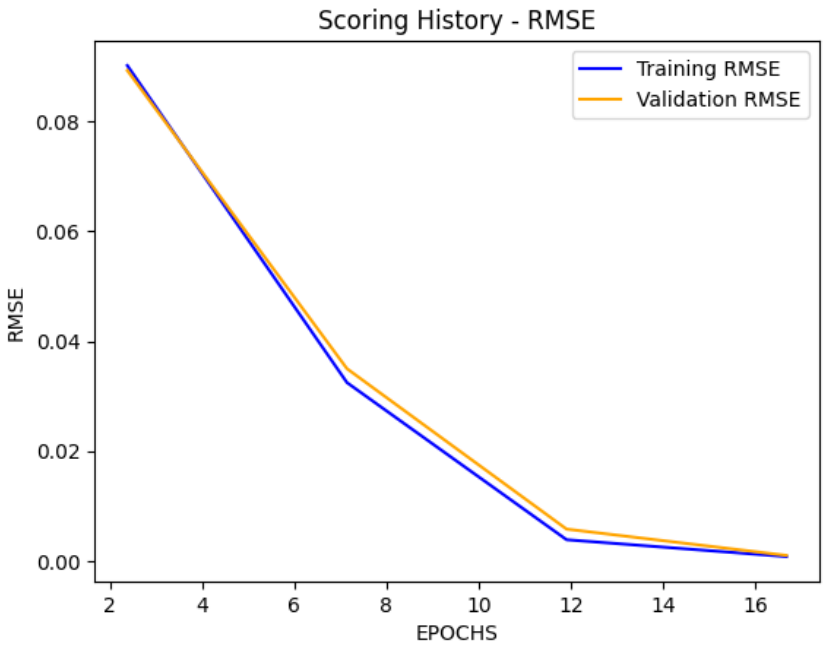

Figura 6 - Scoring history RMSE: Modelo DeepLearning

A partir do gráfico anterior é possível visualizar, que o modelo gerado aparenta ser um bom modelo visto que, o RMSE diminui à medida que o modelo é treinado e a diferença para os dados de validação é pequena.

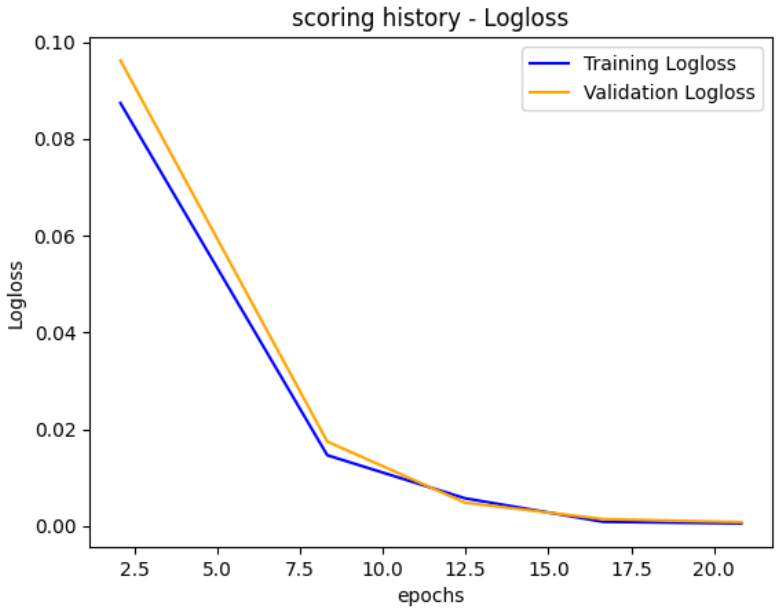

Figura 7 - Scoring history LogLoss: Modelo DeepLearning

A partir do gráfico anterior é possível visualizar, que o modelo gerado aparenta ser um bom modelo visto que, as linhas de treino e validação são próximas, descem inicialmente, e estagnam perto do fim.

