Pokenator: um jogo de adivinhação da Web Semântica para Pokémon

Rávilon Aguiar dos Santos[1], Gustavo Pìnzon Pereira[2]

[1]Universidade Federal de Pelotas (UFPEL), Brasil
[2]Universidade Federal de Pelotas (UFPEL), Brasil

Abstract

A Web Semântica concebe uma Web de dados que é interpretável por máquinas, permitindo que aplicações realizem raciocínio, integração e descoberta sobre fontes heterogêneas. Embora poderosa, seus conceitos podem ser difíceis de compreender para aprendizes não familiarizados com RDF, OWL e SPARQL. Neste artigo apresentamos o *Pokenator*, um jogo interativo de adivinhação que aplica tecnologias da Web Semântica a um domínio lúdico – o universo Pokémon – para servir como estudo de caso didático. Inspirado pelo popular jogo Akinator, o Pokenator faz ao jogador uma série de perguntas de sim/não sobre um Pokémon oculto e reduz os candidatos usando consultas SPARQL sobre um grafo de conhecimento baseado em ontologia. Descrevemos as fontes de dados utilizadas, incluindo a Ontologia Pokémon [1] e metadados adicionais da PokéAPI, delineamos a arquitetura do nosso sistema e suas heurísticas de raciocínio, e discutimos como tal aplicação pode apoiar o ensino de conceitos da Web Semântica.

Keywords

Web Semântica, Ontologia, Grafo de Conhecimento, SPARQL, Pokémon, Akinator

1. Introdução

A Web Semântica fornece um arcabouço para publicar dados estruturados na Web de forma que possam ser consultados e processados por máquinas. Grafos RDF capturam fatos como triplas de sujeito–predicado–objeto; ontologias definem classes e propriedades para organizar esses fatos; e a linguagem de consulta SPARQL permite correspondência de padrões expressivos sobre grafos. Conforme explica a especificação W3C, o SPARQL pode expressar consultas em diversos conjuntos de dados RDF e suporta padrões de grafo obrigatórios e opcionais, bem como restrições de valores [2]. Apesar de sua maturidade, a Web Semântica permanece abstrata para muitos aprendizes. Buscamos criar uma demonstração prática que ancorasse os conceitos da Web Semântica em um domínio familiar e fornecesse um feedback intuitivo.

O clássico jogo online *Akinator* convida os jogadores a pensar em um personagem real ou fictício; o sistema então apresenta uma sequência de perguntas e tenta adivinhar o personagem [3]. Nosso projeto *Pokenator* adapta essa ideia ao universo Pokémon, utilizando tecnologias da Web Semântica para conduzir o processo de raciocínio. Os Pokémon são amplamente reconhecidos pelos estudantes e fornecem atributos ricos e estruturados (tipos, habilidades, habitats, gerações) que se prestam ao modelamento ontológico. O objetivo do Pokenator é tanto entreter quanto expor os aprendizes a RDF, OWL e SPARQL em um contexto didático.

2. Web Semântica e ontologias

2.1. RDF e SPARQL

O Resource Description Framework (RDF) representa informações como grafos direcionados e rotulados de triplas; cada tripla é composta por um sujeito, um predicado e um objeto. SPARQL é a linguagem padrão de consulta para grafos RDF e permite aos usuários especificar padrões de grafo que devem

Workshop CEUR-WS sobre Aplicações da Web Semântica,
Fevereiro de 2026, Porto Alegre, Brasil

corresponder aos dados[2]. A especificação enfatiza que o SPARQL pode consultar dados armazenados nativamente como RDF ou vistos como RDF via middleware, e suporta padrões opcionais, conjunções e disjunções, bem como conjuntos de resultados ou construções de grafo[2]. Essas características sustentam a capacidade do Pokenator de filtrar Pokémon candidatos com base nas respostas do usuário e de construir explicações para seus palpites.

