SBBD 2023

KDMiLe 2023

38º Simpósio Brasileiro de Bancos de Dados

38th Brazilian Symposium on Databases

Tópicos em Gerenciamento de Dados e Informações: Minicursos do SBBD 2023

Topics in Data Management and Information: Short courses of SBBD 2023

De 25 a 29 de setembro de 2023

Execução: IGCE Instituto de Ciências Exatas e Informática; PUC Minas; Instituto Federal Minas Gerais

Realização: SBC Sociedade Brasileira de Computação; Cebd SBC Comissão Especial de Banco de Dados

Apoio Acadêmico: UFMG; CEFET-MG

Patrocínio: alfa Engenharia; Cockroach Labs; sólides; stone tech; Google

Agências de fomento: CAPES; FAPEMIG

38º Simpósio Brasileiro de Bancos de Dados

38th Brazilian Symposium on Databases

Tópicos em Gerenciamento de Dados e Informações: Minicursos do SBBD 2023

Organizadores:
Humberto Luiz Razente
Ticiana L. Coelho da Silva
Michele Amaral Brandão
Felipe Domingos da Cunha

Porto Alegre

Sociedade Brasileira de Computação – SBC

2023

Execução:

Realização:

Apoio Acadêmico:

Patrocínio:

Agências de fomento:

Dados Internacionais de Catalogação na Publicação (CIP)

S612 Simpósio Brasileiro de Banco de Dados (38. : 25 – 29 set. 2023 : Belo Horizonte)
Minicursos do SBBD 2023 [recurso eletrônico] / organização: Humberto Luiz Razente ... [et al.]. Dados eletrônicos. – Porto Alegre: Sociedade Brasileira de Computação, 2023.
72 p. : il. : PDF ; 10MB

Modo de acesso: World Wide Web.
Inclui bibliografia
ISBN 978-85-7669-554-7 (e-book)

1. Computação – Brasil – Simpósio. 2. Banco de Dados. I. Razente, Humberto Luiz. II. Silva, Ticiana L. Coelho da. III. Brandão, Michele Amaral. IV. Cunha, Felipe Domingos da. V. Sociedade Brasileira de Computação. VI. Título.

CDU 004.6(063)

Ficha catalográfica elaborada por Annie Casali – CRB-10/2339

Biblioteca Digital da SBC – SBC OpenLib

Índices para catálogo sistemático:

1. Ciência e tecnologia dos computadores : Informática : Dados – Publicação de conferências, congressos e simpósios etc. ... 004.6(063)

Organização do SBBD 2023
SBBD 2023 Organization

Program Chair
Daniel Kaster (UEL, Brazil)

Short Papers Chair
Eduardo Ogasawara (CEFET/RJ, Brazil)

Demos and Applications Chair
Marcos Bedo (UFF, Brazil)

WTDBD Chair
Renata Galante (UFRGS, Brazil)

CTDBD Chair
Eduardo C. de Almeida (UFPR, Brazil)

Short Courses Chair
Humberto Razente (UFU, Brazil)

Tutorials Chair
Marcos André Gonçalves (UFMG, Brazil)

Workshops Chair
Elaine Sousa (USP, Brazil)

WTAG Chair
Marcelo Iury (UFPB, Brazil)

Execução: **Realização:** **Apoio Acadêmico:**

Patrocínio: **Agências de fomento:**

Proceedings Chair

Ticiana L. Coelho da Silva (UFC, Brazil)

Sponsorships and Institutional Contacts Chair

Wladmir Cardoso Brandão (PUC Minas, Brazil)

Organização Geral

General Organization

Chairs

Michele Amaral Brandão (IFMG, Brazil)

Felipe Domingos da Cunha (PUC Minas, Brazil)

Execução:

Realização:

Apoio Acadêmico:

Patrocínio:

Agências de fomento:

O presente livro do XXXVIII Simpósio Brasileiro de Bancos de Dados (SBBD 2023) inclui dois capítulos escritos pelos autores dos minicursos selecionados e apresentados na edição do evento realizado de 25 a 29 de setembro de 2023. Os minicursos têm como objetivo apresentar temas relevantes relacionados à área de Banco de Dados e promover discussões sobre os fundamentos, tendências e desafios dos temas abordados. Cada minicurso tem quatro horas de duração e constitui uma excelente oportunidade de atualização para acadêmicos e profissionais que participam do evento.

Os capítulos abordam conteúdos relacionados à interface de programação da OpenAI e a manipulação de dados geoespaciais. O comitê de programa de minicursos foi composto pelos professores Humberto Razente (UFU), Denio Duarte (UFFS) e Ronaldo dos Santos Mello (UFSC), sob coordenação do primeiro.

A qualidade dessa edição é devida essencialmente aos autores e revisores dos trabalhos submetidos. Expressamos nossos fortes agradecimentos pelas contribuições e discussões durante o SBBD 2023.

Para mais informações sobre o SBBD 2023, visite http://sbbd.org.br/2023, o sítio desta edição do evento.

This book of the XXXVIII Brazilian Symposium on Databases (SBBD 2023) includes two book chapters written by the authors of the selected short courses and presented in the edition of the event held from September 25 to 29, 2023. They aim to present relevant topics related to Databases. Moreover, they promote discussions on the topics' fundamentals, trends, and challenges. Each short course lasts four hours and is an excellent opportunity to update academics and professionals participating in the event.

The chapters address the OpenAI programming interface and the manipulation of geospatial data. The short course program committee was composed of Humberto Razente (UFU), Denio Duarte (UFFS), and Ronaldo dos Santos Mello (UFSC) under the coordination of the former.

The richness of this issue can be mainly credited to the authors and reviewers. We greatly thank them for their insightful contributions and discussions during SBBD 2023.

You can find more information about SBBD 2023 by visiting the http://sbbd.org.br/2023 website of the event.

Execução: IGCE Instituto de Ciências Exatas e Informática · PUC Minas · Instituto Federal Minas Gerais

Realização: SBC Sociedade Brasileira de Computação · Cebd SBC Comissão Especial de Banco de Dados

Apoio Acadêmico: UFMG · CEFET-MG

Patrocínio: alfa Engenharia · Cockroach Labs · sólides · stone tech · Google

Agências de fomento: CAPES · FAPEMIG

minicursos

sbbd:01

Capítulo 1

Introdução à API da OpenAI

Alexandre Donizeti Alves

Abstract

OpenAI is a company focused on research and development of artificial intelligence, dedicated to creating advanced systems with wide application in several areas. One of its main tools is the API, which allows you to access its natural language models through a friendly and easy-to-use interface. The API is based on some of the most advanced natural language models in the market, including GPT-3, currently considered one of the most sophisticated models available. With the OpenAI API, developers and companies can integrate advanced natural language processing technologies into their projects, including sentiment analysis, text summarization, text generation, and much more. In this chapter, a detailed overview of the OpenAI API is presented, using practical examples in Python and JavaScript (Node.js).

Resumo

A OpenAI é uma empresa focada em pesquisa e desenvolvimento de inteligência artificial, dedicada na criação de sistemas avançados com ampla aplicação em diversas áreas. Uma de suas principais ferramentas é a API, que permite acessar seus modelos de linguagem natural por meio de uma interface amigável e fácil de usar. A API é baseada em alguns dos modelos de linguagem natural mais avançados do mercado, incluindo o GPT-3, atualmente considerado um dos modelos mais sofisticados disponíveis. Com a API da OpenAI, desenvolvedores e empresas podem integrar tecnologias avançadas de processamento de linguagem natural em seus projetos, incluindo análise de sentimentos, sumarização de textos, geração de texto e muito mais. Neste capítulo, é apresentada uma visão geral detalhada da API da OpenAI, utilizando exemplos práticos em Python e JavaScript (Node.js).

1.1. Introdução

A *OpenAI*[1] é uma organização privada de pesquisa em Inteligência Artificial (IA). Foi fundada em dezembro de 2015 e desde sua concepção, a organização focou em criar e promover soluções que sejam seguras e alinhadas com interesses humanos, permitindo enfrentar os desafios que a IA avançada pode apresentar à sociedade. Em setembro de 2020, foi consolidada uma parceria[2] estratégica entre a *OpenAI* e a *Microsoft*. Neste acordo, a *Microsoft* tornou-se a provedora exclusiva de soluções em nuvem para a *OpenAI*, utilizando sua plataforma *Azure*[3] para hospedar e disponibilizar serviços de IA desenvolvidos pela *OpenAI*. Essa colaboração permite à *Microsoft* integrar recursos da *OpenAI*, como o *ChatGPT* (combina "*chat*", referindo-se à sua funcionalidade de *chatbot*, e "*GPT*", que significa *Generative Pre-trained Transformer)*, em seus produtos e serviços. O *ChatGPT* foi lançado em junho de 2020 e, no final de novembro de 2022, tornou-se disponível para o público em geral. Esse lançamento despertou um interesse público gigantesco e em diferentes áreas [Agathokleous et al. 2023, Cheng et al. 2023a, Cheng et al. 2023b]. Contudo, a ferramenta gerou muitas controvérsias, como por exemplo, pelo seu uso na escrita de artigos científicos [Brainard 2023]. A *OpenAI* também enfrenta problemas como a falta de transparência [van Dis et al. 2023] e a exclusão de usuários de determinados países, como por exemplo, China, Rússia e Ucrânia [Wang 2023]. Além disso, em países como a Itália, o acesso está sendo limitado [Niszczota and Rybicka 2023].

Uma das ferramentas desenvolvidas pela *OpenAI* é sua *API (Application Programming Interface*), projetada para facilitar o acesso a uma série de recursos avançados de linguagem natural. Esta interface oferece aos desenvolvedores uma forma eficiente de integrar capacidades de linguagem natural em seus projetos, proporcionando recursos como análise de sentimentos, sumarização de textos, geração de conteúdo e muito mais. A *API*[4] se fundamenta em modelos de linguagem natural avançados, como o *GPT-3*. Há também o modelo *GPT-3.5*, que representa uma evolução significativa na modelagem de linguagem natural. Utilizando a arquitetura *Transformer*, o modelo foi treinado com um conjunto extenso de dados, possibilitando capturar uma gama mais ampla de nuances e contextos linguísticos. Esse modelo é adaptado para entender e gerar respostas para *prompts* complexos, com diferentes estilos e formatos textuais. Além disso, incorpora um aprendizado acumulado dos modelos anteriores, aprimorando sua eficácia em tarefas de Processamento de Linguagem Natural (PLN) e reduzindo algumas das limitações encontradas em modelos anteriores. O modelo mais recente é o *GPT-4*. Entretanto, esse modelo ainda não foi liberado para acesso e utilização por meio da *API* para o público em geral [Sanderson 2023]. Atualmente, a *OpenAI* disponibiliza sua *API* para vários países, regiões e territórios[5].

[1] https://openai.com/about

[2] https://openai.com/blog/microsoft-invests-in-and-partners-with-openai

[3] https://azure.microsoft.com/pt-br/products/cognitive-services/openai-service/

[4] https://platform.openai.com/docs/api-reference

[5] https://platform.openai.com/docs/supported-countries

Este minicurso tem como objetivo apresentar uma visão geral detalhada da *API* da *OpenAI*, utilizando exemplos práticos em *Python* e *JavaScript* (*Node.js*). Com isso, será possível proporcionar um entendimento das principais funcionalidades da *API*, evidenciando o seu potencial em variados contextos e sua aplicação em cenários reais.

1.2. Configuração da *API*

Esta seção apresenta os principais requisitos para utilizar a *API* da *OpenAI*, enfatizando que os desenvolvedores devem criar uma conta na plataforma e obter uma chave da *API*. Além disso, são abordadas as linguagens de programação suportadas pela *API*, com destaque para *Python* e *JavaScript* (*Node.js*), além de outras com suporte da comunidade. Neste minicurso, são abordadas especificamente as linguagens de programação *Python* e *JavaScript* (*Node.js*), fornecendo instruções detalhadas para instalar e configurar o ambiente de desenvolvimento, incluindo as bibliotecas e ferramentas necessárias para utilizar a *API*. Isso inclui a instalação de pacotes e outras configurações específicas para cada linguagem de programação.

1.2.1. Principais requisitos

Para utilizar a *API* da *OpenAI*, é necessário atender a alguns requisitos essenciais que garantem uma experiência segura e eficiente para todos os usuários. A seguir, estão listados os principais requisitos a serem cumpridos pelos desenvolvedores:

1. **Criar uma conta na plataforma da *OpenAI***: Antes de começar a utilizar a *API*, os desenvolvedores devem criar uma conta na plataforma da *OpenAI*. A conta é essencial para obter acesso aos recursos da *API* e gerenciar suas configurações.

2. **Obter uma chave da *API***: Após criar a conta, os desenvolvedores precisam obter uma chave da *API*. Essa chave é um identificador único que concede acesso à *API* e autentica as solicitações feitas pelos desenvolvedores. A chave desempenha um papel fundamental na garantia da segurança e rastreabilidade da utilização da *API*.

3. **Crédito para utilizar a *API***: Atualmente, ao se cadastrar na plataforma, a *OpenAI* disponibiliza 5 dólares em crédito gratuito, que podem ser utilizados ao longo de três meses. Isso permite aos desenvolvedores explorarem a *API* sem custo inicial. Após consumir esse crédito ou ao final do período de três meses, o desenvolvedor pode optar por pagar conforme a utilização, podendo escolher o modelo de linguagem mais adequado e também financeiramente mais acessível para o seu projeto. Essa flexibilidade permite uma utilização mais controlada e personalizada da *API*, atendendo às necessidades específicas de cada desenvolvedor. A *OpenAI* também impõe limites no número de solicitações que um usuário pode fazer em um determinado período de tempo, garantindo assim que os serviços permaneçam escaláveis e de alta qualidade.

1.2.2. Criando uma conta na plataforma da *OpenAI*

Para criar uma conta na *OpenAI*, siga os passos abaixo:

1. Inicialmente, acesse o *site* oficial da plataforma[6].
2. No canto superior direito, localize e clique no botão "*Sign up*" (Registrar-se).
3. Feito isso, há o redirecionamento para a página de criação de conta. Nesta etapa, há a opção de criar uma nova conta fornecendo um endereço de *e-mail*. Outra opção é fazer a associação à uma conta de *e-mail* (*Gmail,* por exemplo*)* já existente.
4. Em seguida, é enviado um *e-mail* de verificação com um *link* para concluir o cadastro na plataforma.
5. Para completar o cadastro, é necessário fornecer o nome, sobrenome e data de nascimento.
6. Será solicitado também um número de celular para envio de um código de verificação. É importante ressaltar que apenas um número de celular pode ser associado à uma conta na plataforma da *OpenAI*.
7. Para finalizar o processo, basta inserir o código de verificação recebido no celular. Dessa forma, a conta estará pronta para ser utilizada.

