Documentação Técnica e Acadêmica: Sistema de IA para Ordenação de Casos de Teste (IARTES)

Documento de suporte a artigo acadêmico sobre ordenação inteligente de casos de teste

1. Introdução e Objetivo

O IARTES (Sistema de Recomendação Inteligente para Ordenação de Testes) é uma solução que utiliza **aprendizado de máquina** e **regras lógicas** para sugerir uma **ordem de execução** de casos de teste que maximize eficiência, minimize reinicializações e respeite dependências e pré/pós-condições. O sistema foi desenvolvido em contexto acadêmico para produção de artigo científico sobre **ordenação de casos de teste**.

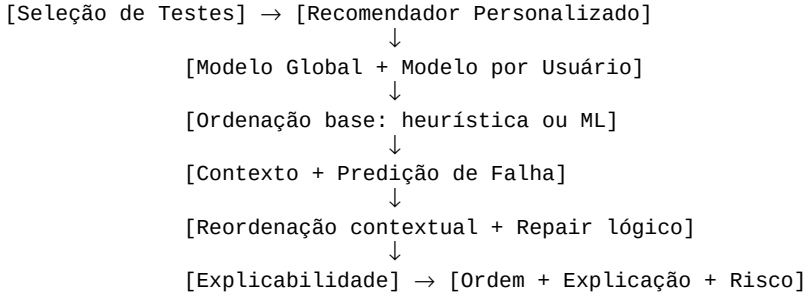
Este documento descreve de forma técnica: (i) os **modelos de ML** utilizados, (ii) o **funcionamento** do pipeline de recomendação, (iii) a forma como a **ordenação** é elaborada (incluindo heurísticas, contexto e predição de falha), e (iv) os mecanismos de **explicabilidade** e **auditoria** da sugestão.

2. Problema Abordado

- **Entrada:** conjunto de casos de teste (selecionados pelo usuário ou toda a suíte), cada um com ações, pré-condições, pós-condições, dependências explícitas, prioridade, módulo e tempo estimado.
- **Saída:** uma **ordem total** dos testes que:
- Respeite **dependências** (explícitas e inferidas por pré/pós-condições).
- Minimize **reinicializações** (quando o estado atual não satisfaz as pré-condições do próximo teste).
- Priorize **risco de falha** e **contexto** (hora, dia, histórico do usuário).
- Seja **explicável** e **auditável**.

O sistema combina **regras determinísticas** (grafo de dependências, topological sort, repair) com **modelos de ML** treinados a partir de **feedback humano** (sucesso/falha, tempo real, rating, necessidade de reset).

3. Arquitetura Geral do Sistema



- **Recomendador em produção:** `PersonalizedMLRecommender`, que utiliza internamente `MLTestRecommender` (global e por usuário).
- **Modelos de ML base:** Random Forest ou Gradient Boosting (regressão para *score* de ordenação).
- **Pós-processamento:** recomendações contextuais, predição de falha (Bayesiano), reordenação por dependências e **repair** final para garantir consistência lógica.

4. Modelo de Dados

4.1 Caso de Teste (TestCase)

- **Identificador**, nome, descrição, **prioridade** (1–5), **módulo**, **tags**.
- **Ações** (`Action`): tipo (criação, verificação, modificação, deleção, navegação), **impacto** (não destrutivo, parcialmente destrutivo, destrutivo), **pré-condições** e **pós-condições** (conjuntos de estados), tempo estimado.
- **Dependências explícitas:** conjunto de IDs de outros testes que devem ser executados antes.
- **Metadados de execução:** tempo médio, taxa de sucesso, última execução, número de execuções.

Pré e pós-condições são agregadas por ação: um teste “depende” logicamente de um estado se alguma de suas ações exige uma pré-condição que outro teste (na seleção) produz como pós-condição.

4.2 Feedback de Execução (ExecutionFeedback)

- test_case_id, data/hora, tempo real, **sucesso**, **seguiu recomendação**, **rating (1–5)**, **necessidade de reset**, notas, estados inicial/final.

Esse feedback é a base do **aprendizado supervisionado**: cada execução gera uma amostra (features da ordem, score calculado) para treinar o modelo.