2.2. Ontologias

Ontologias fornecem o vocabulário e a semântica para um domínio. No Pokenator utilizamos a Ontologia Pokémon (Ontologia PokémonKG) desenvolvida por Kevin Haller [1]. Essa ontologia define classes como `pok:Pokemon`, `pok:Type`, `pok:Ability`, `pok:Habitat` e `pok:Generation`, bem como propriedades incluindo `pok:hasType`, `pok:hasAbility` e `pok:fromGeneration`. Ela foi lançada em 27 de julho de 2019 e a versão atual (v1.0.0) está publicamente disponível [1]. Ontologias permitem raciocínio: por exemplo, se um Pokémon possui dois tipos, Fogo e Voador, a ontologia pode inferir que ele é um Pokémon de tipo duplo.

3. Fontes de dados

3.1. PokémonKG e a Ontologia Pokémon

O Grafo de Conhecimento Pokémon (PokemonKG) agrega informações de fontes comunitárias como Bulbagarden, PokeWiki, PokéAPI e PokemonDB. O grafo de conhecimento contém aproximadamente 129.834 triplas e 18.568 entidades, e é publicado sob uma licença aberta com um endpoint SPARQL. Usamos a Ontologia Pokémon como vocabulário para o nosso grafo de conhecimento e carregamos as triplas em um triplestore (Apache Jena Fuseki). Um raciocinador com capacidades RDFS/OWL permite subsunção de classes e herança de propriedades.

3.2. PokéAPI

Embora a Ontologia Pokémon forneça um esquema semântico, nem todos os atributos estão disponíveis como RDF. Complementamos a ontologia com dados detalhados de instância da PokéAPI, um serviço REST aberto que expõe dados abrangentes de Pokémon. A API se autodenomina "todos os dados de Pokémon de que você precisará em um só lugar", atendendo a mais de 10 bilhões de chamadas por mês e oferecendo uma interface moderna, gratuita e de código aberto [4]. Inclui endpoints para espécies de Pokémon, tipos, habilidades e movimentos. Integramos respostas JSON selecionadas em nosso grafo de conhecimento, mapeando-as para classes e propriedades da ontologia. Como a PokéAPI é um projeto comunitário em constante evolução [4], fazemos um snapshot dos dados relevantes para o nosso grafo de conhecimento para garantir reprodutibilidade.

3.3. Referencial inspirador: Akinator

Nosso design é inspirado no jogo Akinator desenvolvido pela Elokence. No site oficial, o gênio convida o jogador a "pensar em um personagem real ou fictício" e promete adivinhá-lo [3]. Adotamos um fluxo interativo semelhante: o Pokenator pede ao jogador que imagine um Pokémon e então faz perguntas binárias (por exemplo, "Ele é do tipo Fogo?", "Ele pertence à primeira geração?"). Cada resposta é traduzida em um filtro SPARQL que reduz o conjunto de candidatos. Uma heurística escolhe a próxima pergunta para maximizar o ganho de informação, semelhante a uma árvore de decisão. Quando resta um único candidato, o sistema apresenta seu palpite junto com uma explicação das triplas que sustentam a inferência.

3.4. Repositório do projeto

O código-fonte completo do Pokenator, incluindo o back-end, o front-end e os scripts de preparação de dados, está disponível publicamente no GitHub em https://github.com/ravilon/Pokenator. O repositório abriga a implementação Java da engine de raciocínio (construída sobre Apache Jena) juntamente com um back-end Spring Boot que expõe a API do jogo. O front-end é implementado com React e Vite e utiliza Tailwind CSS para estilização. Todo o código é de código aberto sob uma licença permissiva. Incentivamos os leitores a explorar o projeto e contribuir com melhorias.

4. Arquitetura do sistema

A Figura 1 ilustra a arquitetura de alto nível do Pokenator. O front-end, construído com React e o empacotador Vite e estilizado com Tailwind CSS, fornece a interface de usuário para fazer perguntas e mostrar palpites. O back-end é uma aplicação Spring Boot escrita em Java que converte as respostas do usuário em consultas SPARQL e interage com o triplestore. Dentro do back-end, um módulo de raciocínio implementa heurísticas para selecionar a próxima pergunta: ele analisa os candidatos restantes e escolhe uma propriedade cujos valores estejam mais equilibrados entre os candidatos. O grafo de conhecimento reside em um servidor Apache Jena Fuseki e armazena a ontologia e os dados de instância. Opcionalmente, os resultados podem ser enriquecidos com dados recuperados sob demanda da PokéAPI.