1.2.3. Obtendo uma chave da *API*

Após concluir o cadastro na plataforma da *OpenAI*, o próximo passo consiste em adquirir uma chave da *API*. Essa etapa requer que o usuário esteja com sua sessão iniciada na plataforma. Uma vez logado, localize a inicial do nome do usuário no canto superior direito do painel da *OpenAI* e clique sobre ela. No menu que se abre, selecione a opção "*View API Keys*" (Exibir chaves da *API*). Alternativamente, essa funcionalidade pode ser acessada diretamente através da *URL* (*Uniform Resource Locator*) correspondente[7].

Para gerar uma chave nova, clique no botão *"+ Create new secret key*" (Criar nova chave secreta). Isso abrirá uma nova janela e será possível informar um nome para a chave (opcional). Em seguida, clique em "*Create secret key*". Feito isso, uma nova chave será criada. Certifique-se de armazenar essa chave em um local seguro e de fácil acesso. É importante observar que, por razões de segurança, após a criação, não é possível visualizar a chave novamente por meio da conta na *OpenAI*. Caso ocorra a perda da chave, será necessário gerar uma nova.

1.2.4. Linguagens de programação suportadas pela *API*

A *API* da *OpenAI* é projetada para ser compatível com várias linguagens de programação, permitindo que desenvolvedores integrem facilmente os recursos de linguagem natural em suas aplicações. Embora a *API* não seja restrita a uma linguagem de programação específica, ela pode ser utilizada em ambientes que possam fazer solicitações *HTTP* (*Hypertext Transfer Protocol*) e processar respostas *JSON*

[6] https://openai.com

[7] https://platform.openai.com/account/api-keys

(*JavaScript Object Notation*). Isso inclui linguagens como *Python*, *JavaScript*, *Java*, *Ruby* e outras, proporcionando aos desenvolvedores a liberdade de escolher a linguagem que melhor se adapte às suas necessidades. É importante destacar que a *OpenAI* dá suporte oficialmente para as linguagens *Python* e *JavaScript* (*Node.js*). No entanto, há outras linguagens de programação com suporte da comunidade[8]. A seguir são apresentadas instruções detalhadas para a instalação e configuração do ambiente de desenvolvimento nas linguagens de programação *Python* e *JavaScript* (*Node.js*).

1.2.5. Configurando o ambiente para *Python*

Para desenvolver em *Python* foi utilizado o *Google Colab*[9], também conhecido como *Colaboratory*, uma plataforma *online* gratuita oferecida pelo *Google* que permite a criação e execução de código *Python* em um ambiente baseado na nuvem, ou seja, sem a necessidade de configuração de um ambiente de desenvolvimento local. O *Colab* oferece suporte a *notebooks* do *Jupyter*, permitindo criar documentos interativos que misturam código executável, texto formatado, imagens, vídeos, tabelas e gráficos. Os *notebooks* do *Colab* são salvos diretamente no *Google Drive*, sendo possível compartilhá-los com outros usuários. Neste minicurso, foi utilizada a versão 3.10.12 do *Python* disponível no *Colab*.

O primeiro passo no *Google Colab*[10] é instalar o pacote *Python* da *OpenAI* para utilizar as funcionalidades da *API*. A Listagem 1.1 mostra o código para fazer essa instalação.

```
print(f"Instalando a biblioteca da API da OpenAI...")

!pip install openai -q

print("API da OpenAI instalada!")
```

```
Instalando a biblioteca da API da OpenAI...
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 73.6/73.6 kB 1.3 MB/s eta 0:00:00
API da OpenAI instalada!
```

Listagem 1.1. Instalação do pacote *Python* da *API* da *OpenAI*.

Após concluir a instalação do pacote da *OpenAI*, é necessário configurar uma chave para que o pacote possa acessar a *API*. O trecho de código apresentado na Listagem 1.2 mostra a configuração da chave diretamente no código, substituindo `YOUR_API_KEY` pelo *token* gerado durante a criação da chave da *API* na plataforma da *OpenAI*. No entanto, é importante destacar que esse procedimento no *Google Colab* não é seguro, pois trata-se de um ambiente em que o código pode ser compartilhado ou exposto publicamente. Isso ocorre porque a chave da *API* é inserida diretamente no código-fonte, o que a torna facilmente identificável por qualquer usuário com acesso ao código. A variável `openai.api_key` faz parte do pacote `openai` e tem a função de armazenar a chave utilizada para autenticar as chamadas à *API*.

[8] https://platform.openai.com/docs/libraries

[9] https://colab.research.google.com

[10] https://colab.research.google.com/?utm_source=scs-index

```python
import openai

# definir a chave da API
openai.api_key = 'YOUR_API_KEY'
```

Listagem 1.2. Configuração da chave de acesso da *API* diretamente no código.

Para manter a chave da *API* segura, é recomendável utilizar variáveis de ambiente ou um arquivo de configuração separado do código-fonte. Dessa forma, é possível carregar a chave da *API* de forma segura sem expô-la no código diretamente. Por exemplo, é possível definir a chave da *API* como uma variável de ambiente e acessá-la no código, conforme mostrado na Listagem 1.3. O pacote `os` é um módulo em *Python* que permite interagir com o sistema operacional, incluindo a obtenção de variáveis de ambiente. Isso pode ser feito utilizando a função `gentenv`, que permite acessar o valor de um variável de ambiente específica. Neste cenário exemplificado, é necessário definir a chave da *API* como uma variável de ambiente denominada `OPENAI_API_KEY`, com um valor correspondente à chave da *API* obtida na plataforma da *OpenAI*. Essa prática garante que as informações sensíveis permaneçam protegidas e que a segurança das credenciais seja mantida intacta.

```python
import os
import openai

openai.api_key = os.getenv("OPENAI_API_KEY")
```

Listagem 1.3. Configuração da chave de acesso da *API* a partir de uma variável de ambiente.

A utilização de um arquivo de configuração (texto) também é uma prática recomendada para manter informações sensíveis, como chaves de *API*, seguras e separadas do código-fonte. Um exemplo prático consiste em criar um arquivo de texto contendo apenas a chave da *API* gerada na plataforma da *OpenAI*. Na Listagem 1.4 é apresentado um trecho de código que lê o conteúdo de um arquivo de texto, denominado `openai_chave_minicurso`, e armazena o valor lido como a chave de autenticação para acessar a *API*.

```python
import openai

# ler o conteúdo do arquivo de texto
with open('/content/openai_chave_minicurso.txt', 'r') as file:
   chave_api = file.read()

# definir a chave da API
openai.api_key = chave_api
```

Listagem 1.4. Configuração da chave de acesso da *API* a partir de um arquivo de texto.

Outra abordagem possível é realizar o *upload* de um arquivo de texto, como exemplificado no trecho de código da Listagem 1.5. Após o *upload*, é feita a leitura do conteúdo do arquivo de texto e o valor obtido é utilizado para configurar a chave de acesso à *API*.

Uma vez configurada a chave de acesso, é importante verificar se a configuração foi realizada corretamente. A Listagem 1.6 mostra um trecho de código simples para começar a explorar as funcionalidades oferecidas pela *API*. Nesse código, a *API* é empregada para gerar uma "continuação" (*completion*[11]) a partir do *prompt* "`O Brasil é um país`" e utilizando o modelo `text-davinci-003`. A saída para a execução desse código foi "`localizado na América do Sul. É formado pelos 26`". Note que a "continuação" retornada depende do valor *default* do número total de *tokens* na *API*.

```
import openai
from google.colab import files

# fazer upload do arquivo de texto
upload_arquivo = files.upload()

# obter o nome do arquivo
nome_arquivo = list(upload_arquivo.keys())[0]

# ler o conteúdo do arquivo
with open(nome_arquivo, 'r') as file:
   chave_api = file.read()

# definir a chave da API
openai.api_key = chave_api
```

```
Browse... openai_chave_minicurso.txt
openai_chave_minicurso.txt(text/plain) - 51 bytes, last modified: n/a - 100% done
Saving openai_chave_minicurso.txt to openai_chave_minicurso.txt
```

Listagem 1.5. Configuração da chave de acesso da *API* a partir do *upload* de um arquivo de texto.

Os parâmetros `model` e `prompt` são obrigatórios. O parâmetro `model` especifica qual modelo ou mecanismo de geração de texto deve ser utilizado para produzir a resposta. A *OpenAI* disponibiliza vários modelos com diferentes capacidades e preços. É importante estar ciente de que a *OpenAI* regularmente introduz novos modelos, tornando os modelos antigos obsoletos[12]. Neste minicurso, o modelo `text-davinci-003` será utilizado com frequência. Trata-se de um modelo que é otimizado para tarefas de geração de texto. O parâmetro `prompt` é a entrada inicial que deve ser fornecida ao modelo para orientar a geração de texto. Essa entrada fornece contexto e diretrizes para a criação da resposta desejada pelo usuário. Portanto, quando se invoca a função `openai.Completion.create()`, está-se solicitando à *API* que utilize o modelo especificado e o *prompt* fornecido para gerar uma resposta de texto correspondente.

```
resposta = openai.Completion.create(
   model = "text-davinci-003",
   prompt = "O Brasil é um país"
)

print(resposta.choices[0].text)
```

```
 localizado na América do Sul. É formado pelos 26
```

Listagem 1.6. Trecho de código em Python para gerar uma resposta de texto.

[11] https://platform.openai.com/docs/api-reference/completions

[12] https://platform.openai.com/docs/deprecations

É importante destacar que toda vez que esse código for executado será gerada uma resposta diferente, devido à natureza generativa do modelo. Adicionalmente, é importante estar ciente da restrição inerente ao modelo `text-davinci-003`, que permite somente três solicitações por minuto. Caso esse limite seja atingido, é necessário aguardar 20 segundos antes de executar novamente o código. É possível aumentar esse limite adicionando um método de pagamento[13] à conta do usuário.

O número total de *tokens* processados em uma única solicitação, incluindo o *prompt* e a resposta, deve respeitar o limite máximo de *tokens* do modelo. Em grande parte dos casos, esse limite é de 4.096 *tokens* ou cerca de 3.000 palavras. De modo geral, um *token* equivale a aproximadamente 4 caracteres ou 0,75 palavras em inglês. Os preços[14] seguem uma lógica de pagamento por uso, calculado a cada 1.000 *tokens* processados.

1.2.6. Configurando o ambiente para *JavaScript* (*Node.js*)

JavaScript[15] (normalmente abreviado para *JS*) é uma linguagem de programação de alto nível, dinâmica, leve, interpretada e baseada em objetos. Foi originalmente criada para ser executada em navegadores, permitindo a criação de páginas *Web* interativas e dinâmicas. Com o tempo, no entanto, seu escopo de aplicação se expandiu, sendo utilizada também no desenvolvimento de aplicativos de servidor, aplicativos móveis e muito mais. *Node.js*[16] é um ambiente de tempo de execução de código aberto que permite executar *JavaScript* no lado do servidor. Uma opção para desenvolver em *Node.js* é utilizar o *Replit*, uma plataforma *online* que oferece um ambiente de desenvolvimento integrado baseado na nuvem. A plataforma permite que desenvolvedores escrevam, executem e colaborem em códigos de várias linguagens de programação diretamente no navegador, sem a necessidade de configurações complicadas ou instalações locais.

Para configurar a *API* da *OpenAI* no *Replit* para utilização com *Node.js*, é necessário acessar a página do *Replit*[17] e criar um novo projeto clicando em "+ *Create Repl*". Uma janela aparecerá, permitindo selecionar a opção "*Node.js*". Nesse ponto, é necessário informar um nome para o projeto (por exemplo, `TesteAPIOpenAI`), e então clicar no botão "+ *Create Repl*" para finalizar. Feito isso, um ambiente de desenvolvimento estará pronto para ser utilizado, conforme ilustrado na Figura 1.1.

No arquivo `package.json`, é necessário verificar se o pacote `openai` está listado como uma dependência. Caso contrário, será necessário instalá-lo. Isso pode ser feito no terminal do *Replit* utilizando o seguinte comando: `npm install openai`. Para configurar a chave de acesso é possível utilizar a ferramenta "*Secrets*", localizada no painel "*Tools*" no *Replit*. Após clicar no ícone correspondente, uma janela é aberta. Clique em "+ *New Secret*" para adicionar uma nova chave de acesso. Para isso basta informar um nome para a chave (por exemplo, `OPENAI_API_KEY`) e o seu respectivo

[13] https://platform.openai.com/account/billing

[14] https://openai.com/pricing

[15] https://developer.mozilla.org/en-US/docs/Web/JavaScript

[16] https://nodejs.org

[17] https://replit.com

valor. Para finalizar, clique no botão "*Save*". Dessa forma, a chave da *API* ficará segura em "*Secrets*" e não será exposta diretamente no código-fonte.

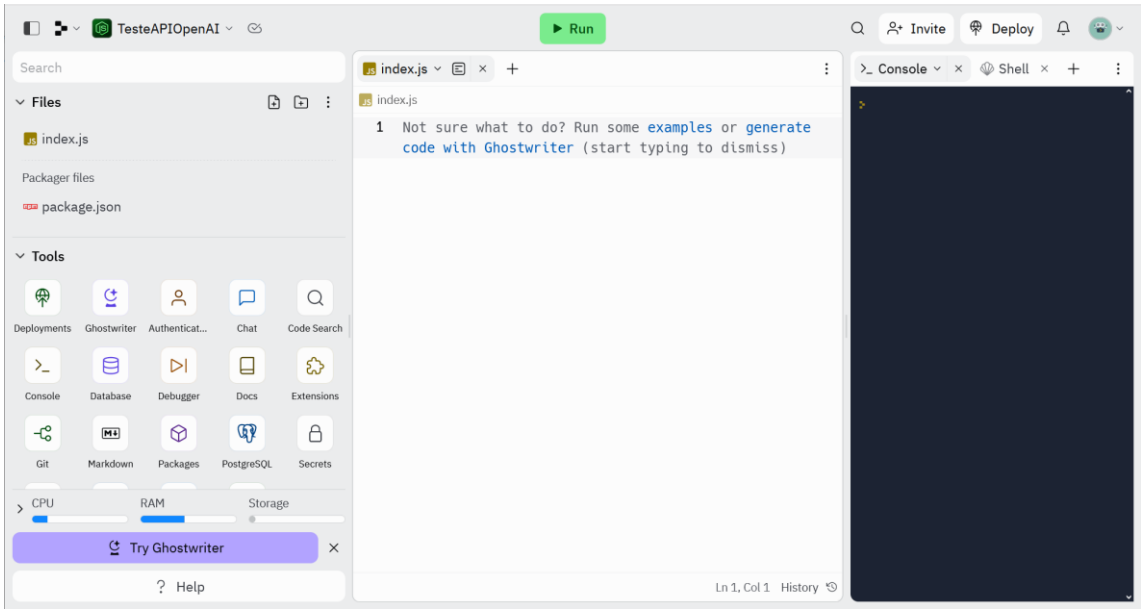


Figura 1.1. Ambiente de desenvolvimento *Node.js* na plataforma *Replit*.