5. Modelos de Aprendizado de Máquina

5.1 Recomendador Base (MLTestRecommender)

- **Objetivo:** prever um **score de qualidade** para uma **ordenação completa** (não para um teste isolado).
- **Algoritmos disponíveis:**
- **Random Forest Regressor:** 100 árvores, profundidade máxima 10, `random_state=42`.
- **Gradient Boosting Regressor:** 100 estimadores, profundidade 5, learning rate 0,1.
- **Pré-processamento:** `StandardScaler` nas features (normalização).
- **Saída:** valor contínuo (quanto maior, melhor a ordem). Esse score é usado para comparar ordens e em busca gulosa (troca de adjacentes).

O modelo **não** classifica testes individualmente; ele avalia **sequências**. As amostras de treinamento são geradas a partir de ordens já executadas e do feedback associado.

5.2 Recomendador Personalizado (PersonalizedMLRecommender)

• Mantém um **modelo global** (compartilhado) e **um modelo por usuário** (carregado/gravado no banco).

• **Peso de personalização** em função do **nível de experiência** (cadastro):

• Iniciante: 20% personalizado, 80% global.

• Intermediário: 50% / 50%.

• Avançado: 70% personalizado, 30% global.

• Expert: 85% personalizado, 15% global.

• **Combinação das ordens:** ensemble por *score* de posição: para cada teste, calcula posição normalizada na ordem global e na ordem personalizada; score combinado = `weight_personal *score_personal + (1 - weight_personal)* score_global`; ordenação final pelo score combinado (maior primeiro).

• Se o modelo do usuário não estiver treinado ou tiver poucos dados (< 5 amostras), usa apenas o modelo global.

5.3 Ensemble Recommender (opcional no código)

• Combina **Random Forest**, **Gradient Boosting** e opcionalmente **MLP (Rede Neural)** via `VotingRegressor`.

• Pesos típicos: 0,4 / 0,4 / 0,2 (ou 0,5 / 0,5 sem MLP).

• Usado para experimentos; em produção o fluxo principal usa `PersonalizedMLRecommender` com modelo base RF ou GB.

6. Extração de Features

6.1 FeatureExtractor

• **Features por teste (individuais):** prioridade, número de ações, tempo estimado, taxa de sucesso, vezes executado, contagens por tipo de ação (destrutivas, não destrutivas, criação, verificação, modificação, deleção, navegação), número de pré/pós-condições, razão de mudança de estado, número de dependências, flag de destrutivo, tempo médio por ação, dias desde última execução.

• **Features pareadas (para par de testes):** compatibilidade de estado (quanto as pós-condições do primeiro satisfazem as pré-condições do segundo), diferença de prioridade, mesmo módulo, sobreposição de tags, penalidade de ordem ruim (destrutivo → não destrutivo), diferença de tempo, se o segundo depende do primeiro.

6.2 Features para Score de Ordenação (treinamento do ML)

As amostras de treinamento usam **features agregadas da sequência**:

• Número de testes, tempo total, prioridade média, número de testes destrutivos.

• Número de **transições compatíveis** (pós do teste i intersecta pré do teste i+1).

• Número de **transições mesmo módulo**.

Ou seja, o modelo aprende a mapear **estatísticas da ordem** (não a ordem em si) para um **score único**.

7. Cálculo do Score de Qualidade (Função Objetivo)

O score usado como *target* no treinamento é calculado deterministicamente:

- **Base:** 100.
- **Penalidades:** quebra de dependência (-20 por quebra); teste destrutivo seguido de não destrutivo no mesmo módulo (-5); reset necessário no feedback (-15); falha no feedback (-10).
- **Bônus:** transição compatível de estado (+10 × proporção); mesmo módulo consecutivo (+3); sucesso (+10); seguiu recomendação (+5); rating (1–5) contribui com `(rating - 3) * 5`.

O score final é limitado a ≥ 0. Esse mesmo critério pode ser usado para avaliar ordens quando o modelo não está treinado (heurística pura).

8. Elaboração da Ordem: Heurística e ML

8.1 Quando o modelo não está treinado (ou use_heuristics=True)

Ordenação heurística:

Resolver **dependências**: em cada passo, considerar apenas testes cujas dependências já foram "executadas" na ordem atual; se nenhum for elegível (ciclo), liberar todos.

Entre os **executáveis**, ordenar por: prioridade (alta primeiro), módulo, não destrutivo antes de destrutivo, tempo estimado (menor primeiro).

Inserir o primeiro da lista e repetir até esgotar os testes.

Assim, a ordem inicial é **sempre viável** em termos de dependências explícitas.