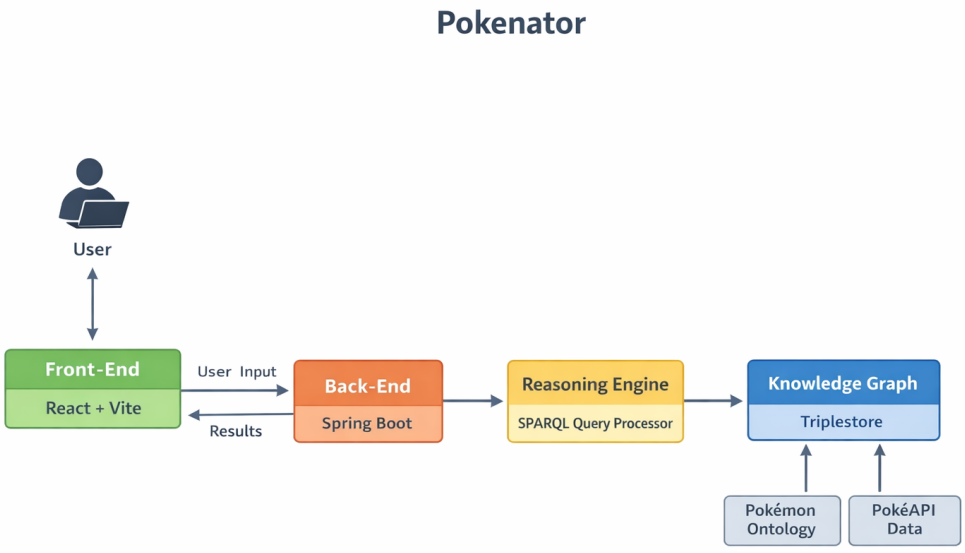

Figure 1: Visão geral da arquitetura do Pokenator. As respostas do usuário na interface web são traduzidas em consultas SPARQL pela engine de raciocínio. As consultas são executadas sobre um triplestore populado com a Ontologia Pokémon e dados de instância; os resultados são usados para atualizar o conjunto de candidatos e determinar a próxima pergunta.

5. Implementação

5.1. Preparação dos dados

Primeiro carregamos a Ontologia Pokémon no Apache Jena Fuseki. Em seguida utilizamos uma ferramenta Java construída sobre o Jena para buscar dados JSON da PokéAPI e converter campos relevantes

(tipo, habilidade, habitat, geração e estatísticas base) em triplas RDF consistentes com a ontologia. Cada instância de Pokémon recebeu um IRI persistente (por exemplo, `poke:Pikachu`). Quando atributos estavam ausentes na ontologia, nós a estendemos adicionando classes e propriedades em um módulo separado para evitar alterar a especificação original.

5.2. Seleção heurística de perguntas e motor de busca

A engine de raciocínio conduz o jogo de adivinhação mantendo um **conjunto de candidatos** com todos os Pokémon consistentes com as respostas do jogador. Esse conjunto é armazenado em memória como uma lista de IRIs que inicialmente contém todos os Pokémon no grafo de conhecimento. Toda vez que o jogador responde a uma pergunta de sim/não, a engine emite uma consulta SPARQL correspondente para filtrar os candidatos. Por exemplo, se o jogador responde "sim" para "Ele é do tipo Fogo?", a engine envia uma consulta que exige que a variável `?p` tenha `pok:hasType` igual a `pok:Fire`. Uma resposta "não" resulta em uma consulta que exclui os tipos Fogo. As respostas são combinadas de forma conjuntiva para refinar o conjunto de candidatos após cada etapa.