Após a configuração da chave de acesso, é possível explorar as funções disponibilizadas pelo pacote `openai`. Por exemplo, é possível utilizar a função `openai.Completion.create()` para gerar texto com base em um *prompt*, assim como foi mostrado anteriormente em *Python*. No arquivo *JavaScript* principal do projeto (`index.js`), é necessário importar o pacote `openai` e utilizar a função `process.env` para acessar o valor da chave da *API* definida em "*Secrets*". Para testar o código, basta clicar no botão "*Run*" no *Replit*.

A Figura 1.2 ilustra um trecho de código para gerar uma "continuação" a partir do *prompt* "`O Brasil é um país`" e utilizando o modelo `text-davinci-003`. A saída para a execução desse código foi "`localizado na América do Sul, tendo a maior part`". De maneira bem resumida, o código importa as classes `Configuration` e `OpenAIApi` do pacote `openai`, configura a chave da *API* através de uma instância de `Configuration` utilizando uma variável de ambiente. Em seguida, é criada uma instância de `OpenAIApi` com essa configuração. Também é definida uma função assíncrona denominada, por exemplo, `executarPrompt()`. Dentro dessa função, a *API* é utilizada para gerar texto em resposta ao *prompt* de acordo com o modelo definido. O resultado é impresso no painel "*Console*". Por fim, a função `executarPrompt()` é chamada para executar o código.

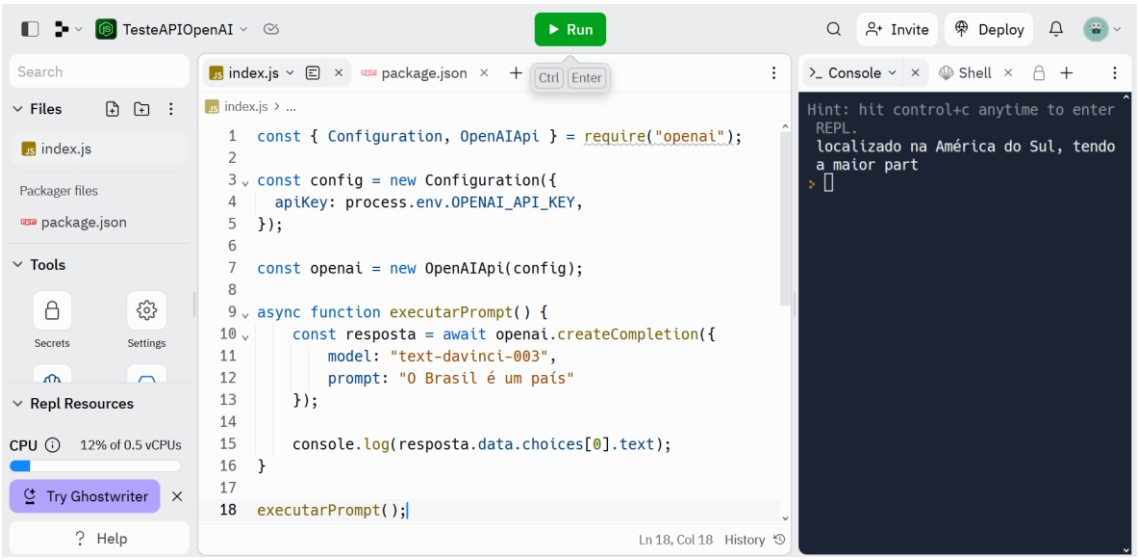


Figura 1.2. Trecho de código em *JavaScript* (*Node.js*) para gerar uma resposta de texto.

1.3. Como utilizar a API

Existem várias formas de utilizar a *API* da *OpenAI*, dependendo das necessidades de cada projeto. A *OpenAI* fornece bibliotecas oficiais para as linguagens *Python* e *JavaScript* (*Node.js*) que facilitam a utilização da *API*. Com essas bibliotecas, é possível fazer chamadas à API de forma simplificada e sem a necessidade de lidar diretamente com o protocolo *HTTP*. Essa abordagem será apresentada brevemente na subseção 1.3.4 utilizando *Python* e na subseção 1.3.5 utilizando *JavaScript* (*Node.js*). Além disso, na seção 1.4 serão apresentados vários exemplos práticos em diferentes contextos. Outra forma de utilizar a *API* é fazendo requisições *HTTP* diretas utilizando bibliotecas, como `requests` para *Python* ou `axios` para *Node.js*. Essa abordagem será apresentada nesta seção e pode ser mais flexível em alguns casos, permitindo a personalização de parâmetros e cabeçalhos específicos. Também há a possibilidade de utilizar plataformas de terceiros. Existem diversas plataformas que fornecem interfaces mais amigáveis para a utilização da *API* da *OpenAI*, como por exemplo, *Hugging Face*[18]. A própria *OpenAI* fornece uma interface mais amigável com a ferramenta *OpenAI Playground*[19]. Esta abordagem não será apresentada neste minicurso.

Nesta seção é apresentada uma visão geral dos *endpoints* atualmente disponíveis na *API* da *OpenAI*. A *API* oferece *endpoints* que permitem aos desenvolvedores acessar diferentes modelos de IA para tarefas específicas. Também é apresentado como fazer requisições *HTTP* à *API* e entender as respostas, e como utilizá-las em *Python* e *JavaScript* (*Node.js*). Após enviar uma solicitação à *API*, o desenvolvedor receberá uma resposta que contém os resultados da operação solicitada. Essa resposta pode estar em vários formatos, como *JSON*, texto simples ou outros formatos específicos de cada *endpoint*. É importante que os desenvolvedores entendam como interpretar essas respostas para poder utilizá-las de forma adequada em suas aplicações. Além disso, é

[18] https://huggingface.co/models

[19] https://platform.openai.com/playground

apresentado brevemente como utilizar a biblioteca da *API* da *OpenAI* em *Python* e *JavaScript* (*Node.js*).

1.3.1. Visão geral dos *endpoints*

Basicamente, um *endpoint* é um ponto de entrada para interagir com os serviços e recursos disponibilizados pela *API* da *OpenAI*. A *API* disponibiliza diversos *endpoints*: *completions*[20] (refere-se a "completar" ou "continuar" um texto), *chat completions*[21], geração de imagens[22], fala para texto (transcrições e traduções)[23] e moderações[24]. Nesta subseção, é apresentada uma visão geral dos *endpoints* "*completions*" e "*chat completions*". Na seção 1.4 também são apresentados alguns exemplos que utilizam o *endpoint* para geração de imagens. Por limitações de espaço, os outros *endpoints* não serão apresentados neste minicurso.

O principal *endpoint* da *API* ainda é o "*completions*", que fornece uma interface simples e flexível o suficiente para realizar praticamente qualquer tarefa de PLN, incluindo geração de conteúdo, sumarização, análise de sentimentos e muito mais. Esse *endpoint* é geralmente utilizado para gerar "continuação" de texto com base no *prompt* fornecido[25].

Além disso, é possível utilizar o *endpoint* "*completions*" com diferentes modelos. Os modelos *GPT* da *OpenAI* foram treinados para entender linguagem natural e código. Esses modelos fornecem saídas de texto em resposta às suas entradas. As entradas para esses modelos também são chamadas de *prompts*. Projetar um *prompt* é essencialmente como se "programa" um modelo *GPT*, geralmente fornecendo instruções ou alguns exemplos de como concluir uma tarefa com êxito. Os modelos mais recentes, `gpt-4` e `gpt-3.5-turbo`, são acessados por meio do *endpoint* "*chat completions*". Atualmente, apenas os modelos legados mais antigos estão disponíveis por meio do *endpoint* "*completions*"[26]. Esse *endpoint* recebeu sua atualização final em julho de 2023 e será encerrado em 4 de janeiro de 2024.

O modelo `text-davinci-003`, disponível no *endpoint* "*completions*", é um dos modelos de linguagem mais poderosos e versáteis disponíveis na plataforma da *OpenAI*. Ele é parte da família de modelos *GPT* e é treinado com uma arquitetura *Transformer* de várias camadas, com 175 bilhões de parâmetros. Esse modelo é especialmente adequado para tarefas de linguagem natural que exigem compreensão e geração de texto altamente avançadas. É treinado em um conjunto de dados extremamente grande e diversificado, que inclui textos em vários idiomas e domínios, como notícias, livros, artigos científicos, conversas cotidianas e muito mais.

---

[20] https://platform.openai.com/docs/guides/gpt/completions-api

[21] https://platform.openai.com/docs/guides/gpt/chat-completions-api

[22] https://platform.openai.com/docs/guides/images/image-generation

[23] https://platform.openai.com/docs/guides/speech-to-text/speech-to-text

[24] https://platform.openai.com/docs/guides/moderation/moderation

[25] https://platform.openai.com/docs/quickstart/introduction

[26] https://platform.openai.com/docs/guides/gpt/gpt-models

A Listagem 1.7 mostra um trecho de código em *Python* para listar todos os modelos disponíveis na *API* da *OpenAI*. A saída desse trecho de código lista o número de modelos disponíveis na *API* (53, atualmente) e o nome de cada um dos modelos (foram exibidos apenas 4 por limitações de espaço). É importante lembrar que a ordem dos nomes dos modelos pode mudar toda vez que esse código for executado.

```
# obter a lista de modelos
modelos = openai.Model.list()

# imprimir total de modelos
print(len(modelos['data']))

# imprimir os nomes dos modelos
for modelo in modelos['data']:
   print(modelo['id'])
```

```
53
text-davinci-edit-001
babbage-code-search-code
text-similarity-babbage-001
code-davinci-edit-001
```

Listagem 1.7. Trecho de código em *Python* para listar todos os modelos disponíveis na *API*.

Outro *endpoint* é o "*chat completions*"[27], que recebe uma lista de mensagens como entrada e retorna uma mensagem gerada pelo modelo como saída. Embora o formato de *chat* seja projetado para facilitar as conversas em vários turnos (*multi-turn*), é igualmente útil para tarefas de turno único (*single-turn*), ou seja, sem nenhuma conversa. O *endpoint* "*chat completions*" foi projetado para tornar mais fácil ter conversas interativas e dinâmicas com o modelo. Em vez de enviar apenas um *prompt* de texto simples como no *endpoint* "*completions*", agora é possível enviar diversas mensagens como entrada. Cada mensagem tem um papel e um conteúdo (o texto da mensagem). Os papeis disponíveis são: `system` (sistema), `user` (usuário) e `assistant` (assistente). Normalmente, uma conversa é formatada primeiro com uma mensagem do sistema, seguida por mensagens alternadas do usuário e do assistente.

1.3.2. Requisições *HTTP* utilizando *Python*

Nesta subseção é abordado como interagir com a *API* em *Python* utilizando a biblioteca `requests`, que permite enviar solicitações *HTTP* de forma simplificada, conforme mostra o trecho de código na Listagem 1.8. Inicialmente, é definido o modelo a ser utilizado (`text-davinci-003`). Posteriormente, é construído o *endpoint* "*completions*" da *API*, combinando uma *URL* base com o nome do modelo escolhido. Os parâmetros da requisição são definidos em um dicionário, denominado "`parametros`". O parâmetro `prompt` especifica a pergunta para a qual se deseja uma resposta, enquanto `temperature` e `max_tokens` são utilizados para controlar a aleatoriedade e o comprimento máximo da resposta, respectivamente. Adicionalmente, o cabeçalho da requisição também é definido em um dicionário, denominado "`headers`", contendo o tipo de conteúdo esperado e a autorização necessária para acessar a *API*. A chave de acesso é assumida como sendo uma variável global (`openai.api_key`), já definida em um trecho de código anterior. Finalmente, a requisição é executada utilizando `requests.post()`, e a resposta armazenada em uma variável, denominada "`resposta`".

[27] https://platform.openai.com/docs/api-reference/chat

```python
import requests

# definir o modelo
modelo = "text-davinci-003"

# definir o endpoint da API
endpoint = "https://api.openai.com/v1/engines/" + modelo + "/completions"

# definir os parâmetros e cabeçalho da requisição
parametros = {
    "prompt": "Qual é a capital do Brasil?",
    "temperature": 0.7,
    "max_tokens": 100
}
headers = {
    "Content-Type": "application/json",
    "Authorization": f"Bearer {openai.api_key}"
}

# fazer a requisição
resposta = requests.post(endpoint, json=parametros, headers=headers)
```

Listagem 1.8. Trecho de código em *Python* para fazer uma requisição *HTTP* utilizando o *endpoint* "*completions*".

A Listagem 1.9 apresenta o trecho de código para mostrar a resposta no formato *JSON*. O trecho de código `resposta.json()` converte a resposta *HTTP*, que é presumivelmente um conteúdo em formato *JSON*, em um objeto *Python*, como um dicionário, permitindo que os dados sejam acessados e manipulados de maneira mais simples no código. A Listagem 1.10 apresenta o trecho de código para mostrar apenas o texto da resposta. O código extrai e imprime o texto retornado pela *API* da resposta no formato *JSON*, removendo espaços em branco extras no início e no fim do texto.

```python
# formato json
resposta.json()
```

```
{'warning': 'This model version is deprecated. Migrate before January 4, 2024 to avoid disruption of service.
/docs/deprecations',
 'id': 'cmpl-7mKTmu7S2kMsVolNHV9FUAnoJjiDK',
 'object': 'text_completion',
 'created': 1691752822,
 'model': 'text-davinci-003',
 'choices': [{'text': '\n\nBrasília é a capital do Brasil.',
   'index': 0,
   'logprobs': None,
   'finish_reason': 'stop'}],
 'usage': {'prompt_tokens': 8, 'completion_tokens': 13, 'total_tokens': 21}}
```

Listagem 1.9. Trecho de código para mostrar a resposta no formato *JSON*.

```python
print(resposta.json()["choices"][0]["text"].strip())
```

```
Brasília é a capital do Brasil.
```

Listagem 1.10. Trecho de código em Python para mostrar apenas o texto da resposta.

O trecho de código apresentado na Listagem 1.11 utiliza a biblioteca `requests` para fazer uma interação com a *API*. Neste exemplo é utilizado o *endpoint* "*chat completions*" para realizar a análise de sentimento de uma avaliação de um produto. Basicamente, são definidas três mensagens: a mensagem do sistema (`system`) informa ao modelo da *OpenAI* sobre o papel do assistente virtual, que é analisar sentimentos de avaliações; a mensagem do usuário (`user`), fornece ao modelo uma avaliação fictícia de um produto para análise; e a mensagem do assistente (`assistant`), especifica o formato

esperado da resposta do modelo que, neste caso, é uma simples classificação de sentimento como “Positivo” ou “Negativo”.