8.2 Quando o modelo está treinado

- **Ordenação base:** mesma heurística acima.
- **Refinamento por ML:** busca local por **troca de adjacentes**. Para cada par (i, i+1), troca e avalia o score previsto pelo modelo; se melhorar, mantém a troca. Isso preserva a estrutura geral da heurística e ajusta a ordem com base no aprendizado.

No **PersonalizedMLRecommender**, a ordem pode ser resultado do **ensemble** entre ordem global e ordem personalizada (por scores de posição), e depois passar pelo pós-processamento descrito abaixo.

9. Recomendações Contextuais e Predição de Falha

9.1 Contexto

• **Time bucket:** manhã (5–11h), tarde (12–17h), noite (18–22h), madrugada.

• **Dia da semana / fim de semana.**

• **Último teste e último módulo** executados pelo usuário (do último feedback).

• **Afinidade por módulo e bucket:** desempenho histórico do usuário (sucesso) por (módulo, bucket), com suavização Laplace; módulo do último teste recebe bônus de afinidade.

9.2 Predição de Falha (Risco)

• **Modelo:** probabilidade de falha com **prior Beta** (Bayesiano):

• `P(falha) ≈ (falhas + α) / (total + α + β)`, com α = prior_fail *strength,* β = *(1 - prior_fail)* strength (ex.: prior_fail=0,2, strength=8).

• **Dados:** estatísticas por teste (execuções, falhas) **do usuário** e **globais**.

• **Fusão:** se o usuário tem dados, usa peso do usuário que cresce com o número de amostras (até ~85%); caso contrário, usa global ou prior.

• **Prior quando não há histórico:** ajustado por heurística (ex.: +0,05 se destrutivo, +0,03 se prioridade ≥ 4).

O resultado é um **risk_map** (test_id → probabilidade em [0,1]) usado na reordenação para **priorizar testes de maior risco** (executá-los mais cedo, quando o contexto é estável).

9.3 Reordenação Contextual (_contextual_reorder)

• **Entrada:** lista de testes, ordem base (IDs), risk_map, affinity_map, módulo inicial (último executado).

• **Dependências inferidas:** para cada pré-condição de um teste, se algum outro teste na seleção produz essa condição (pós-condição), cria-se uma dependência; em caso de múltiplos “provedores”, escolhe-se **um** (o de menor índice na ordem base) para evitar ciclos e super-restrições.

• **Algoritmo:** em cada passo, considera apenas testes cujas dependências (explícitas + inferidas) estão satisfeitas; simula estado (conjunto de pós-condições já produzidas); destrutivos zeram o estado. Entre os candidatos **executáveis**, ordena por:

• risco (maior primeiro),

• afinidade + bônus mesmo módulo,

• prioridade,

• compatibilidade com estado atual,

• penalidade por pré-condições internas não satisfeitas,

• módulo, não destrutivo antes de destrutivo, posição na ordem base, tempo estimado.

• **Saída:** nova ordem (lista de TestCase) que respeita dependências e incorpora risco e contexto.

10. Repair Lógico (_repair_order_for_logic)

Para garantir que a **ordem final** seja **sempre logicamente consistente** (sem violações de dependências ou pré-condições internas), aplica-se um “repair” após a reordenação contextual:

• Constrói um **grafo de dependências**: arestas (A, B) se B depende de A (dependências explícitas e, opcionalmente, inferidas por pré-condições; para inferidas, escolhe-se um provedor por pré-condição para evitar ciclos).

• **Ordenação topológica estável (Kahn):** ordem de execução respeitando o grafo; desempate pela posição na ordem atual.

• Se, ao incluir dependências inferidas, surgir ciclo (topo sort não retorna todos os nós), repete-se usando **apenas** dependências explícitas.

• **Saída:** lista de IDs na ordem reparada; tempo estimado e número de resets são recalculados sobre essa ordem.

Assim, a IA **não apenas sugere** uma ordem, mas **corrige** possíveis inconsistências antes de devolver ao usuário.

11. Explicabilidade da Recomendação

11.1 RecommendationExplainer

Quando o **modelo global** está treinado, usa-se o **RecommendationExplainer** com o mesmo modelo (Random Forest) e o **FeatureExtractor**:

• **Importância de features:** usa `feature_importances_` do Random Forest (ou permutation importance) para as 6 features de ordem (num_tests, total_time, avg_priority, num_destructive, compatible_transitions, same_module_transitions).