Para decidir qual pergunta fazer a seguir, a engine enumera perguntas candidatas com base em um pequeno conjunto de famílias de predicados. Essas incluem propriedades diretas como `pok:hasType`, `pok:hasColour`, `pok:hasShape` e `pok:foundIn`, bem como a geração, que é alcançada via o predicado inverso `pok:featuresSpecies` entre os recursos de geração e as espécies. Para cada família de predicados, a engine consulta o triplestore para contar quantos dos candidatos restantes possuem cada valor possível. Valores já restringidos por respostas anteriores ou já perguntados na sessão atual são ignorados. Apenas valores cujas contagens sejam estritamente entre 0 e o número total de candidatos restantes são considerados, pois podem particionar o conjunto.

Para cada pergunta potencial, a engine registra o número de candidatos que responderiam "sim" (isto é, aqueles que possuem o valor) e "não". Em seguida, ordena as perguntas candidatas pela proximidade dessa contagem "sim" à metade dos candidatos restantes, porque uma divisão equilibrada gera o máximo de informação. Entre as poucas perguntas com a melhor divisão, uma é selecionada aleatoriamente para evitar comportamento determinístico. Uma vez selecionada, a pergunta é marcada como feita na sessão para não ser repetida.

Se o conjunto de candidatos se tornar pequeno (dois elementos ou menos) ou o seletor de perguntas não encontrar um par predicado–valor adequado, a engine muda para o modo de palpite. Ela pergunta "É o [Pokémon]?" para uma das espécies restantes e, se necessário, oferece o outro candidato caso o usuário responda negativamente. Essas heurísticas garantem que o jogo converge rapidamente, ao mesmo tempo em que preserva um grau de variabilidade entre sessões. O back-end é implementado em Java usando Apache Jena para consultas SPARQL e exposto como um serviço Spring Boot.

5.3. Construção dinâmica de consultas e complexidade

Toda vez que o jogador responde a uma pergunta, o Pokenator atualiza seu conjunto de candidatos construindo uma nova consulta SPARQL. A consulta começa com um padrão básico que seleciona todas as instâncias de Pokémon (‘?p‘) da classe `pok:Pokemon`. Para cada resposta afirmativa, adicionamos um padrão de tripla adicional da forma ‘?p <predicado> <valor>‘ para exigir esse valor; para respostas negativas adicionamos uma cláusula `FILTER NOT EXISTS` para excluir essa tripla. Como as consultas SPARQL são expressas em texto, o servidor monta essas cláusulas dinamicamente concatenando strings de modelo. A consulta resultante fica assim (simplificada):

```
SELECT ?p WHERE {
  ?p a pok:Pokemon .
  # restrições positivas
  ?p pok:hasType pok:Fire .
  ?p pok:fromGeneration pok:Generation_V .
  # restrições negativas
  FILTER NOT EXISTS { ?p pok:hasAbility pok:Levitate . }
```

}

A engine executa essa consulta no triplestore para obter a lista de candidatos restantes. Como cada restrição reduz o espaço de busca, o custo de avaliar a consulta tipicamente diminui com o tempo. No pior caso, porém, a consulta contém uma conjunção de m padrões positivos e negativos e roda contra um grafo de conhecimento com n Pokémon. A avaliação de consultas em SPARQL é NP-completa em geral, mas com nossos padrões acíclicos simples a complexidade efetiva é aproximadamente $O(n \times m)$, pois o store verifica cada candidato contra cada restrição.

Ao selecionar a próxima pergunta, a engine emite consultas de agregação da forma 'SELECT <valor> (COUNT(DISTINCT ?p) AS ?count) WHERE ... GROUP BY <valor>' para contar quantos dos candidatos atuais exibem cada valor para uma dada família de predicados. Como o número de famílias de predicados é pequeno (cerca de cinco) e cada consulta de agregação examina apenas o conjunto de candidatos reduzido, a sobrecarga permanece gerenciável mesmo em um grafo com dezenas de milhares de triplas. A randomização entre os principais candidatos mitiga ainda mais o risco de comportamento determinístico, ao mesmo tempo em que favorece perguntas que maximizem o ganho de informação.