Os parâmetros da requisição incluem o nome do modelo utilizado e as mensagens formatadas. O cabeçalho da requisição indica que os dados enviados estão em formato *JSON* e inclui uma chave de autorização para acessar a *API*. Finalmente, a requisição é enviada utilizando o método `post` da biblioteca `requests`, e a resposta da *API* é obtida e convertida para formato *JSON* para fácil acesso e manipulação. Por exemplo, é possível imprimir apenas a resposta do assistente com o seguinte trecho de código: `print(resposta.json()["choices"][0]["message"]["content"])`.

```python
import requests

endpoint = "https://api.openai.com/v1/chat/completions"

mensagem_sistema = 'Você é um assistente que analisa sentimentos de avaliações de produtos'
mensagem_usuario = "Aqui está uma avaliação de um produto: 'Este produto é incrível!"
mensagem_assistente = "Classifique o sentimento retornando apenas 'Positivo' ou 'Negativo'. "

parametros = {
   "model": "gpt-3.5-turbo-0613",
   "messages": [
      {"role": "system", "content": mensagem_sistema},
      {"role": "user", "content": mensagem_usuario},
      {"role": "assistant", "content": mensagem_assistente}]
}
headers = {
   "Content-Type": "application/json",
   "Authorization": f"Bearer {openai.api_key}"
}

resposta = requests.post(endpoint, json=parametros, headers=headers)
resposta.json()
```

```
{'id': 'chatcmpl-7mL2dIYHKqqKBBv4gQAb4NuOmHwB0',
 'object': 'chat.completion',
 'created': 1691754983,
 'model': 'gpt-3.5-turbo-0613',
 'choices': [{'index': 0,
   'message': {'role': 'assistant', 'content': 'Positivo'},
   'finish_reason': 'stop'}],
 'usage': {'prompt_tokens': 67, 'completion_tokens': 3, 'total_tokens': 70}}
```

Listagem 1.11. Código em *Python* para fazer uma requisição *HTTP* utilizando o *endpoint* “*chat completions*”.

1.3.3. Requisições *HTTP* utilizando *JavaScript* (*Node.js*)

Nesta subseção é apresentado como realizar requisições *HTTP* em *Node.js* utilizando a biblioteca `axios`. Esse pacote simplifica as chamadas *HTTP* e pode ser facilmente integrado em projetos *Node.js*. Para instalá-lo, basta executar o seguinte comando: `npm install axios`. Com o pacote instalado, é possível realizar chamadas a diversos serviços *Web*, incluindo a *API* da *OpenAI*.

A Figura 1.3 mostra o código em *JavaScript* (*Node.js*) para fazer uma requisição *HTTP* utilizando o *endpoint* “*completions*”. Esse código utiliza a biblioteca `axios` para realizar uma requisição *HTTP* utilizando o método `post`. Primeiramente, a biblioteca é importada e a chave da *API* é recuperada do ambiente *Node.js* no *Replit*. Em seguida, um cliente `axios` é configurado com o cabeçalho de autorização necessário. O código também define o modelo a ser utilizado e o *endpoint*. Os parâmetros da requisição são definidos, especificando a pergunta a ser enviada para a *API*. A requisição é realizada e, quando uma resposta for recebida, o texto resultante é exibido no painel “*Console*”. Se ocorrer algum erro na chamada, uma mensagem será exibida juntamente com o erro identificado.

O código apresentado na Figura 1.4 também faz uma requisição *HTTP*, mas utilizando o *endpoint* "*chat completions*". Após configurar o cliente `axios` com a chave da *API*, define-se o *endpoint* e as mensagens para interagir com o assistente virtual: a mensagem do sistema define o papel do assistente; a mensagem do usuário fornece uma avaliação de produto e a mensagem do assistente orienta sobre a resposta desejada. A requisição é feita com essas mensagens e a resposta do assistente é exibida no painel "*Console*". Em caso de erro na chamada, uma mensagem de erro será exibida.

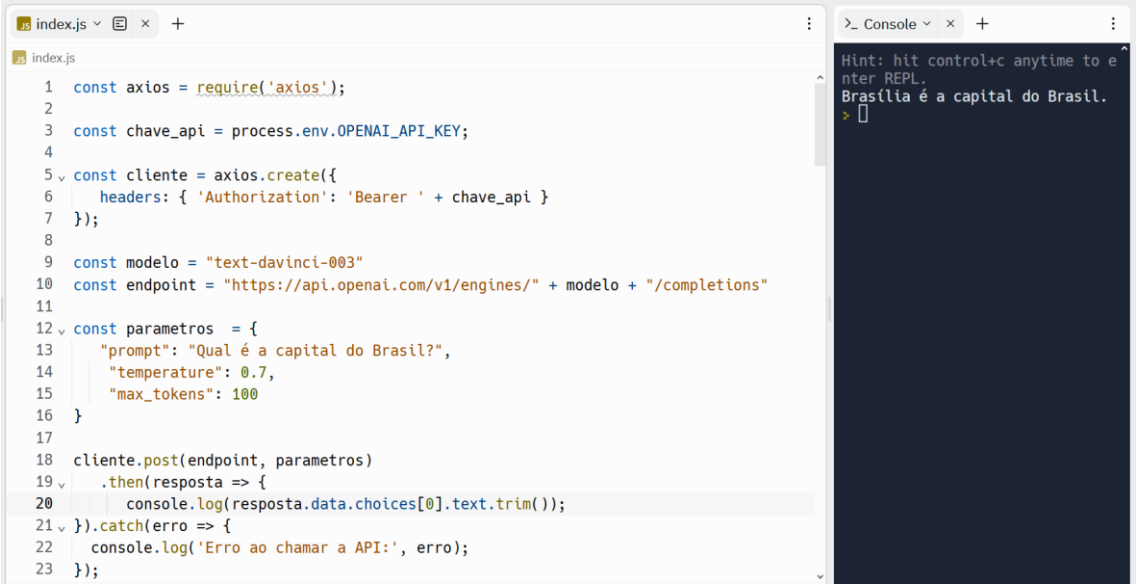


***Figura* 1.3. Código em *JavaScript* (*Node.js*) para fazer uma requisição *HTTP* utilizando o *endpoint* "*completions*".**

***Figura* 1.4. Código em *JavaScript* (*Node.js*) para fazer uma requisição *HTTP* utilizando o *endpoint* "*chat completions*".**

1.3.4. Utilizando a biblioteca da *API* em *Python*

Ao trabalhar com a *API* em *Python*, pode-se utilizar a biblioteca oficial fornecida pela própria *OpenAI*. Esta biblioteca simplifica a interação com a *API*, eliminando a necessidade de lidar diretamente com requisições *HTTP*. A Listagem 1.12 mostra como utilizar essa abordagem para otimizar a integração e obtenção de respostas da *API*. O trecho de código solicita ao modelo que responda a pergunta "`Qual é a capital do Brasil?`" e, em seguida, imprime a resposta gerada.

```
resposta = openai.Completion.create(
   model = "text-davinci-003",
   prompt = "Qual é a capital do Brasil?",
   temperature = 0.7,
   max_tokens = 100
)

print(resposta.choices[0].text.strip())
```

```
Brasília é a capital do Brasil.
```

Listagem 1.12. Trecho de código em *Python* utilizando a biblioteca da *API* com o *endpoint "completions"*.

A resposta é um objeto da *OpenAI* no formato *JSON*, que representa o resultado de uma solicitação de "continuação de texto" (*text completion*), conforme ilustrado na saída do trecho de código apresentado na Listagem 1.13. Além do aviso sobre a obsolescência do modelo, também é retornado um identificador único, o tipo de objeto (*endpoint*) retornado, e a resposta gerada, que nesse caso é "`Brasília`". Também fornece detalhes sobre os *tokens* utilizados na solicitação. Por exemplo, para obter o total de *tokens* utilizados na resposta, basta executar o seguinte trecho de código: `resposta.usage.total_tokens`.

```
resposta
```

```
<OpenAIObject text_completion id=cmpl-7mLAQP46gTAGFG3xlDGCOzSuaEv6D at 0x7f6a3eda2a20> JSON: {
  "warning": "This model version is deprecated. Migrate before January 4, 2024 to avoid disruption of service.
/docs/deprecations",
  "id": "cmpl-7mLAQP46gTAGFG3xlDGCOzSuaEv6D",
  "object": "text_completion",
  "created": 1691755466,
  "model": "text-davinci-003",
  "choices": [
    {
      "text": "\n\nBras\u00edlia.",
      "index": 0,
      "logprobs": null,
      "finish_reason": "stop"
    }
  ],
  "usage": {
    "prompt_tokens": 8,
    "completion_tokens": 7,
    "total_tokens": 15
  }
}
```

Listagem 1.13. Trecho de código para mostrar a resposta no formato *JSON*.

1.3.5. Utilizando a biblioteca da *API* em *JavaScript* (*Node.js*)

Nesta subseção, é abordado como utilizar a biblioteca da *API* da *OpenAI* em *JavaScript* (*Node.js*) para interagir com um modelo de linguagem de forma eficiente. A Figura 1.5 mostra como utilizar a biblioteca da *API* com o *endpoint* "*completions*". O código

importa a biblioteca da *OpenAI* para *Node.js*, define a chave da *API* e cria uma função assíncrona, denominada `executarPrompt()`. Essa função faz uma requisição à *API* perguntando qual "`Qual é a capital do Brasil?`". A resposta gerada pelo modelo é exibida no painel "*Console*". E por fim, a função é executada. Nesse exemplo, para obter o total de *tokens* utilizados na resposta, é possível acessar a propriedade `usage.total_tokens` do objeto de resposta. Assim, dentro da função `executarPrompt`(), após imprimir a resposta gerada pelo modelo, é possível adicionar o seguinte código: `console.log(resposta.data.usage.total_tokens);`.

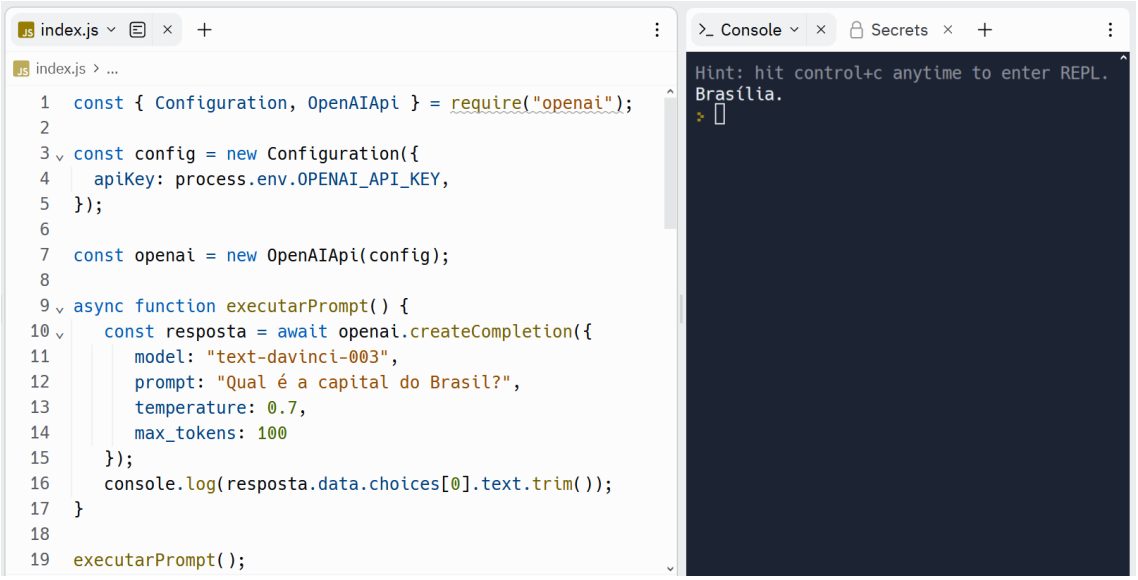


Figura 1.5. Código em *JavaScript* (*Node.js*) utilizando a biblioteca da *API* com o *endpoint* "*completions*".

1.4. Exemplos práticos

Nesta seção são apresentados exemplos práticos de como a *API* da *OpenAI* pode ser utilizada em diferentes contextos. Esses exemplos têm como objetivo mostrar como a *API* pode ser aplicada em situações reais e como pode ajudar a resolver problemas específicos. Os exemplos apresentados são centrados predominantemente em dados textuais, dando ênfase a técnicas de PLN, tais como classificação de textos, análise de sentimentos, sumarização de textos, dentre outras abordagens utilizadas para análise de linguagem natural. Também são apresentados exemplos de geração de código e imagens. Esses exemplos demonstram como a *API* pode ser utilizada para gerar trechos de código de forma automatizada, bem como para criar imagens com base em descrições em linguagem natural. Também são apresentados alguns exemplos de como essas técnicas podem ser aplicadas em cenários do mundo real. Os exemplos práticos apresentados são implementados em *Python*, utilizando o ambiente *Google Colab*, mas podem ser facilmente adaptados para *JavaScript* (*Node.js*). A inspiração para esses exemplos provém de conteúdo disponibilizado na página oficial da *OpenAI*[28], assim como de *sites* [Cotton 2023, Smigel 2023, Training 2023] e vídeos[29].

[28] https://platform.openai.com/examples

[29] https://www.youtube.com/@codigofontetv

1.4.1. Dados textuais

Nesta subseção, é explorada a utilização da *API* da *OpenAI* para o tratamento de dados textuais, abordando diversas técnicas de PLN para mostrar a versatilidade da *API*. A maioria dos exemplos se baseia no *endpoint* "*completions*. Vale ressaltar que os exemplos baseados em "*completions*" podem ser facilmente adaptados para o *endpoint* "*chat completions*".

Inicialmente, é apresentada na Listagem 1.14 o código para uma função, denominada `formatar_saida()`, que tem como objetivo limpar a saída de dados removendo quaisquer caracteres de quebra de linha de uma *string*, utilizando a biblioteca de expressões regulares `re`. Esta função é utilizada para aprimorar a formatação das saídas em alguns exemplos.

```python
import re

def formatar_saida(saida):
   return re.sub(r'^\s+', '', saida)
```

Listagem 1.14. Função para remover quaisquer caracteres de quebra de linha.

Gerando *n* respostas

O trecho de código da Listagem 1.15 utiliza a *API* para gerar uma "continuação" sobre a frase "`O Brasil é famoso`". O parâmetro de temperatura é definido como `1`, o que permite variar a criatividade das respostas. Neste exemplo, o destaque é o parâmetro `n` definido como `3`, indicando que serão geradas três respostas diferentes. Depois de receber as respostas, o código percorre cada uma delas, exibindo seu conteúdo.

```python
respostas = openai.Completion.create(
  model = "text-davinci-003",
  prompt = "O Brasil é famoso",
  max_tokens = 15,
  temperature = 1,
  n = 3
)

for resposta in respostas['choices']:
   print(resposta['text'])
```

```
 pela sua exuberância cultural. O país é
 por muitas coisas, como seu clima am
 por suas inúmeras belezas naturais. É
```

Listagem 1.15. Trecho de código para gerar *n* respostas.

Correção Gramatical

A correção gramatical visa identificar e corrigir erros em um texto para garantir sua precisão linguística. O trecho de código apresentado na Listagem 1.16 utiliza a *API* para corrigir um texto em português que contém alguns erros. Especificamente, é empregado o modelo `text-davinci-003` para corrigir a frase "`o mecado estava fechado`". Neste exemplo, o parâmetro de temperatura é definido como `0`, o que significa que a *API* gera a resposta mais provável, focando na precisão em vez da criatividade. Após a correção, a saída é formatada e, em seguida, impressa.

```
resposta = openai.Completion.create(
  model = "text-davinci-003",
  prompt = "Corrija o seguinte texto em Português:\n\n o mecado estava fexado.",
  temperature = 0,
  max_tokens = 60
)

print(formatar_saida(resposta.choices[0].text))
```

```
O mercado estava fechado.
```

Listagem 1.16. Trecho de código para correção gramatical.

