

analise_de_hiperparametro

June 11, 2024

1 Busca de Índice e Consulta de documento

Autor: Davi J. Leite Santos

Versão: 0.0.3

Data: 25 de Abril de 2024

Localização: Ribeirão das Neves, Minas Gerais - Brasil

1.1 Contato

- **Endereço:** Ribeirão das Neves, Minas Gerais - Brasil
- **Email:** davi.jls@outlook.com
- **LinkedIn:** davi-j-leite-santos
- **Website:** davijls.com.br

1.2 Principais Competências

- Cibersegurança
- Segurança da Informação
- Operações de TI

2 Sobre o código

Este código Python apresenta um script robusto e detalhado desenvolvido para criar e analisar um índice de documentos com foco na otimização do desempenho da busca de termos. O processo de indexação de texto e recuperação de termos é fundamentado no uso de “stemming” e “chunking” (segmentação de texto). O código é parte de um projeto pessoal de Davi J. Leite Santos sobre cibersegurança e operações de TI, contendo informações técnicas sobre a manipulação de grandes conjuntos de dados de texto.

Detalhamento das Componentes Principais do Código ## 1. Importações e Configurações Iniciais O código inicia com a importação de módulos necessários como json para manipulação de dados JSON, re para expressões regulares, os para interações com o sistema operacional, time para medição de tempo e nltk, uma biblioteca de processamento de linguagem natural. Além disso, o código baixa os recursos do NLTK necessários para “stemming” e tokenização.

2.1 2. Definição de Funções

- `load_index`: Carrega um índice e um vocabulário previamente salvos em arquivos JSON.
- `create_chunks`: Divide um texto em segmentos de tamanho definido.

- `search_term`: Busca um termo no índice após convertê-lo para sua forma raiz (“stem”) usando o `RSLPStemmer` para o Português.
- `save_progress`: Salva o progresso de alguma operação em um arquivo de texto.
- `process_file`: Processa um arquivo de texto dividindo-o em chunks, tokenizando-o e atualizando o índice e o vocabulário com novas entradas.
- `create_inverted_index_with_chunks`: Cria um índice invertido com a possibilidade de configurar o tamanho dos chunks.
- `measure_search_time_with_different_chunk_sizes`: Mede o tempo de busca em um índice para avaliar o desempenho baseado em diferentes tamanhos de chunks.
- `index_documents_with_different_chunk_sizes`: Cria índices segmentados com base em diversos tamanhos de chunks para avaliar o desempenho de armazenamento e busca.
- `analyze_hyperparameters`: Executa a função de indexação com diferentes configurações de tamanho de chunk e documenta os resultados, incluindo o tempo médio de busca e o espaço de armazenamento usado. *## 3. Execução da Análise de Hiperparâmetros* O código então configura e executa a função `analyze_hyperparameters`, passando os diretórios de arquivos, termos de busca, tamanhos de chunks desejados, e os caminhos para arquivos de saída e progresso. Esta função é a principal executora que desencadeia todo o processo de análise de desempenho de indexação e busca.

3 Conclusão

Este código é uma ferramenta sofisticada e detalhada para testar e analisar o desempenho de sistemas de indexação de documentos, focando principalmente no impacto de diferentes estratégias de segmentação de dados. Ele integra várias práticas avançadas de engenharia de software, como modularização de código, reuso de funções e escrita eficiente e segura de dados. Ele exemplifica bem como scripts Python podem ser empregados em tarefas complexas de processamento de dados e análise de desempenho em campos como a cibersegurança e TI.

4

```
[ ]: import json
import re
import os
import time
import nltk
from nltk.stem import RSLPStemmer
```

```
[ ]: # Baixar os recursos necessários do NLTK
nltk.download("rslp")
nltk.download("punkt")

# Inicializar o Stemmer para português
stemmer = RSLPStemmer()
```

```
[nltk_data] Downloading package rslp to
[nltk_data] C:\Users\davim\AppData\Roaming\nltk_data...
[nltk_data] Package rslp is already up-to-date!
```

```
[nltk_data] Downloading package punkt to
[nltk_data] C:\Users\davim\AppData\Roaming\nltk_data...
[nltk_data] Package punkt is already up-to-date!
```