5.2. Análise dos gráficos da evolução do fitness

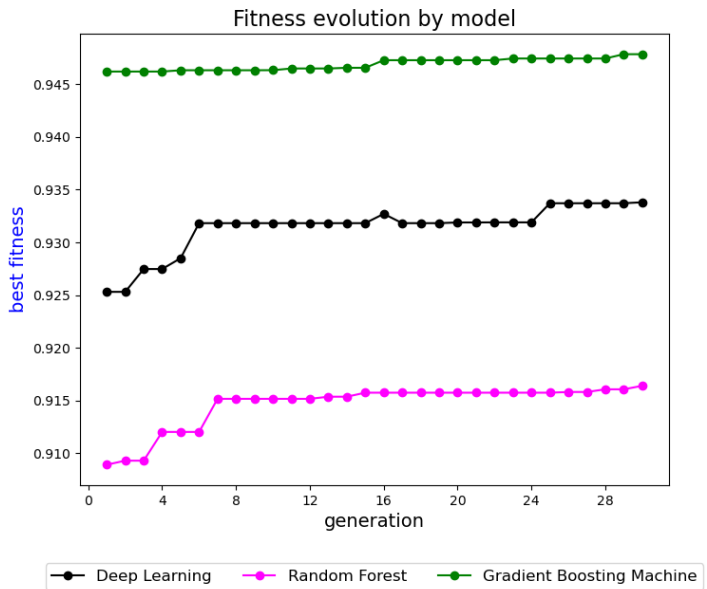

Figura 8 - Evolução do fitness das melhores soluções por geração e por modelo

Ao analisar a imagem anterior, fica evidente que os modelos de GBM apresentam um desempenho superior em comparação com os modelos de DL e, por último, com os de RF. Além disso, podemos observar que os dados têm uma tendência de crescimento a cada nova geração, embora nas últimas gerações essa tendência de crescimento tenha diminuído em relação ao início.

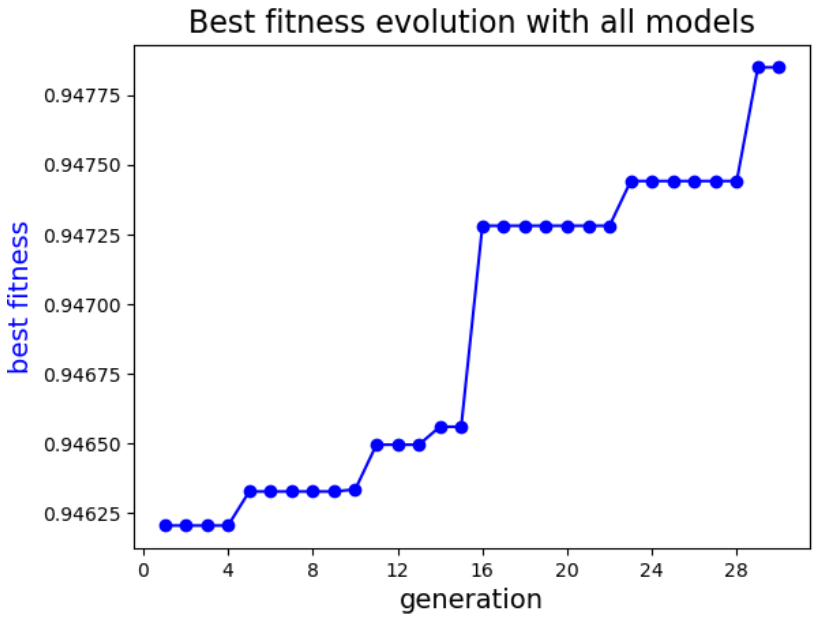

Figura 9 - Evolução do fitness das melhores soluções por geração

Ao analisar a imagem anterior, podemos observar que o algoritmo apresenta uma tendência de crescimento contínuo. Nota-se que, quando ocorre esse crescimento, ele tende a se manter-se por algumas gerações.

6. Conclusão

A realização deste trabalho prático foi uma experiência enriquecedora, pois permitiu consolidar conhecimentos sobre vários conceitos de Inteligência Artificial, nomeadamente em relação aos algoritmos genéticos e aos algoritmos de Machine Learning. Durante o processo de aprendizagem, foram abordados temas como o treino de modelos, a seleção de recursos, a otimização de parâmetros, entre outros. Foi interessante aprender como os algoritmos genéticos podem ser utilizados para encontrar soluções ótimas em problemas complexos.