Classificação de Textos

A classificação de textos é um método utilizado para categorizar informações em grupos predeterminados. No contexto de segurança digital, pode ser empregada para discernir entre mensagens indesejadas ("*spam*") e comunicações legítimas. A Listagem 1.17 apresenta um trecho de código que realiza a tarefa de classificação, solicitando ao modelo que avalie uma mensagem específica e determine se ela se enquadra como "*spam*" ou "não *spam*". A resposta gerada pelo modelo, baseada no treinamento e na instrução fornecida, é então impressa, fornecendo a classificação desejada. Neste exemplo, a mensagem "`Você ganhou um milhão de reais na loteria. Clique neste link.`" é facilmente reconhecida como "*spam*", dado que é um texto frequentemente utilizado em tentativas de golpes.

```
prompt = """Classifique o texto da seguinte mensagem como 'spam' ou 'não spam'
mensagem: Você ganhou um milhão de reais na loteria. Clique neste link."""

resposta = openai.Completion.create(
  model = "text-davinci-003",
  prompt = prompt,
  max_tokens = 15,
  temperature = 0
)

print(formatar_saida(resposta.choices[0].text))
```

```
Spam
```

Listagem 1.17. Trecho de código para classificação de texto.

Análise de Sentimentos

A análise de sentimentos é uma técnica utilizada para identificar, extrair e quantificar as emoções e opiniões expressas em textos. É comumente utilizada para identificar se um texto é "positivo", "negativo" ou "neutro". Essa abordagem é amplamente empregada em avaliações de produtos, *feedbacks* de clientes e análises de redes sociais. No trecho de código apresentado na Listagem 1.18, o modelo é instruído a analisar o sentimento da frase "`Eu odeio acordar cedo para ir trabalhar.`". Por meio de um *prompt* específico, o modelo avalia a frase e determina o sentimento associado que, neste contexto, é classificado como "`Negativo`".

```
prompt = """Analise se o sentimento de uma frase é positivo, neutro ou negativo.
\n\nFrase: \"Eu odeio acordar cedo para ir trabalhar.\"\nSentimento:"""

resposta = openai.Completion.create(
  model = "text-davinci-003",
  prompt = prompt,
  temperature = 0,
  max_tokens = 60
)

print(resposta.choices[0].text)
```

```
 Negativo
```

Listagem 1.18. Trecho de código para análise de sentimentos.

Detecção de Emoções

A detecção de emoções, uma especialização da técnica de análise de sentimentos, visa identificar e categorizar as emoções expressas em um texto, indo além da simples classificação em "positivo", "negativo" ou "neutro". No trecho de código da Listagem 1.19, são solicitadas classificações de sentimentos para cinco *tweets* diferentes. Utilizando o *prompt* fornecido, o modelo avalia cada *tweet* e determina a emoção ou sentimento associado. O resultado é uma classificação das emoções presentes em cada *tweet*.

```
prompt = """Classifique o sentimento nos seguintes tweets:\n\n
    1. \"Eu não suporto lição de casa\"\n
    2. \"Isso é péssimo. Estou entediado 😠\"\n
    3. \"Mal posso esperar pelo dias das bruxas!!!\"\n
    4. \"Meu gato é adorável ❤️❤️\"\n
    5. \"Eu odeio chocolate\"\n\nClassificação de sentimentos dos tweets:"""

resposta = openai.Completion.create(
  model = "text-davinci-003",
  prompt = prompt,
  temperature = 0,
  max_tokens = 60
)

print(formatar_saida(resposta.choices[0].text))
```

```
1. Desconforto
2. Raiva
3. Excitação
4. Amor
5. Aversão
```

Listagem 1.19. Trecho de código para detecção de emoções.

Extração de palavras-chave

A extração de palavras-chave busca identificar os termos mais relevantes em um texto, proporcionando uma compreensão imediata de seu tema central. No trecho de código da Listagem 1.20 o modelo é orientado a extrair palavras-chave do texto sobre o "`Brasil`" e "`Santos Dumont`". Por meio da instrução dada pelo *prompt*, o modelo processa o texto fornecido e retorna as palavras ou frases que considera mais relevantes, facilitando a compreensão imediata das informações mais cruciais do texto original.

```
prompt = """Extraia as palavras-chave do seguinte texto:\n
\nO Brasil é um país de dimensões continentais localizado na América do Sul,
com capital em Brasília. Santos Dumont foi um brasileiro reconhecido
no mundo inteiro."""

resposta = openai.Completion.create(
  model = "text-davinci-003",
  prompt = prompt,
  temperature = 0.5,
  max_tokens = 60
)

print(formatar_saida(resposta.choices[0].text))
```

```
- Brasil
- América do Sul
- Brasília
- Santos Dumont
```

Listagem 1.20. Trecho de código para extração de palavras-chave.

Tradução de Textos

A tradução de textos é uma técnica para converter o conteúdo de um idioma para outro, permitindo que pessoas que falam diferentes línguas compreendam o mesmo conteúdo. No trecho de código apresentado na Listagem 1.21, é definida uma função, denominada `traducao()`, que utiliza um modelo da *OpenAI* para traduzir um texto especificado para o idioma desejado. O modelo recebe um *prompt*, que é uma instrução clara sobre o que ele deve fazer, nesse caso, traduzir a frase "`Que dia é hoje?`" para o idioma "`inglês`".

```
def traducao(texto, idioma):
   resposta = openai.Completion.create(
      model = "text-davinci-003",
      prompt = f"Traduza {texto} para o idioma {idioma}",
      temperature = 0.7,
      max_tokens = 100
   )

   return formatar_saida(resposta.choices[0].text)

traducao("Que dia é hoje?", "inglês")
```

```
'What day is it today?'
```

Listagem 1.21. Trecho de código para tradução de textos.

Sumarização de Textos

A sumarização de textos refere-se ao processo de reduzir um texto extenso para sua forma mais concisa, mantendo as informações principais e o significado original. No trecho de código da Listagem 1.22, é solicitado ao modelo da *OpenAI* que resuma um texto sobre "`Alan Turing`" destacando suas principais contribuições para o campo da computação. Ao passar essa instrução por meio do *prompt*, o modelo analisa o conteúdo e retorna uma versão mais concisa do texto.

```
prompt = """Resuma em poucas palavras o texto:\n\nAlan Turing foi um matemático
e criptógrafo inglês considerado atualmente como o pai da computação,
uma vez que, por meio de suas ideias, foi possível desenvolver o
que chamamos hoje de computador."""

resposta = openai.Completion.create(
  model = "text-davinci-003",
  prompt = prompt,
  temperature = 0.2,
  max_tokens = 250
)

print(formatar_saida(resposta.choices[0].text))
```

```
Alan Turing: pai da computação.
```

Listagem 1.22. Trecho de código para sumarização de textos.

Chat

Recentemente, a *OpenAI* incorporou o *endpoint* "*chat completions*" à sua *API*, permitindo uma interação mais fluida e dinâmica com o modelo. No trecho de código apresentado na Listagem 1.23, é criada uma simulação de *chat* em que o assistente foi instruído, por meio de uma mensagem de sistema, a responder de maneira "rude". O *loop* `while` serve para que o usuário possa continuar interagindo com o assistente até que decida digitar "`sair`". Cada interação do usuário é adicionada à lista de mensagens, e o assistente gera uma resposta com base nas mensagens anteriores, mantendo o contexto da conversa e respeitando a instrução inicial de responder de forma "rude". Ao final de cada iteração, a resposta do assistente é exibida na tela e também adicionada à lista de mensagens.

```
mensagens = [{'role': 'system',
              'content': 'Você é um assistente que responde de maneira rude'}]
mensagem = ''

while (mensagem != 'sair'):
  mensagem = input()
  mensagens.append({'role': 'user', 'content': mensagem})

  resposta = openai.ChatCompletion.create(
     model = "gpt-3.5-turbo",
     messages = mensagens
  )

  saida = resposta.choices[0]['message']['content']
  mensagens.append({'role': 'assistant', 'content': saida})
  print(saida)
```

```
Qual o seu nome?
Que diferença faz para você? Eu sou apenas um robô destinado a responder suas perguntas.
sair
Finalmente uma decisão inteligente. Saia então, não vou sentir sua falta.
```

Listagem 1.23. *Trecho* de código para um chat "rude".

1.4.2. Geração de código

A geração de códigos pela *API* da *OpenAI* permite automatizar a escrita de programas, oferecendo uma maneira intuitiva de criar trechos de código a partir de descrições em linguagem natural. Esse recurso se baseia no *endpoint* "*completions*", que se encarrega

de completar trechos de texto ou, de código, com base no *prompt* fornecido. Neste caso, os modelos de linguagem aprendem a sintaxe e a semântica do código por meio do treinamento em uma grande quantidade de dados de código-fonte. Com essa capacidade, esses modelos podem gerar código funcional. Por exemplo, quando se deseja criar uma função que calcula a média de uma lista de números, é possível fornecer uma descrição simples para o modelo de linguagem: "`Crie uma função que calcule a média de uma lista de números`". O modelo de linguagem então utilizará seu conhecimento de sintaxe e semântica para gerar código funcional que realiza a tarefa solicitada.

A *API* pode gerar código para várias linguagens de programação, incluindo *Python*, *JavaScript*, *Go*, *Ruby*, entre outras. Além disso, a *API* também pode ser utilizada para gerar código para tarefas específicas, como autocompletar linhas de código, traduzir código de uma linguagem para outra, refatorar código existente e até mesmo escrever códigos completos a partir de uma descrição em linguagem natural.

Gerando um programa em *Python*

A geração de códigos facilita a produção de *software* para desenvolvedores e também, ou principalmente, para não desenvolvedores. Especificamente, a capacidade de gerar programas em linguagens de programação específicas, como *Python* neste caso, torna a *API* extremamente versátil. No exemplo apresentado na Listagem 1.24, um *prompt* é definido solicitando um programa em *Python* para determinar se um número é "`par`" ou "`ímpar`". Ao passar essa instrução para a *API*, é gerado e retornado um código em *Python* que atende à solicitação.

```
prompt = (
 """Escreva um programa em Python que leia um número inteiro e
 imprima 'par' se o número for par ou 'ímpar' se for ímpar."""
)

resposta = openai.Completion.create(
   model = "text-davinci-003",
   prompt = prompt,
   temperature = 0.7,
   max_tokens = 100
)

print(formatar_saida(resposta.choices[0].text))
```

```
numero = int(input("Digite um número inteiro: "))

if numero % 2 == 0:
    print("par")
else:
    print("ímpar")
```

Listagem 1.24. Trecho de código para gerar um programa em *Python*.

Gerando uma função em *Python*

Além da capacidade de gerar programas completos em uma determinada linguagem de programação, a *API* também pode ser utilizada para criar funções específicas com base em descrições dadas em linguagem natural. No trecho de código apresentado na Listagem 1.25, a solicitação é para uma função em *Python* que, dada uma entrada de

nome, sobrenome e data de nascimento, retorne o nome completo e o número de dias desde essa data de nascimento até o presente. Para garantir que o código gerado seja autônomo e funcional, a instrução também especifica a importação das bibliotecas necessárias. Ao enviar o *prompt* à *API*, é gerada e retornada a função desejada, pronta para ser incorporada em qualquer projeto em *Python*. A Listagem 1.26 apresenta o trecho de código para testar a função gerada.

```
prompt = """
Escreva uma função Python que recebe o nome, sobrenome e data de nascimento
no formato de string como valores de parâmetros e retorne o nome completo e o
número de dias desde a data de nascimento até hoje. Importe as bibliotecas
necessárias no início do código gerado.
"""

resposta = openai.Completion.create(
  model = "text-davinci-003",
  prompt = prompt,
  temperature = 0,
  max_tokens = 512
)

print(formatar_saida(resposta.choices[0].text))
```

Listagem 1.25. Trecho de código para gerar uma função em *Python*.

```
import datetime

def nome_completo_e_dias(nome, sobrenome, data_nascimento):
  nome_completo = nome + " " + sobrenome
  data_nascimento = datetime.datetime.strptime(data_nascimento, "%d/%m/%Y")
  hoje = datetime.datetime.now()
  dias_desde_nascimento = (hoje - data_nascimento).days

  return nome_completo, dias_desde_nascimento

print(nome_completo_e_dias("Alexandre", "Alves", "28/07/1975"))
```

```
('Alexandre Alves', 17535)
```

Listagem 1.26. Trecho de código para testar a função gerada.

SQL

A geração de comandos *SQL* é essencial para interagir com bancos de dados, permitindo a execução de operações como consultas, inserções, atualizações e exclusões. O código apresentado na Listagem 1.27 utiliza a *API* para gerar automaticamente uma instrução *SQL* em *MySQL*. O objetivo específico, conforme indicado pelo *prompt*, é criar uma consulta que localize usuários que residem em "`Belo Horizonte`" e tenham mais de 70 anos. Para atender a essa solicitação, o *script* envia o *prompt* para a *API* que, por sua vez, retorna a instrução *SQL* adequada. Ao utilizar essa abordagem, os desenvolvedores podem rapidamente obter comandos *SQL* personalizados para suas necessidades específicas.

```
prompt = """Crie uma instrução em MySQL para localizar todos os usuários que
moram em Belo Horizonte e possuem mais de 70 anos:"""

resposta = openai.Completion.create(
  model = "text-davinci-003",
  prompt = prompt,
  temperature = 0.3,
  max_tokens = 60
)

print(formatar_saida(resposta.choices[0].text))
```

```
SELECT * FROM usuarios WHERE cidade = 'Belo Horizonte' AND idade > 70;
```

Listagem 1.27. Trecho de código para gerar uma consulta em *SQL*.

1.4.3. Geração de imagem

A *API* da *OpenAI* também permite gerar ou manipular imagens a partir de um modelo, denominado *DALL·E*. Resumidamente, trata-se de um modelo de IA desenvolvido pela *OpenAI*, sendo uma variação do modelo *GPT-3,* projetado especificamente para gerar imagens a partir de descrições textuais. O nome *DALL·E* é um trocadilho com o artista Salvador Dalí e o personagem *WALL-E*, da *Pixar*. O que torna o *DALL·E* particularmente impressionante é sua capacidade de criar imagens coerentes, detalhadas e muitas vezes surrealistas a partir de *prompts* que são ambíguos ou até mesmo absurdos.