5.4. Geração de explicações

Para tornar o processo de raciocínio transparente, o Pokenator retorna uma explicação para seu palpite. A consulta SPARQL que levou ao candidato final é preservada, e as triplas que correspondem à consulta (tipos, habilidades, geração) são apresentadas ao jogador. Por exemplo, se o Pokémon adivinhado for *Charizard*, o sistema pode listar que ele é do tipo Fogo/Voador e originário da Geração I. Como a ontologia inclui semântica RDFS/OWL, as explicações podem incluir fatos inferidos (por exemplo, pertencer à classe *DualType*).

6. Interface do usuário e jogabilidade

A aplicação web Pokenator apresenta uma interface simples e intuitiva para os jogadores. O front-end faz uma série de perguntas de sim/não sobre o Pokémon oculto e exibe o conjunto atual de candidatos possíveis. À medida que o jogador responde, a lista de candidatos se reduz em tempo real. Quando resta um candidato único, o sistema revela seu palpite e explica o raciocínio listando os atributos relevantes. Esse fluxo interativo ajuda os usuários a compreender como o grafo de conhecimento subjacente e as heurísticas de raciocínio operam.

A Figura 2 resume o fluxo de interação. O jogador primeiro pensa em um Pokémon oculto. O Pokenator então faz repetidamente perguntas de sim/não; após cada resposta, a engine de raciocínio atualiza seu conjunto de candidatos por meio de uma consulta SPARQL e decide qual pergunta fazer a seguir. Quando resta apenas um candidato, o sistema faz um palpite e explica seu raciocínio.

A Figura

7. Avaliação e discussão

Nosso objetivo principal foi criar um demonstrador para o ensino de conceitos da Web Semântica em vez de competir com a eficiência de algoritmos gerais de adivinhação. Entretanto, avaliamos o Pokenator com Pokémon selecionados e descobrimos que o sistema normalmente converge para um candidato único em 5–7 perguntas quando a ontologia contém atributos discriminatórios suficientes. Em casos em que os atributos estão ausentes ou são ambíguos (por exemplo, Pokémon que compartilham tipos e habilidades), o sistema pode necessitar de mais perguntas ou não conseguir produzir um palpite único.

A integração dos dados da PokéAPI melhorou a cobertura, particularmente para habilidades e listas de movimentos. No entanto, a heterogeneidade das fontes de dados exige um mapeamento cuidadoso para a ontologia. Trabalhos futuros poderiam explorar a ligação do Pokenator a conjuntos de dados adicionais (como locais de Pokémon ou cadeias de evolução) e a implementação de regras de raciocínio mais sofisticadas.

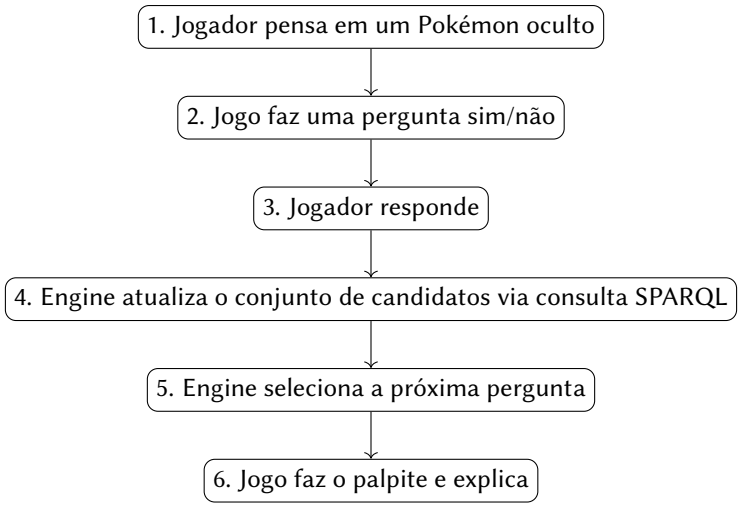

Figure 2: Fluxo de jogabilidade do Pokenator. A partir de um Pokémon oculto escolhido pelo jogador, o jogo faz perguntas binárias, atualiza seu conjunto de candidatos usando consultas SPARQL e uma engine de raciocínio, seleciona novas perguntas com base em heurísticas e, por fim, faz um palpite com explicação.

