

Objetivo geral

Compreender, **implementar e comparar** algoritmos clássicos de IA e Computação Natural, prestando e analisando dados reais. O(a) aluno(a) deverá identificar se o problema é de **classificação** ou **regressão**, aplicar os **tratamentos necessários** à base (limpeza, imputação, codificação, escalonamento) e reportar resultados de forma reproduzível.

Escopo do trabalho

Parte 1 — Árvore de Decisão *manual* (10 perguntas)

- Implementar **do zero** (sem `sklearn.tree`) uma árvore de decisão com pelo menos **10 perguntas** binárias (sim/não).
- Tema livre (ex.: orientação de hobby/carreira, triagem simples, etc.).
- Entregar: um .py executável por terminal e um pequeno diagrama (texto/mermaid) da árvore.

Diagrama da árvore em Mermaid (entrega)

Mermaid é uma linguagem simples para criar diagramas em arquivos .md. O GitHub e editores como VS Code (com extensão) renderizam automaticamente.

Como entregar

- Criar `src/part1_tree_manual/tree_diagram.md` com um bloco `mermaid`.
- Referenciar esse arquivo no `README.md`.
- (Opcional) Exportar para imagem (SVG/PNG) para incluir no relatório.

Exemplo 1 — Fluxograma (árvore de decisão)

```
1 flowchart TD
2 start([Inicio]) --> q1{Gosta de trabalhar em equipe?}
3 q1 -- Sim --> q2{Prefere atividades ao ar livre?}
4 q1 -- Não --> q3{Gosta de lidar com números?}
5
6 q2 -- Sim --> s1[Esportes coletivos / trilhas]
7 q2 -- Não --> q4{Confortavel com tecnologia?}
8 q4 -- Sim --> s2[Tecnologia / programacao]
9 q4 -- Não --> s3[Hobbies criativos]
10
11 q3 -- Sim --> s4[Financias / engenharia]
12 q3 -- Não --> q5{Habilidade artistica?}
13 q5 -- Sim --> s5[Design / audiovisual]
14 q5 -- Não --> s6[Leitura / fotografia]
```

Listing 1. Bloco Mermaid para fluxograma

Exemplo 2 — Fluxograma com 10 perguntas (esqueleto)

```
1 flowchart TD
2 q1{P1} -->|Sim| q2{P2}
3 q1 -->|Não| q3{P3}
4 q2 -->|Sim| q4{P4}
5 q2 -->|Não| q5{P5}
6 q3 -->|Sim| q6{P6}
7 q3 -->|Não| q7{P7}
8 q4 -->|Sim| q8{P8}
9 q4 -->|Não| q9{P9}
10 q5 -->|Sim| q10{P10}
11 q5 -->|Não| sA[Saida A]
12 q8 -->|Sim| sB[Saida B]
13 q8 -->|Não| sC[Saida C]
14 q9 -->|Sim| sD[Saida D]
15 q9 -->|Não| sE[Saida E]
16 q10 -->|Sim| sF[Saida F]
17 q10 -->|Não| sG[Saida G]
18 q6 --> sH[Saida H]
19 q7 --> sI[Saida I]
```

Listing 2. Estrutura P1..P10 na árvore

Exemplo 3 — Mindmap (alternativo)

```
1 mindmap
2 root((Arvore de Decisao))
3   Equipe?
4     Sim
5       Ar livre?
6         Sim::::ok Esportes coletivos
7         Não Tecnologia?
8           Sim TI/Programacao
9           Não Hobbies criativos
10      Não
11        Numeros?
12          Sim Financas/Engenharia
13          Não Artístico?
14            Sim Design/Audiovisual
15            Não Leitura/Fotografia
```

Listing 3. Mindmap em Mermaid

Exportar Mermaid para imagem (opcional, via CLI)

Requer Node.js:

```
1 # Instalar o CLI (sem salvar no projeto)
2 npx -y @mermaid-js/mermaid-cli -i src/part1_tree_manual/tree_diagram.md -o reports/figs/tree.svg
3 # No LaTeX/relatório, incluir como:
4 # \includegraphics[width=\ linewidth]{reports/figs/tree.svg}
```

Listing 4. Exportando .md para SVG/PNG

Parte 2 — Supervisionado (Kaggle/UCI): KNN, SVM e Árvore

- Escolher **um** dataset do **UCI** (<https://archive.ics.uci.edu/>) ou **Kaggle** (<https://www.kaggle.com>).