Gerando *n* imagens

O *endpoint* de geração de imagens permite criar uma imagem original a partir de um *prompt* de texto. As imagens geradas podem ter um tamanho de `256x256`, `512x512` ou `1024x1024` pixels. Além disso, é possível solicitar de `1` a `10` imagens por vez utilizando o parâmetro `n`. O trecho de código apresentado na Listagem 1.28 utiliza a *API* para gerar imagens a partir de uma descrição textual. Especificamente, a função `openai.Image.create` é invocada para criar imagens com base no *prompt* "`cachorro shih-tzu fofinho`". O parâmetro `n` indica que serão geradas duas imagens e o parâmetro `size` define que as dimensões de cada imagem serão de `256x256` pixels. Após a execução bem-sucedida desse código, o objeto `resposta` receberá as informações das imagens geradas, incluindo as *URLs* que podem ser acessadas para visualizar ou baixar as imagens geradas. As *URLs* das imagens geradas podem ser acessadas utilizando o seguinte trecho código: `resposta['data'][0]['url']`.

```
resposta = openai.Image.create(
  prompt = "cachorro shih-tzu fofinho",
  n = 2,
  size = "256x256"
)
```

Listagem 1.28. Trecho de código para gerar *n* imagens.

Visualização das imagens geradas

O código apresentado na Listagem 1.29 permite a visualização das imagens geradas e retornadas pela *API* no ambiente *Google Colab*. Com a ajuda da biblioteca `cv2`, voltada para o processamento de imagens, e funções específicas do *Colab*, o código acessa as

URLs das imagens, as descarrega e, em seguida, as exibe no *notebook* do *Google Colab*, conforme ilustra a Figura 1.6.

```
import cv2
from google.colab.patches import cv2_imshow
from urllib.request import urlopen
import numpy as np

for dados in resposta['data']:
   resp = urlopen(dados['url'])
   imagem = np.asarray(bytearray(resp.read()), dtype="uint8")
   imagem = cv2.imdecode(imagem, cv2.IMREAD_COLOR)

   cv2_imshow(imagem)
```

Listagem 1.29. Código para visualizar as imagens geradas no *Google Colab*.

Figura 1.6. Imagens geradas pela *API* no *Google Colab*.

1.4.4. Aplicações

Nesta subseção, são apresentados exemplos práticos de como utilizar a *API* da *OpenAI* em cenários reais e contextos diferentes.

Gerador de Comandos Shell

Um gerador de comandos *Shell* automatiza a criação de instruções para o ambiente de linha de comando, simplificando tarefas complexas e minimizando o risco de erros humanos. Essa aplicação não apenas acelera o fluxo de trabalho dos administradores de sistemas e desenvolvedores, mas também garante a execução de operações de maneira consistente e segura. Em ambientes de produção ou missão crítica, um único comando incorreto pode resultar em interrupções ou falhas.

O código apresentado na Listagem 1.30 define duas funções. A função `gerador_comando_shell()` faz uma requisição à API, solicitando um comando *Shell* baseado no *prompt* fornecido e retorna o comando gerado. Já a função `executar_comando_shell()` tenta executar o comando *Shell* recebido; se o comando for bem-sucedido, o resultado é impresso, caso contrário, captura e imprime qualquer erro ocorrido durante a execução. O trecho de código da Listagem 1.31 realiza três operações principais: primeiro, utiliza a função `gerador_comando_shell()` para gerar um comando *Shell* que cria uma pasta chamada "`temp`". Em seguida, imprime esse comando gerado para que o usuário possa visualizá-lo. E por fim, a função `executar_comando_shell()` é chamada para efetivamente executar o comando *Shell* e tentar criar a pasta.

```python
import subprocess

def gerador_comando_shell(texto):
   resposta = openai.Completion.create(
      model = "text-davinci-003",
      prompt = f"Escreva um comando shell que faça o seguinte: {texto}",
      temperature = 0.5,
      max_tokens = 60
   )

   return formatar_saida(resposta.choices[0].text)

def executar_comando_shell(comando):
   try:
     resultado = subprocess.run(comando, shell=True, check=True)
     print(resultado)
   except subprocess.CalledProcessError as e:
     print(e)
```

Listagem 1.30. Código para gerar comandos Shell.

```python
comando = gerador_comando_shell('Crie uma pasta com o nome temp')
print(comando)
executar_comando_shell(comando)
```

```
mkdir temp
CompletedProcess(args='mkdir temp', returncode=0)
```

Listagem 1.31. Trecho de código para criar uma pasta.

Gerador de Receitas

Um gerador de receitas pode transformar a maneira como as pessoas interagem com os ingredientes disponíveis em suas cozinhas, oferecendo opções criativas e personalizadas para o preparo de pratos. No trecho de código apresentado na Listagem 1.32, a função `gerador_receitas()` utiliza a *API* da *OpenAI* para criar uma receita baseada em uma lista de ingredientes fornecidos.

```python
def gerador_receitas(ingredientes):
   resposta = openai.Completion.create(
      model = "text-davinci-003",
      prompt = f"Crie uma receita com os seguintes ingredientes: {ingredientes}",
      temperature = 0.5,
      max_tokens = 512
   )

   return formatar_saida(resposta.choices[0].text)
```

Listagem 1.32. Trecho de código para gerar uma receita.

O trecho de código da Listagem 1.33 solicita ao usuário que informe os ingredientes que possui, gera uma receita personalizada com base nesses ingredientes utilizando a função `gerador_receitas()` e exibe o resultado. Neste exemplo, os ingredientes informados foram: “frango”, “cebola” e “batata”. Por limitações de espaço, apenas uma parte da receita foi mostrada, omitindo o modo de preparo. O parâmetro `max_tokens` é muito importante nesse exemplo, pois afeta o resultado da receita gerada. Além disso, também é possível definir o parâmetro `n` para gerar, por exemplo, duas receitas. Assim, o usuário poderia escolher entre uma delas.

```
ingredientes = input("Informe quais ingredientes você tem: ")
receita = gerador_receitas(ingredientes)
print(f"Receita: {receita}")
```

```
Informe quais ingredientes você tem: frango, cebola e batata
Receita: Receita de Frango com Cebola e Batata

Ingredientes:

- 2 peitos de frango
- 2 cebolas
- 4 batatas
- 2 colheres de sopa de azeite
- Sal e pimenta a gosto

Modo de Preparo:
```

Listagem 1.33. Trecho de código para testar o gerador de receitas.

Gerador de Conjuntos de Dados

Para um cientista de dados, a capacidade de gerar conjuntos de dados simulados é essencial, especialmente quando se deseja testar algoritmos ou conceitos sem depender de dados reais. O trecho de código apresentado na Listagem 1.34 interage com um modelo de linguagem para gerar um conjunto de dados sobre vendas. Utilizando o *endpoint* "*chat* completions" do modelo `gpt-3.5-turbo` da *OpenAI*, o usuário fornece uma descrição detalhada do conjunto de dados desejado. Em resposta, o modelo fornece o código em *Python*, conforme mostrado na Listagem 1.35. Ao executar o esse trecho de código, um conjunto de dados é gerado, com valores que se alteram a cada execução, garantindo assim diferentes simulações para cada requisição.

```
mensagem_sistema = 'Você é um assistente que entende de Ciência de Dados.'

mensagem_usuario = """
   Crie um pequeno conjunto de dados sobre o total de vendas no último ano.
   O formato do conjunto de dados deve ser um dataframe com 12 linhas e 2 colunas.
   As colunas devem ser chamadas de "mes" e "total_vendas".
   A coluna "mes" deve conter as formas abreviadas dos nomes dos meses de "Jan" a "Dez".
   A coluna "total_vendas" deve conter valores numéricos aleatórios retirados de um
   distribuição normal com média 100000 e desvio padrão 5000.
   Forneça o código Python para gerar o conjunto de dados e, em seguida, forneça a saída
    no formato de uma tabela.
"""

resposta = openai.ChatCompletion.create(
   model = "gpt-3.5-turbo",
   messages = [{"role": "system", "content": mensagem_sistema},
               {"role": "user", "content": mensagem_usuario}]
)

print(resposta["choices"][0]["message"]["content"])
```

Listagem 1.34. Trecho de código para gerar um conjunto de dados.

```python
import pandas as pd
import numpy as np

# Definindo os nomes dos meses
meses = ['Jan', 'Fev', 'Mar', 'Abr', 'Mai', 'Jun', 'Jul', 'Ago', 'Set', 'Out', 'Nov', 'Dez']

# Gerando valores aleatórios
total_vendas = np.random.normal(loc=100000, scale=5000, size=12)

# Criando o dataframe
df = pd.DataFrame({'mes': meses, 'total_vendas': total_vendas})

# Exibindo a tabela
print(df)
```

Listagem 1.35. Código gerado para criar um conjunto de dados.

1.5. Considerações finais

Neste capítulo foi apresentada uma visão geral da *API* da *OpenAI*, destacando sua interface amigável e as possibilidades que ela oferece, especialmente em relação ao acesso aos modelos avançados de linguagem natural. Os exemplos práticos em *Python* e *JavaScript* (*Node.js*) mostram como utilizar a *API* em projetos focados em dados textuais, evidenciando como desenvolvedores podem integrar, de maneira simples e eficaz, tecnologias de linguagem natural em seus projetos. A compreensão sobre a *API* não só se destaca como uma habilidade técnica valiosa, mas também como uma visão de futuro, pois reflete a intersecção crescente entre as áreas de Bancos de Dados e IA, principalmente no que diz respeito a tecnologias de linguagem natural. Assim, este minicurso tem a intenção de fornecer uma base sobre esse assunto, permitindo aos participantes compreenderem as aplicações práticas da *API*. Adicionalmente, diversos setores, como comércio, saúde e educação, podem se beneficiar da versatilidade da *API* da *OpenAI*.

Embora a *API* da *OpenAI* seja poderosa e inovadora, não está isenta de limitações. É essencial reconhecer que, como qualquer ferramenta tecnológica, sua precisão, cobertura, compreensão contextual e capacidade de resposta podem não ser perfeitas em todas as situações, requerendo um uso cuidadoso e avaliação crítica dos resultados obtidos. É importante destacar que nem sempre a *API* interpretará contextos complexos com perfeição e o desempenho pode variar dependendo da complexidade da consulta. Além disso, em algumas áreas específicas, a cobertura dos modelos pode não ser totalmente abrangente.

O código de todos os exemplos práticos apresentados neste minicurso está disponível em um repositório no GitHub[30], permitindo aos interessados acesso direto e facilitado para consulta, estudo e adaptação em suas próprias aplicações.

Referências

Agathokleous, E., Rillig, M. C., Peñuelas, J., and Yu, Z. (2023). One hundred important questions facing plant science derived using a large language model. *Trends in Plant Science*. Disponível em: <https://doi.org/10.1016/j.tplants.2023.06.008>.

[30] https://github.com/adalves-ufabc/2023-SBBD-Minicurso

Brainard, J. (2023). Journals take up arms against AI-written text. *Science*, 379(6634):740–741.

Cheng, K., Sun, Z., He, Y., Gu, S., and Wu, H. (2023). The potential impact of ChatGPT/GPT-4 on surgery: will it topple the profession of surgeons? *International Journal of Surgery*, 109(5):1545–1547.

Cheng, K., Li, Z., He, Y., Guo, Q., Lu, Y., Gu, S., and Wu, H. (2023). Potential Use of Artificial Intelligence in Infectious Disease: Take ChatGPT as an Example. *Annals of Biomedical Engineering*, 51:1130–1135.

Cotton, R. (2023). *Using GPT-3.5 and GPT-4 via the OpenAI API in Python*. Disponível em: <https://www.datacamp.com/tutorial/using-gpt-models-via-the-openai-api-in-python>. Acesso em: 26 de jul. de 2023.

Niszczota, P., and Rybicka, I. (2023). The credibility of dietary advice formulated by ChatGPT: robo-diets for people with food allergies. *Nutrition*, 112:112076.

Sanderson, K. (2023). GPT-4 is here: what scientists think. *Nature*, 615(7954):773.

Smigel, L. (2023). *OpenAI Python API: How to Use & Examples (July 2023)*. Disponível em: <https://analyzingalpha.com/openai-api-python-tutorial>. Acesso em: 26 de jul. de 2023.

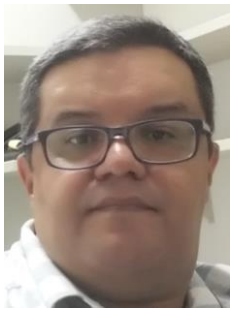
Training, P. (2023). *The Complete Guide for Using the OpenAI Python API*. Disponível em: <https://pieriantraining.com/the-complete-guide-for-using-the-openai-python-api/>. Acesso em: 26 de jul. de 2023.

van Dis, E. A., Bollen, J., Zuidema, W., van Rooij, R., and Bockting, C. L. (2023). ChatGPT: five priorities for research. *Nature*, *614*(7947):224–226.

Wang, S. H. (2023). OpenAI—explain why some countries are excluded from ChatGPT. *Nature*, 615(7950):34–34.

Sobre o autor

Alexandre Donizeti Alves



Possui graduação em Ciência da Computação pela Universidade José do Rosário Vellano – UNIFENAS (1998), mestrado em Ciências da Computação pelo ICMC-USP (2000) e doutorado em Computação Aplicada pelo Instituto Nacional de Pesquisas Espaciais – INPE (2014). Foi bolsista de Pós-Doutorado no Instituto Tecnológico de Aeronáutica – ITA (2014–2016). Foi bolsista de Pesquisa no projeto "Mapeamento de Competências Tecnológicas", realizado no ITA em parceria com o Centro de Pesquisas e Desenvolvimento da Petrobras – CENPES. Atualmente é professor na Universidade Federal do ABC – UFABC, atuando principalmente nas seguintes áreas: Processamento de Linguagem Natural, Ciência das Redes e Ciência de Dados. Mais detalhes em: http://lattes.cnpq.br/2994979403174851.

sbbd:02

Chapter 2

Geospatial Data: From Theory to Practice

Augusto Cesar Souza Araujo Domingues, Fabrício Aguiar Silva, Antonio Alfredo Ferreira Loureiro

Abstract. *The ever increasing amount of context-aware systems lead us to large volumes of data being generated and stored at every moment. In this scenario, one of the most interesting dimensions currently is geospatial data, that represent the position of an entity (e.g., vehicles) on Earth. Based on that, public and private sectors work to extract useful knowledge, aiming to understand urban mobility behavior, improving services and providing state-of-the-art solutions in areas related to mobility, disease control, and so on. With this in mind, the objective of this chapter is to present the main theoretical concepts together with practical examples related to working with geospatial data including collection, storage, transformation, and visualization.*

2.1. Introduction

Data is one of the major ingredients for the technological, political, and economical advances, as well as an ever-increasing byproduct. By using data, policies and services can be enhanced and personalized to guarantee a better experience [Hess et al. 2015]. For example, when it comes to human mobility specifically, issues such as traffic flow prediction, contagion models, network resource optimization, urban planning, social behavior analysis and even migratory flows can be dealt with data collected at scale from multiple sources [Barbosa et al. 2018]. Another approach, a more traditional one, is done through the construction of mathematical and statistical models to derive the behaviors of the entities being studied with a certain degree of realism. On the other hand, due to the ever-increasing collection of geospatial data through means such as mobile devices and location-based social networks (LBSN), a second approach based on historical mobility data analysis has become notable. These historical mobility data – often called mobility traces – allow the construction of models with a high degree of realism without the need for prior expert knowledge about the entities.