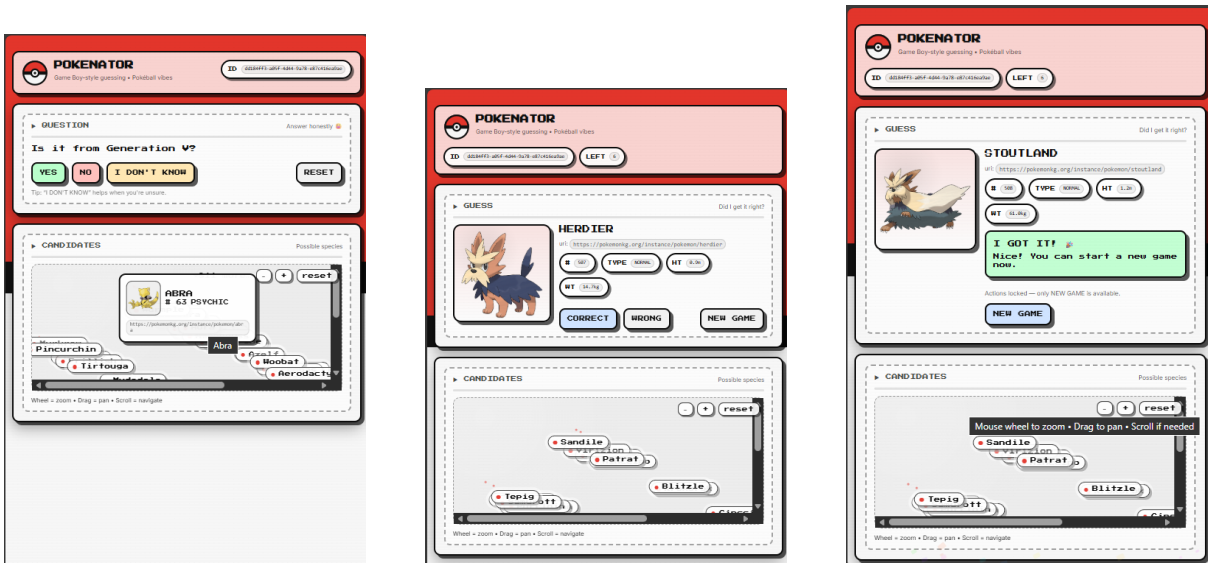

Figure 3: Capturas de tela da interface do usuário do Pokenator. Esquerda: o sistema pergunta se o Pokémon é da Geração V. Centro: a lista de candidatos restantes após responder a várias perguntas. Direita: o palpite final com uma explicação dos atributos inferidos.

8. Conclusão

O Pokenator demonstra como tecnologias da Web Semântica podem sustentar uma aplicação educacional envolvente. Ao combinar um grafo de conhecimento orientado por ontologia com raciocínio via SPARQL e uma heurística simples, o sistema convida os usuários a explorar dados estruturados de Pokémon enquanto entram em contato com conceitos-chave como triplas RDF, ontologias, consultas e inferência. O projeto também evidencia a riqueza de recursos abertamente disponíveis como a Ontologia Pokémon e a PokéAPI [1, 4]. Esperamos que o Pokenator inspire os aprendizes a experimentar ferramentas da Web Semântica e considerar como a representação do conhecimento e o raciocínio podem apoiar aplicações web interativas.

References

[1] K. Haller, Pokémon ontology, https://pokemonkg.org/ontology/version/1.0.0, 2019. Revision: v1.0.0.

[2] W. W. W. Consortium, Sparql 1.1 query language, https://www.w3.org/TR/sparql11-query/, 2013. W3C Recommendation.

[3] Elokence, Akinator: The web genie, https://en.akinator.com/, 2007. Online guessing game that inspired this work.

[4] P. Hallett, P. contributors, Pokéapi: The restful pokémon api, https://pokeapi.co/, 2026. Accessed 22 Feb 2026.