- **Preparar** a base: lidar com nulos, categóricas (one-hot), escalonamento para KNN/SVM, divisão treino/teste (estratificada em classificação) e **validação cruzada (k-fold)**.
- Treinar e avaliar:
 1. **KNN**
 2. **SVM**
 3. **Árvore de Decisão** (pode usar `scikit-learn`)
- **Métricas**:
Classificação: acurácia, precisão, revocação, F1 (macro); opcional ROC-AUC e matriz de confusão.
Regressão: MAE, RMSE, R^2 .
- Entregar: códigos, tabela comparativa de métricas, discussão breve de pré-processamento e resultados.

Parte 3 — Algoritmo Genético (AG)

- Implementar um **AG do zero** (sem libs de GA) e escolher **um** problema dentre os 12 abaixo:
1. Mochila 0–1 (Knapsack)
 2. Caixeiro Viajante (TSP)
 3. Bin Packing
 4. N-Rainhas
 5. Escalonamento de tarefas (scheduling)
 6. Timetabling (alocação de aulas/recursos)
 7. Seleção de atributos (feature selection)
 8. Ajuste de hiperparâmetros simples (ex.: $C\gamma$ do SVM)
 9. Otimização contínua (Rastrigin/Sphere)
 10. Regressão simbólica (expressões simples)
 11. Planejamento de rotas com restrições
 12. Portfólio simples com restrições (soma=1, risco/retorno)

Relatar: representação (codificação), fitness, operadores (seleção, cruzamento, mutação), parâmetros, critério de parada e resultados.

Parte 4 — Enxame e Imunes¹

Implemente **um algoritmo de enxame e um algoritmo imune**, para **qualquer** dos 12 problemas (pode usar o mesmo problema para comparar):

- **Enxame**: *PSO* (Particle Swarm Optimization) **ou** *ACO* (Ant Colony; sugere-se TSP).
- **Imunes**: *CLONALG* (Seleção Clonal) **ou** *NSA* (Seleção Negativa).
- **Relate**: modelagem, parâmetros, critério de parada, melhores soluções e comparação breve com o AG.

¹Interpretação do enunciado: “algoritmos de *enxame e imunes*”. Se desejar outra família, sinalize.

Diretrizes técnicas mínimas

- Linguagem: **Python 3.10+**.
- Permitido: numpy, pandas, matplotlib, scikit-learn (*somente* Parte 2), scipy e seaborn (opcional).
- **Proibido nas partes de implementação:** bibliotecas que já forneçam AG/PSO/ACO/AIS prontos.
- Reproduzibilidade: requirements.txt, run.sh/Makefile, seed fixa quando aplicável.
- Dados: incluir *amostras* pequenas no repositório ou script de download.

Entrega (SIGAA) e repositório

- Submeter no SIGAA: **link do repositório Git público** (GitHub/GitLab/Bitbucket) e **PDF do relatório**.
- Criar uma **tag** de versão: v1.0-submissao.
- O README deve permitir rodar cada parte com **um comando**.

Template de repositório (oficial)

```
1 ia-trabalho-2025-2/
2 |- README.md
3 |- LICENSE
4 |- requirements.txt
5 |- Makefile
6 |- run.sh
7 |- data/
8 |   |- raw/
9 |   |- external/
10 |   |- processed/
11 |- notebooks/
12 |   |- EDA.ipynb
13 |- src/
14 |   |- part1_tree_manual/
15 |   |   |- tree_manual.py
16 |   |   |- tree_diagram.md
17 |   |- part2_ml/
18 |   |   |- preprocess.py
19 |   |   |- train_knn.py
20 |   |   |- train_svm.py
21 |   |   |- train_tree.py
22 |   |   |- utils_metrics.py
23 |   |- part3_ga/
24 |   |   |- ga.py
25 |   |   |- problems/
26 |   |   |   |- knapsack.py
27 |   |   |   |- tsp.py
28 |   |   |   ...
29 |   |   |- run_ga.py
30 |   |- part4_swarm_immune/
31 |   |   |- pso.py      # ou aco.py
32 |   |   |- clonalg.py # ou nsa.py
33 |   |   |- run_meta.py
34 |   |- common/
35 |       |- utils_io.py
36 |       |- seeds.py
```

```

37 |- reports/
38 |   |- report.pdf
39 |   |- figs/
40 |- tests/
41   |- test_ga.py
42   |- test_swarm.py

```

Listing 5. Estrutura de diretórios sugerida

README.md (modelo)

```

1 # Trabalho Prático IA (2025/2)
2
3 ## Como reproduzir
4 python -m venv .venv && source .venv/bin/activate
5 pip install -r requirements.txt
6 # Parte 1
7 python src/part1_tree_manual/tree_manual.py
8 # Parte 2
9 python src/part2_ml/train_knn.py
10 python src/part2_ml/train_svm.py
11 python src/part2_ml/train_tree.py
12 # Parte 3
13 python src/part3_ga/run_ga.py --problem tsp --iters 2000
14 # Parte 4
15 python src/part4_swarm_immune/run_meta.py --algo pso --problem rastrigin
16
17 ## Decisões técnicas
18 - Dataset: NOME (Kaggle/UCI) link.
19 - Tarefa: Classificação/Regressão métricas usadas.
20 - Pré-processamento: imputação, encoding, scaling.
21 - Validação: hold-out + k-fold (k=?).
22 - Sementes aleatórias: ver `src/common/seeds.py` .