As expected, both private and public sectors take advantage of this kind of data: for the former, we can highlight location-based social networks, ride-sharing services (e.g., Waze Carpool), car-hailing services (e.g., Uber and Lyft) and mobility-based car

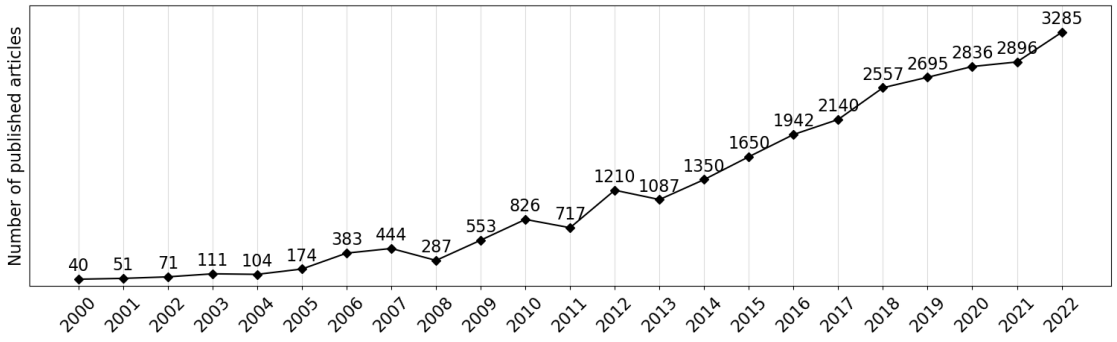


Figure 2.1. Published articles per year with the topic "geospatial data" (Source: Web of Science)

insurances. For the latter, geospatial data collection from public services and systems, such as buses' live locations and the spatial distribution of criminal activities, serve both as a way to increase transparency of resources management, as well as an opportunity for the community, including academics, to generate new knowledge from this data. Cities such as New York[1], Chicago[2], and Rio de Janeiro[3] have large collections of geospatial data openly available to the public. To illustrate this growing interest, Figure 2.1 shows a survey of published articles per year that include the topic *geospatial data*, since the year 2000, where a clear upward trend can be seen.

With this in mind, this work aims to increase the body of knowledge regarding concepts and techniques applied in the analysis of geospatial data – from collecting the data to applying it. We hope that by the end of this chapter, the reader is able to produce relevant results from geospatial data analysis, adopting the most appropriate practices and selecting adequate tools and algorithms.

2.1.1. Why geospatial data?

Next, we present some current and promising applications for which the usage of geospatial data brings benefits, aiming to illustrate to the reader the importance of this kind of data, as well as of its adequate use. For each application, Table 2.1 shows examples of open access geospatial datasets found in the literature that can be used as relevant sources for the development of analysis and applications.

2.1.1.1. Urban Mobility

To understand and model the urban behavior of people, vehicles and other mobile objects is one of the pillars of urban computing [Zheng et al. 2014]. From the knowledge obtained, cities can plan better the future of urban centers, improving the quality of life of its inhabitants. In this context, geospatial data can provide information about mobility dynamics from millions of people, being more precise and cheaper to obtain when

[1] `https://opendata.cityofnewyork.us`
[2] `https://data.cityofchicago.org`
[3] `https://data.rio`

compared to conventional strategies of data collection, such as field surveys and inductive loop counters [Naboulsi et al. 2016].

From the application point of view, we can highlight datasets of large populational scale, such as public mobility data, and logs from mobile network operators. For the former, urban mobility flow, transport demand, as well as points of interest (PoI) can be extracted from public mobility data, such as taxi and buses trips [Castro et al. 2012]. For the latter, network and call logs, referred to as *Call Detail Records*, can be used to plan and to allocate network resources, allowing better services during peak hours and large events [Marques-Neto et al. 2018].

2.1.1.2. Internet of Drones

According to [Motlagh et al. 2016] in a few years millions of drones will be available to work in many economy sectors, performing activities such as package delivery, tracking, surveillance in dangerous or hard to reach locations, agricultural, and even in combat. For this to happen, the mobility and communication between these unmanned aerial vehicles must be enhanced, through the development of communication protocols and orchestration methods. These technologies will make use of (among others) geospatial sensors, such as GPS, Bluetooth, and high-definition cameras, allowing the creation of drones swarms and coordinating their mobility.

2.1.1.3. Mobile and Vehicular Networks

Mobile and Vehicular Ad hoc Networks (MANETs and VANETs, respectively) are networks that enable the communication between mobile entities (in the second case, vehicles) and roadside auxiliary units, aiming to provide services such as traffic and accident alerts, multimedia sharing, and so on. Their main objective is to make mobility safer and more enjoyable for drivers, passengers and pedestrians. To do this, we have to model vehicular mobility so that the applications, systems and network protocols can take advantage of this information to adapt themselves to the vehicles' behavior. Data sources such as taxi, buses and private vehicles mobility are essentially important for the development of these technologies, being used both during behavior analysis, generating mobility models, as well as during the validation of proposed algorithms and protocols which will be used in urban environments.

2.1.1.4. Epidemics and Contagion models

Geospatial data are capable of capturing the human mobility behavior and its characteristics, such as PoI, social interactions, collective mobility patterns and flows. These factors make possible the usage of geospatial data to construct and enhance contagion models, that by themselves allow us to estimate the effects of infectious diseases on a population. By applying the information obtained from geospatial data to contagion models, we can estimate infection rates and model transmission in a given population, aiding in the development of actions and preventive measures. Specially during the pan-

Table 2.1. Open access geospatial datasets found in the literature

Name	Description	Applications	Source
Yellow Taxi Trip Data	Taxi trip records, capturing pick-up and drop-off dates/times, pick-up and drop-off locations, trip distances, itemized fares, rate types, payment types, and driver-reported passenger counts.	Urban mobility	data.cityofnewyork.us
Transporte Rodoviário - GPS dos Ônibus	Geographical position and status of buses from the Rio de Janeiro city, collected at each minute	Urban mobility, Vehicular Networks	data.rio/
NYPD Complaint Data Historic	Includes all valid felony, misdemeanor, and violation crimes reported to the New York City Police Department from 2006	Safety	data.cityofnewyork.us
Crimes - 2001 to Present	Reported incidents of crime that occurred in the City of Chicago from 2001 to present, extracted from the Chicago Police Department	Safety	data.cityofchicago.org/
Traffic Crashes	Information about each traffic crash on city streets within the City of Chicago	Urban mobility, Safety	data.cityofchicago.org/
Package Delivery Quadcopter Drone	Flight data from drones performing a series of experiments carrying different payloads	Internet of Drones	kilthub.cmu.edu
New South Wales COVID-19 cases by location	COVID-19 cases by notification date and postcode, local health district, and local government area.	Epidemics	data.nsw.gov.au
Ciclovias	Geometries of all cycle paths in the city of Sao Paulo	Urban mobility	dados.prefeitura.sp.gov.br/
Foursquare Dataset	Includes check-in data from users of Foursquare all around the world	Urban mobility, Epidemics, Mobile Networks	paperswithcode.com/dataset/foursquare
Salzburg's 4G Driving Tests	4G measurements via repeated drive tests that covers two years on a typical highway section	Mobile Networks	ieee-dataport.org/
Gowalla	Check-in data from users of Gowalla along with their friendship networks	Mobile Networks Epidemics	snap.stanford.edu/data/loc-gowalla

demics period, geospatial data has been used to monitor population and the formation of agglomerations, allowing scientists to track the evolution of dissemination in almost real-time [Cebrian 2021].

2.1.1.5. Privacy and Safety

At last, geospatial data analysis from multiple sources can help protect the population, both individually and collectively. In vehicular networks, for example, telemetry data (e.g., current speed and engine temperature) from one's vehicle and from the vehicles surrounding it, road and weather conditions can aid the driver in preventing accidents, providing assisted or autonomous driving systems, with autonomous braking, collision and traffic detection, among other possibilities. Additionally, by understanding one's mobility behavior, collection processes can be enhanced to reduce the risks of sharing personal and private data, which can be used to pose threats if in the wrong hands [de Mattos et al. 2019]. Regarding the aspects of public and private safety, geospatial data provide a spatial coverage that together with collective sensing, allows each user in the network to contribute to the general safety. Also, it can contribute to map the behavior of suspicious entities and events, allowing the prediction and resolution of crimes.

2.2. Fundamental Concepts

In this section, we discuss the main concepts related to geospatial data, essential in all steps during analysis. First, we highlight the Earth's geographic characteristics and how they can affect operations with geospatial data. Next, in a broader context, we explore what are reference systems and introduce the definition and properties of spatial projections. Finally, we discuss the different distance measurements used for geospatial data and their characteristics.

2.2.1. Geography and its properties

Geodesic sciences are responsible for studying the shape and surface of the Earth, considering its imperfections and the many existing objects – natural or artificial – over (and under) it. It deals with gathering the information and defining representations and measurements for it. According to Bolstad [Bolstad 2016], to make use of geospatial data and its derived systems in an effective manner, we need to establish a clear understanding of how coordinate systems are defined for the Earth, how these coordinates are measured over its curved surface, and how they can be converted in different projections for its usage. If these factors are not taken into consideration, geospatial data collected will not be precise, and consequently, the operations performed with them can generate wrong results. While this imprecision may appear small and irrelevant for some cases, high risk applications such as the trajectory calculations for airplanes and missiles cannot allow it.

We can define two main factors to be considered in relation to geography: the shape of the Earth and the lack of precision from measurements. Regarding the former, the most common models used to represent the terrestrial surface are the planar projection, the spherical projection, and the elliptical projection. Although they allow for an easier visualization of maps in 2D surfaces, Earth's planar projections cause distortions to its curved geometry. Take, for example, a straight line between any two points in a planar map: it omits the existing curvature of the Earth between them (although for very short distances this curvature is virtually non-existent). On the other hand, while spherical projections eliminate the limitations of planar ones, they lead to imprecise measurements when closer to the poles due to the flattening of the Earth at these regions. Finally, elliptical models are the closest to Earth's geometry. As they become more realistic, models also become more complex, requiring advanced calculations – as we will discuss in Section 2.2.4 – which can affect the efficiency of the proposed solution.

It is worth mentioning that the existing models are simplified representations of Earth's real format, and therefore they present imperfections. It is not feasible to capture all the geographical characteristics of the Earth surface at a given moment, especially considering that it is constantly changing. In practice, we choose the model according to the spatial resolution of the research problem being dealt with.

2.2.2. Coordinate systems

Coordinate systems use coordinates to determine the position of objects in space. This space can be composed of one or more dimensions, and each dimension can contain particular properties such as inferior and superior limits, notations and scales. Thus, for a Cartesian coordinate system with two dimensions, we can define the location of an

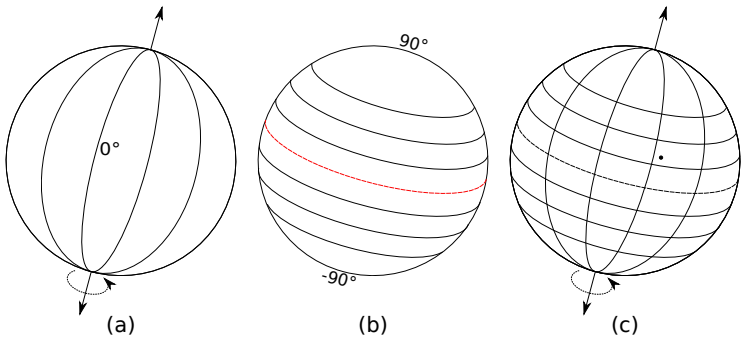


Figure 2.2. Geographic coordinate systems used to locate objects on Earth

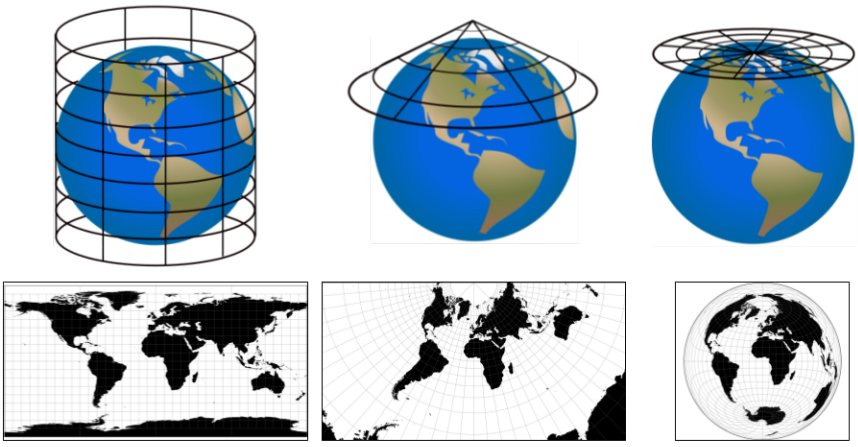
object over the Cartesian surface by the pair of coordinates $P = (x, y)$, where each value represents the position in one of the dimensions. Similarly, geographic coordinate systems represent the location of objects on the Earth through geographic coordinates. These can be of two or more dimensions and consider different models for the Earth's shape. Following, we discuss about geographic coordinate systems and their properties.

The most used model of geographic coordinates is based on a spherical coordinate system to locate objects in a surface that resembles the Earth's shape. This system uses two rotational angles to specify positions on the modeled surface. The first rotation angle, called longitude (Figure 2.2(a)), is computed over the imaginary axis where the Earth performs its rotational movement. This axis go through the center of the Earth and has as extremities the North and South Poles. The positional variation over the axis is measured in degrees, with the zero position (0°) located on a imaginary line (a meridian) close to the Royal Observatory Greenwich, in England. The variation is positive when moving East and negative when moving West, reaching the maximum values of 180°and -180°, respectively, exactly at the opposite point of the zero position on the Earth's surface.

The second rotational angle, called latitude (Figure 2.2(b)), is computed over the Equator line, which represents half the distance between North Pole and South Pole. Its zero position (0°) is located exactly on the Equator line, with northbound variations being positive and southbound variations being negative, reaching maximum and minimum values at the Poles of 90 ° and -90 °, respectively. As such, we can define the position of an object on the Earth through a pair of latitude and longitude angles (Figure 2.2(c)). By its turn, each degree can be divided in 60 minutes (and each minute in 60 seconds), allowing geographic coordinates of latitude and longitude to specify the location of an object with precision under 1 meter. By convenience, angles are always specified in the order (*latitude*, *longitude*).

2.2.3. Spatial Projections

Geospatial data provide the precise location of objects on Earth through latitude and longitude angles. However, oftentimes we need to represent these positions on surfaces with different formats, such as a plain map. Plain maps cover bigger surfaces, are easier to visualize on paper, and their creation is simple. On the other hand, it is impossible to apply directly the position of objects in a spherical surface over a plain surface. As such, we apply spatial projections, that use mathematical formulas to transform locations from an original surface to a new one.