5 Funções de criação de chunk para a análise de granularidade

```
[ ]: # Função para carregar o índice e o vocabulário
def load_index(index_file, vocab_file):
    with open(index_file, "r", encoding="utf-8") as f:
        index = json.load(f)
    with open(vocab_file, "r", encoding="utf-8") as f:
        vocab = json.load(f)
    return index, vocab

[ ]: # Função para criar chunks de um documento
def create_chunks(text, chunk_size):
    words = text.split()
    return [
        " ".join(words[i : i + chunk_size]) for i in range(0, len(words),
↪chunk_size)
    ]

[ ]: # Função para buscar termos no índice
def search_term(index, vocab, term):
    stemmed_term = stemmer.stem(term)
    if stemmed_term in vocab:
        word_id = vocab[stemmed_term]
        if str(word_id) in index:
            return index[str(word_id)]
    return {}

[ ]: # Função para salvar o progresso em um arquivo
def save_progress(progress_file, progress):
    with open(progress_file, "w", encoding="utf-8") as f:
        f.write("Progress:\n")
        for key, value in progress.items():
            f.write(f"{key}: {value}\n")

[ ]: # Função para carregar o índice e o vocabulário
def load_index(index_file, vocab_file):
    with open(index_file, "r", encoding="utf-8") as f:
        index = json.load(f)
    with open(vocab_file, "r", encoding="utf-8") as f:
        vocab = json.load(f)
    return index, vocab
```

```
[ ]: # Função para simular a criação de arquivos de índice e vocabulário
def create_temp_files(index_file, vocab_file, chunk_size):
    # Criar dados fictícios para o índice e vocabulário
    index_data = {f"word{chunk_size}": [chunk_size, chunk_size + 1]}
    vocab_data = {f"word{chunk_size}": chunk_size}

    # Salvar dados fictícios nos arquivos
    with open(index_file, "w", encoding="utf-8") as f:
        json.dump(index_data, f)
    with open(vocab_file, "w", encoding="utf-8") as f:
        json.dump(vocab_data, f)
```

```
[ ]: # Função para processar arquivo com granularidade ajustável
def process_file(file_path, chunk_size, doc_id, vocab, index):
    if not os.path.exists(file_path):
        print(f"Arquivo não encontrado: {file_path}")
        return
    with open(file_path, "r", encoding="utf-8") as f:
        text = f.read()
    chunks = create_chunks(text, chunk_size)
    for chunk_id, chunk in enumerate(chunks):
        tokens = nltk.word_tokenize(chunk, language="portuguese")
        for pos, token in enumerate(tokens):
            stemmed_token = stemmer.stem(token)
            if stemmed_token not in vocab:
                vocab[stemmed_token] = len(vocab)
            word_id = vocab[stemmed_token]
            if word_id not in index:
                index[word_id] = {}
            if doc_id not in index[word_id]:
                index[word_id][doc_id] = []
            index[word_id][doc_id].append((chunk_id, pos))
```

```
[ ]: def create_inverted_index_with_chunks(files, chunk_size):
    index = {}
    vocab = {}
    vocab_id = 0
    doc_id = 0

    for file_path in files:
        with open(file_path, "r", encoding="utf-8") as f:
            words = f.read().split()

            chunk_id = 0
            while chunk_id * chunk_size < len(words):
                chunk = words[chunk_id * chunk_size : (chunk_id + 1) * chunk_size]
                for pos, word in enumerate(chunk):
```

```

        if word not in vocab:
            vocab[word] = vocab_id
            vocab_id += 1
        term_id = vocab[word]
        if term_id not in index:
            index[term_id] = {}
        if doc_id not in index[term_id]:
            index[term_id][doc_id] = []
        index[term_id][doc_id].append(pos + chunk_id * chunk_size)
        chunk_id += 1
        doc_id += 1

    return index, vocab

```

```

[ ]: # Função para simular a medição do tempo de busca com diferentes tamanhos de chunk
    ↪ chunk
def measure_search_time_with_different_chunk_sizes(index, vocab, search_terms):
    total_search_time = 0.0
    num_searches = len(search_terms)
    for term in search_terms:
        start_time = time.time()
        _ = index.get(term, [])
        total_search_time += time.time() - start_time

    avg_search_time = total_search_time / num_searches
    return avg_search_time

```

```

[ ]: # Função para simular a indexação de documentos com diferentes tamanhos de chunk
def index_documents_with_different_chunk_sizes(files_dir, chunk_sizes):
    index_file, vocab_file = files_dir
    index, vocab = load_index(index_file, vocab_file)

    results = []
    for chunk_size in chunk_sizes:
        chunked_index = {
            k: v[:chunk_size] if isinstance(v, list) else v for k, v in index.
            ↪ items()
        }
        chunked_vocab = {k: v for k, v in vocab.items() if len(k) <= chunk_size}

        index_size = len(json.dumps(chunked_index).encode("utf-8"))
        vocab_size = len(json.dumps(chunked_vocab).encode("utf-8"))
        total_size = index_size + vocab_size

        results.append(
            {
                "chunk_size": chunk_size,

```

```

        "index": chunked_index,
        "vocab": chunked_vocab,
        "index_size": index_size,
        "vocab_size": vocab_size,
        "total_size": total_size,
    }
)
return results

```

```

[ ]: # Função principal para realizar a análise de hiperparâmetros
def analyze_hyperparameters(
    files_dir, search_terms, chunk_sizes, output_file, progress_file
):
    results = index_documents_with_different_chunk_sizes(files_dir, chunk_sizes)

    with open(output_file, "w", encoding="utf-8") as f_output, open(
        progress_file, "w", encoding="utf-8"
    ) as f_progress:
        for result in results:
            avg_search_time = measure_search_time_with_different_chunk_sizes(
                result["index"], result["vocab"], search_terms
            )
            result["avg_search_time"] = avg_search_time

            f_output.write(f"Chunk Size: {result['chunk_size']}\n")
            f_output.write(f"Index Size: {result['index_size']} bytes\n")
            f_output.write(f"Vocab Size: {result['vocab_size']} bytes\n")
            f_output.write(f"Total Size: {result['total_size']} bytes\n")
            f_output.write(
                f"Average Search Time: {result['avg_search_time']:.6f}
↪segundos\n"
            )
            f_output.write("\n")

            f_progress.write(json.dumps(result) + "\n")

            f_output.flush()
            f_progress.flush()

```

```

[ ]: # Definir os parâmetros
files_dir = ("index_geral.json", "vocab_geral.json")
search_terms = ["educação", "linguagem", "notícia", "ideia", "politica"]
chunk_sizes = [50, 100, 200, 500] # Tamanhos de chunks para testar
output_file = "hyperparameter_analysis.txt"
progress_file = "hyperparametro_progress.txt"

```

```
[ ]: analyze_hyperparameters(  
    files_dir, search_terms, chunk_sizes, output_file, progress_file  
)
```

```
[ ]:
```

```
[ ]:
```