```

Listing 6. Conteúdo base de README.md

requirements.txt (mínimo)

```

1 numpy
2 pandas
3 matplotlib
4 scikit-learn
5 scipy
6 seaborn

```

Listing 7. requirements.txt

Makefile (atalhos)

```

1 .PHONY: setup part1 part2 part3 part4 clean
2
3 setup:
4 \t python -m venv .venv && .venv/bin/activate && pip install -r requirements.txt
5
6 part1:
7 \t python src/part1_tree_manual/tree_manual.py
8
9 part2:
10 \t python src/part2_ml/train_knn.py && \
11 \t python src/part2_ml/train_svm.py && \
12 \t python src/part2_ml/train_tree.py

```

```
13
14 part3:
15 \t python src/part3_ga/run_ga.py --problem tsp --iters 2000
16
17 part4:
18 \t python src/part4_swarm_immune/run_meta.py --algo pso --problem rastrigin
19
20 clean:
21 \trm -rf __pycache__ .pytest_cache data/processed/* reports/figs/*
```

Listing 8. Makefile

```
run.sh
1 #!/usr/bin/env bash
2 set -e
3 python -m venv .venv
4 source .venv/bin/activate
5 pip install -r requirements.txt
6 make part1
7 make part2
8 make part3
9 make part4
```

Listing 9. run.sh

Relatório (PDF)

Estrutura (2–6 páginas):

1. Resumo (5–8 linhas)
2. Parte 1: árvore, diagrama/mermaid, exemplos de execução
3. Parte 2: dataset (fonte/link), tipo de tarefa, pré-processamento, **tabela comparativa** e discussão
4. Parte 3: AG — modelagem, parâmetros, resultados (gráfico fitness vs. iteração opcional)
5. Parte 4: Enxame e Imunes — problema, parâmetros, resultados; comparação breve com AG
6. Conclusões e trabalhos futuros
7. Reproduzibilidade: como rodar, tag de versão
8. Referências

Rubrica (100 pts)

- **Parte 1 — Árvore manual (10%)**: lógica, clareza, diagrama.
- **Parte 2 — ML (35%)**: preparo da base (10), métricas (5), resultados & comparação (15), discussão (5).
- **Parte 3 — AG (20%)**: implementação (10), modelagem/fitness (5), resultados e análise (5).
- **Parte 4 — Enxame & Imunes (25%)**: implementação (12), parametrização (5), resultados e análise (8).
- **Reproduzibilidade & organização (10%)**: README, requirements, Makefile/run.sh, estrutura do repo.

Bônus até +5 pts: testes unitários simples (tests/), gráficos claros, estudo de hiperparâmetros.

Checklist de submissão

- Link do repositório público no SIGAA.
- PDF do relatório no SIGAA.
- Tag v1.0-submissao.
- README com passos de execução por parte.

Apêndice A — Esqueletos opcionais (referência)

Árvore manual (exemplo mínimo)

```

1 def ask(prompt):
2     while True:
3         a = input(f"{prompt} (sim/não): ").strip().lower()
4         if a in {"sim", "s"}: return "sim"
5         if a in {"nao", "não", "n"}: return "nao"
6         print("Use 'sim' ou 'não'.")
7
8 def main():
9     perguntas = [
10         "Gosta de trabalhar em equipe?",
11         "Prefere atividades ao ar livre?",
12         "Gosta de lidar com números?",
13         "Prefere habilidades artísticas?",
14         "Sente-se confortável com tecnologia?",
15         "Gosta de resolver problemas complexos?",
16         "Prefere atividades que envolvam comunicação?",
17         "Interessa-se por cuidar de pessoas?",
18         "Gosta de desafios físicos?",
19         "Prefere ambientes organizados?"
20     ]
21     a = [ask(q) for q in perguntas]
22     # Exemplo de regra simples:
23     if a[0]=="sim" and a[4]=="sim" and a[5]=="sim":
24         print("Sugestão: Engenharia/Computação/Ciência de Dados.")
25     else:
26         print("Sugestão geral: explore o que mais marcou 'sim'.")
27
28 if __name__ == "__main__":
29     main()