• **Fatores analisados:** agrupamento por módulo, compatibilidade de estados (transições compatíveis), priorização (altos no início), minimização de impacto destrutivo (destrutivos no final), tempo total estimado.

• **Scores por teste:** para cada teste na ordem, calcula um “score explicativo” (bonificação por prioridade no início, mesmo módulo, estado compatível com o anterior; penalização por destrutivo cedo) e lista de razões textuais.

• **Comparação com alternativas:** se ordens alternativas forem fornecidas, compara tempo total e número de resets.

• **Explicação textual:** gera frases em linguagem natural resumindo os fatores e a estrutura da ordem.

11.2 Quando o modelo não está treinado

A aplicação usa uma função **heurística** (`_generate_basic_explanation`) que analisa a ordem recomendada e produz fatores e reasoning similares (módulo, prioridade, destrutivos, tempo), sem importância de features do ML.

12. Detecção de Anomalias

O **AnomalyDetector** usa **Isolation Forest** (scikit-learn) sobre features extraídas dos **feedbacks** (tempo real, rating, sucesso, reset, seguiu recomendação; opcionalmente prioridade e tempo estimado do teste).

Objetivo: identificar execuções atípicas.

- **Padrões detectados:** alta taxa de falha por teste, degradação de desempenho ao longo do tempo, alta variabilidade de tempo (coeficiente de variação).
- **Alertas:** taxa de anomalias alta, testes que falham frequentemente, degradação de performance, múltiplas anomalias recentes.

Isso pode ser usado para **auditoria** (sinalizar períodos ou testes problemáticos) e, em extensões futuras, para ponderar o peso do feedback no treinamento.

13. Ciclo de Aprendizado e Feedback

Usuário **recebe** uma ordem recomendada (com explicação, risco por teste e contexto).

Usuário **executa** (total ou parcialmente) e envia **feedback** por teste: sucesso, tempo real, rating, se precisou de reset, se seguiu a ordem.

O sistema **adiciona** o feedback ao histórico e monta a **ordem efetivamente executada**.

Score da ordem é calculado com a função objetivo (incluindo feedback).

Amostra de treinamento (features agregadas da ordem + score) é adicionada aos dados do modelo **global** e do modelo **do usuário**.

Retreinamento: a cada 10 novos feedbacks (global) ou quando o usuário atinge 5+ amostras (modelo do usuário), os modelos são retreinados e, quando aplicável, o modelo do usuário é salvo no banco.

Assim, a IA **melhora com o uso**: quanto mais feedbacks, melhor a qualidade da ordenação sugerida e da explicabilidade (quando baseada no modelo treinado).

14. Dados e Reprodutibilidade

- **Banco:** SQLite (usuários, feedbacks, recomendações, modelos por usuário, estatísticas).
- **Modelos:** serialização em pickle (modelo global em arquivo; modelos por usuário no banco).
- **Random_state:** 42 nos estimadores scikit-learn para reprodutibilidade.
- **Versão:** o documento e o código servem como especificação; para o artigo, recomenda-se registrar versão do código, schema do banco e hiperparâmetros (n_estimators, max_depth, prior Beta, etc.) em anexo ou repositório.

15. Resumo para Redação do Artigo

- **Modelo:** regressão (Random Forest / Gradient Boosting) para **score de ordenação**; combinação global + personalizada por usuário com peso por nível de experiência.
- **Features:** agregadas por sequência (tempo, prioridade, transições compatíveis, mesmo módulo, destrutivos).

- **Objetivo:** maximizar score que combina respeito a dependências, minimização de resets, compatibilidade de estado, prioridade e feedback (sucesso, rating, reset).
- **Ordem:** heurística (topological + prioridade/módulo/destrutivo/tempo) quando não há modelo; refinamento por ML (swap de adjacentes) e ensemble global/personalizado quando há modelo; em seguida reordenação contextual (risco + contexto) e **repair** por topological sort para garantir consistência lógica.
- **Predição de falha:** Beta Bayesiano sobre histórico de execuções (usuário e global).
- **Explicabilidade:** importância de features, fatores qualitativos e scores por teste; explicação textual para auditoria.

Este documento cobre o funcionamento completo da IA de ordenação do IARTES e pode ser usado como base para a seção de metodologia e implementação do artigo acadêmico.