```

Algoritmo Genético (GA) — esqueleto

```

1 import random
2
3 class GA:
4     def __init__(self, pop_size, cx_rate, mut_rate, fitness_fn, create_ind, mutate, crossover,
5      max_iters=1000, seed=42):
6         random.seed(seed)
7         self.pop = [create_ind() for _ in range(pop_size)]
8         self.cx_rate, self.mut_rate = cx_rate, mut_rate
9         self.fitness_fn = fitness_fn
10        self.mutate, self.crossover = mutate, crossover
11        self.max_iters = max_iters

```

```

12     def select(self, k=2):
13         cand = random.sample(self.pop, k)
14         cand.sort(key=self.fitness_fn, reverse=True)
15         return cand[0]
16
17     def step(self):
18         new_pop = []
19         while len(new_pop) < len(self.pop):
20             p1, p2 = self.select(), self.select()
21             c1, c2 = p1[:,], p2[:,]
22             if random.random() < self.cx_rate:
23                 c1, c2 = self.crossover(c1, c2)
24             if random.random() < self.mut_rate:
25                 c1 = self.mutate(c1)
26             if random.random() < self.mut_rate:
27                 c2 = self.mutate(c2)
28             new_pop += [c1, c2]
29         self.pop = new_pop[:len(self.pop)]
30
31     def run(self):
32         best = max(self.pop, key=self.fitness_fn)
33         for _ in range(self.max_iters):
34             self.step()
35             cand = max(self.pop, key=self.fitness_fn)
36             if self.fitness_fn(cand) > self.fitness_fn(best):
37                 best = cand
38         return best, self.fitness_fn(best)

```

PSO (exame) — esqueleto

```

1 import numpy as np
2 rng = np.random.default_rng(42)
3
4 class PSO:
5     def __init__(self, f, bounds, n=30, w=0.7, c1=1.4, c2=1.4, iters=500):
6         self.f, self.bounds = f, np.array(bounds) # [(min,max), ...]
7         dim = len(bounds)
8         low, high = self.bounds[:,0], self.bounds[:,1]
9         self.X = rng.uniform(low, high, size=(n, dim))
10        self.V = rng.uniform(-abs(high-low), abs(high-low), size=(n, dim))*0.1
11        self.pbest = self.X.copy()
12        self.pbest_val = np.apply_along_axis(self.f, 1, self.X)
13        gidx = np.argmin(self.pbest_val)
14        self.gbest = self.pbest[gidx].copy()
15        self.gbest_val = self.pbest_val[gidx]
16        self.w, self.c1, self.c2, self.iters = w, c1, c2, iters
17
18    def step(self):
19        r1, r2 = rng.random(self.X.shape), rng.random(self.X.shape)
20        self.V = self.w*self.V + self.c1*r1*(self.pbest - self.X) + self.c2*r2*(self.gbest -
21            self.X)
22        self.X = self.X + self.V
23        self.X = np.minimum(np.maximum(self.X, self.bounds[:,0]), self.bounds[:,1]) # clamp
24        vals = np.apply_along_axis(self.f, 1, self.X)
25        improve = vals < self.pbest_val
26        self.pbest[improve] = self.X[improve]
27        self.pbest_val[improve] = vals[improve]
28        if self.pbest_val.min() < self.gbest_val:
                idx = self.pbest_val.argmin()

```

```

29         self.gbest, self.gbest_val = self.pbest[idx].copy(), self.pbest_val[idx]
30
31     def run(self):
32         for _ in range(self.iters):
33             self.step()
34         return self.gbest, self.gbest_val

```

CLONALG (imunes) — esqueleto

```

1 import numpy as np
2 rng = np.random.default_rng(42)
3
4 class CLONALG:
5     def __init__(self, f, create, mutate, pop=50, clones=5, sel=10, iters=200):
6         self.f, self.create, self.mutate = f, create, mutate
7         self.pop = [create() for _ in range(pop)]
8         self.clones, self.sel, self.iters = clones, sel, iters
9
10    def run(self):
11        best, bestv = None, np.inf
12        for _ in range(self.iters):
13            vals = np.array([self.f(x) for x in self.pop])
14            idx = vals.argsort()[:self.sel] # menor é melhor
15            selected = [self.pop[i] for i in idx]
16            new = []
17            for rank, s in enumerate(selected, start=1):
18                ncl = max(1, self.clones//rank)
19                for _ in range(ncl):
20                    new.append(self.mutate(s))
21            while len(new) < len(self.pop):
22                new.append(self.create())
23            self.pop = new
24            vmin = vals.min()
25            if vmin < bestv:
26                bestv, best = vmin, selected[0]
27        return best, bestv

```