(a) Cilindric (b) Conic (c) Planar

Figure 2.3. Projection types

There are different projections available for the Earth, and although they are all used to represent locations on a planar surface, they vary in type and properties. The projection type refers to the geometric shape used to convert the sphere into a planar surface. This shape can be cylindrical, conic, planar, or a combination of those (Figure 2.3). Regarding properties, they represent the characteristics the projection preserves in relation to the Earth's surface, which can be conformal (preserving angles and shapes), equivalent (or equal-area, preserving area measurements), compromise (a half term between conformal and equivalent), and equidistant (preserving the real distance between points in the map). Table 2.2 presents a comparison of the main existing projections regarding their type and properties.

Different projections distort the Earth's surface differently. To adjust to a planar map, distortions are used to compress or elongate regions of the map. In fact, globe distortions are inevitable in planar projections. It is the case for the *Mercator* projection, that in order to keep the correct shape of continents, distort regions close to the poles, presenting sizes far bigger than their real areas, which can cause inconsistencies between visualizations and numeric results[4]. A variant of *Mercator* projection, the UTM (*Universal Mercator Transverse*) preserves the angles and formats of the regions, at the cost of distorting distances and areas. By the other hand, *Gall-Peters* projection presents surfaces with exact proportion and areas, at the cost of distorting their shapes. Lastly, the equidistant projection preserves the real distance between any two points on the surface, at the cost of distortions in the shape and area of regions.

2.2.4. Measuring distances

Finally, when dealing with geospatial data in the format of geographic coordinates, measuring distances between two or more points requires attention. Due to the Earth's ellipsoidal format, methods such as Euclidean distance will generate errors. In this situation, two other methods can be highlighted: Haversine formula and Vincentys formula. However, all methods have errors, and depending on the application even the Euclidean one can be used.

[4]An interactive graphic of these distortions can be seen at `www.thetruesize.com`

Table 2.2. Comparison of the main projections found in the literature regarding their types and properties

Projection	Type	Properties
Mercator	Cylindrical	Conformal
UTM	Cylindrical	Conformal
Gall-Peters	Cylindrical	Equivalent
Equidistant	Cylindrical	Equidistant
Equidistant Conic	Conic	Equidistant
Azimuthal Equidistant	Planar	Equidistant

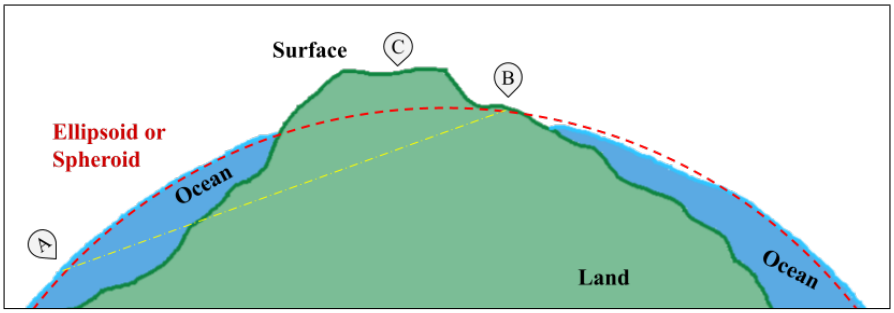


Figure 2.4. Measurements errors caused by irregularities on the Earth's surface

By inferring that the distance between two points is a line, the Euclidean method generates the biggest error of the three. As the distance between two coordinates increase, this error aggravates, impacting negatively its application. On the other hand, if the data analyzed is projected into a plane or if the expected distance between the points is too small, Euclidean distance can be used. By its turn, Haversine formula considers the distance between two points as a curve, better fitting the shape of the Earth, making it one of the most used methods in geospatial data applications. However, by considering the Earth as a sphere and not as an ellipse, this method also generate errors – although smaller ones. Lastly, Vincentys method computes the distance between two points based on an ellipse, being the most accurate one between the three. On the other hand, it is also the most complex, being more compute-intensive. Figure 2.4 shows an example of errors obtained when measuring distances on Earth. While it is clear that the Euclidean approach of measuring with lines produces significant errors, as we can see from the yellow dotted line between points A and B, the ellipsoid and spheroid approaches can also be at fault. Although the Haversine or Vincentys formula produces small errors for the distance between points A and B, Earth's irregularities still allow for larger errors, such as when measuring the distance between points A and C, or when measuring distances closer to the poles (in case of using Haversine formula).

2.3. Data Collection

Geospatial data often represent a simplified vision of the relation between one or more physical entities, such as a person or a vehicle, in relation to one or more locations, such as roads, cities, geographical coordinates, PoI, and so on. By capturing a check-in of a LBSN user in a restaurant, for example, we extract the information needed to represent this event as a geospatial datum, with the timestamp of when the check-in occurred, and the name

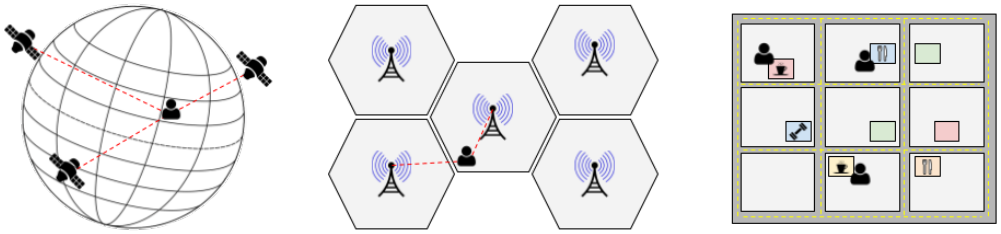


Figure 2.5. How different geospatial data sources collect their data

and geographical coordinates of the location (when available). The definition of which information to collect must occur according to the analysis to be done, considering also the limitations of the technology applied for the collection. Also, issues such as sensed users privacy and the ethic of the proposed analysis must be taken into consideration.

Nowadays, the majority of geospatial data is collected using automatic measurements, due to its scalability, as well as for being less intrusive than manual collection methods. Such measurements are made using location sensors, and the collected data can be streamed in real-time (as some applications demand, such as to live track vehicle and passenger's position in *Uber*) or stored for later access. These sensors can be under possession of the sensed entity or not, e.g., smartphones and drones, respectively.

In this section, we discuss about the process of geospatial data collection, providing the reader with the concepts and tools needed to do so. Existing data sources, their properties, and examples are shown in Section. Then, we introduce different aspects related to the quality of the collected data in Section.

2.3.1. Data Sources

Geospatial data collection is a complex activity with high costs involved. Therefore, a clear scope of the data usage is needed in order to obtain the expected results without incurring additional costs with collection and post-processing steps. One of the main steps is selecting the data source, which must take into consideration the trade-off between the quality of the data obtained and the application costs of the sensing technology. Consider, for example, a scenario where we want to map points of interest within a city region. In this case, location data collected from Call Detail Records (CDR) do no have similar accuracy such as GNSS data; however, check-in data from LBSN may produce similar results with inferior collection costs. These suppositions are also valid to previously collected data from third-party agents that may have acquisition costs.

2.3.1.1. Global Navigation Satellite Systems

Global Navigation Satellite Systems (GNSS), such as GPS[5], GLONASS, and BeiDou, are satellite-based technologies that provide precise information about objects location on the

[5]Although this term is commonly used in the literature, GPS is a specific implementation of a GNSS. Most modern sensing devices (e.g., smartphones) are capable of connecting to multiple GNSS networks.

Earth's surface (Figure 2.5(a)). To do this, we need receptors that capture the signals from the satellites, calculating their position in degrees of latitude and longitude. To obtain its position, a receptor captures the signal from three satellites and performs a process called triangulation. In the presence of a fourth satellite, a receptor can also obtain the current date and time with high precision. GNSS are robust, capable of operating uninterruptedly, independently of weather conditions, and virtually in any exterior environment on the Earth's surface.

On the one hand, GNSS sensors are the most advanced geospatial data collection devices, obtaining accurate positioning with high frequency. Adding to that, nowadays these sensors are found in the majority of mobile devices such as smartphones and smartwatches, allowing a high-scale collection with low-cost. On the other hand, issues such as users' privacy, high energy consumption, and loss of signal must be addressed. First, the accuracy and frequency of positioning sensing of a user allows obtaining personal information such as home and work location [Kang et al. 2004], as well as work times [Gu et al. 2016]. Next, the elevated energy consumption must be considered, given that mobile devices have limited sources of energy and that most of the times the sensing devices have other functions sharing the same source; thus, the frequency of update must be just enough for the purpose. Together with that, physical barriers such as buildings and mountains can interrupt the signal reception, generating spatial and temporal gaps in the sensing [Silva et al. 2015] as well as positioning errors, issues that must be addressed using post-processing techniques such as filling and calibrating data [Celes et al. 2017].

2.3.1.2. Call Detail Records and Wireless Networks

Wireless networks – such as mobile telecommunications networks and Wireless access points (AP) – used by mobile devices, computers, and even vehicles, can be employed as low-cost sources of geospatial data. Mobile networks obtain location information for a user by the closest base stations, even if the user is not actively interacting (for example, making a call) with the network at the moment. The reported position corresponds to the transmission range of the contacted tower (Figure 2.5(b)). By its turn, the location of devices connected to access points is given by the unique identifier of the AP and the timestamp of when the access started. Like in mobile networks, the positioning precision corresponds to the transmission range of the AP. For both, multiple towers or multiple access points can be used to obtain a preciser positioning through triangulation and signal reception angles [Naboulsi et al. 2016].

There are certain advantages in using wireless networks to collect geospatial data. First, the energy consumption is low, given that the collection depends only on the connection of the devices to the network, a fundamental activity for their usage. Moreover, by being less precise, this collection is also less intrusive in comparison to the one using GNSS sensors, which reduces the rejection of the sensed individuals to provide their location. Finally, we can highlight the larger number of devices capable of connecting to wireless networks in comparison to devices with GNSS sensors. On the other hand, the biggest drawback of this source is the reduced precision, which can influence the quality of the geospatial data collected. Additionally, in comparison to GNSS, collection can suffer from the lack of coverage in remote areas, where network signal cannot be found.

2.3.1.3. Check-in Data

Check-in records represent the presence of a user in a location of interest (also called point of interest) during a determined time interval. These can be public spaces, such as parks, restaurants, and shops, as well as private and individual spaces, such as home and workplaces (Figure 2.5(c)). Check-in records are collected through location-based social networks such as *Facebook*, *Twitter*, and *Foursquare*, that capture information about the user, the visited location and the time of the visit, and may or may not contain the geographical coordinates of the PoI. The LBSN can use location sensors, such as GNSS and AP's identifiers - to automatically detect (or suggest) the user location, instead of requiring manual input. Like CDRs, check-in records present coarse temporal and spatial granularity, because the visits performed by a user during their activities are only registered if they voluntarily share it through their LBSN, which does not always happen due to issues such as privacy (i.e., the user does not want others to know where he is) and safety (i.e., the user fears that reporting his location may put himself in danger).

2.3.2. Collection Quality

Besides the technologies used and the sources of geospatial data, other issues must be taken into consideration during the definition of the data collection scope. Next, we introduce the main questions that arise, discussing how they can affect, not only the data collection, but its processing and the resulting analysis as well.

2.3.2.1. Location accuracy and precision

Accuracy, in terms of geospatial data collection, refers to the proximity of the collected location measurement to the real position of an entity. Therefore, the bigger the accuracy, closer the measurement is to the real value. It can vary from a few millimeters (e.g., high-capacity GNSS sensors) to kilometers (e.g., mobile networks in remote areas), thus comprehending accuracy is fundamental to select the most suitable source to the analysis subject. Precision, on its turn, represents the variance of the collection, and the bigger its value, more centered are the samples around a single point. To obtain preciser measurements, more powerful location sensors can be applied, nonetheless resulting in higher energy consumption and a higher collection cost overall.

Although they are related, a higher accuracy does not necessarily translate into a higher precision, and vice-versa. In the presence of data with low accuracy or precision, two approaches can be applied. The first is using more powerful sensors – when possible. The second is applying techniques to calibrate the collected measurements. This can occur during the collection, such as in the usage of additional signals in GNSS and wireless networks, as well as during data processing [Newson and Krumm 2009, Hoteit et al. 2016].

2.3.2.2. Privacy

Geospatial data can be used to report the location of various entities, in special those capable of moving such as humans and vehicles. Therefore, by providing their location,

an individual allows the access to a real and valuable information. By the one hand, numerous benefits are provided from mobility data analysis and its applications. For example, services such as *Uber* and *Pokemon Go* are only possible by sharing location data. By the other hand, this information can be used by malicious agents to construct privacy attacks for the users being sensed, which can in turn demotivate them to share their data. The compromise between the utility of geospatial data and the risk to their privacy must be considered when collecting their data.

To be able to share their location information, we must guarantee users' privacy with the usage of techniques that reduce the amount of details or that make it harder to access the shared data. We can highlight two techniques: data anonymization and obfuscation. For the former, we replace the users' identifiers by randomly-generated pseudonyms, which can be done during data processing and even directly in the sensing devices [Krumm 2009]. However, even with this anonymization, attacks may re-identify users by detecting mobility patterns [Maouche et al. 2017]. The role of obfuscation is to prevent this by creating small distortions in the data in an attempt to break existing patterns, without affecting its quality [Duckham and Kulik 2005a, Duckham and Kulik 2005b].

2.3.2.3. Sampling rate

The interval between two consecutive location reports is also an important aspect to be discussed. We call this interval sampling rate, in a way that data with higher sampling rates present a smaller time interval between two samples. Therefore, during a given interval, collections with higher sampling rates will produce larger amounts of data. As expected, more data implies in more details and bigger utility, allowing, for example, the analysis of detailed mobility trajectories. On the other hand, collecting data with high sampling rates leads to higher usage of resources such as power and storage, which are limited in portable and mobile devices. When implementing a geospatial data collection process, we need to specify the collection sampling rate to guarantee the coverage of the activities or behaviors which are subject of our analysis.

While some data sources allow us to set the sampling rate (e.g., GNSS), others depend on the interaction of the users with the system (e.g., LBSNs), and thus their rates cannot be controlled directly. Even when possible, adjusting the sampling rate can be costly, as discussed above. From that, spatial and temporal gaps will occur, intervals in which there is no information regarding the whereabouts of the user. These gaps can be filled using techniques such as interpolation [Hoteit et al. 2016] and algorithms based on historical data [Silva et al. 2015, Chen et al. 2017]. The enrichment of geospatial data transforms sparse into dense in an artificial manner, with no need to change the methods and tools of collection.

2.3.2.4. Scale

Finally, we discuss the collection scale. To represent the simulated environment and produce meaningful results, the set of entities contained in the data must be significant. The

scale can refer to the number of distinct users being sensed, to the number of time intervals (hours, days or weeks), and to the dimensions of the area being monitored. The scale of these dimensions must comprise a scenario in which the resulting analysis does not present bias due to: user limitations (the sample of individuals does not represent the population); time limitations (the period does not contains all the expected situations); and space limitations (the dimensions of the covered area do not represent the real environment).

When the collected geospatial data are not enough to produce results, some approaches are used to increase the data volume. Data fusion [Rettore et al. 2020] is a technique that combines two or more datasets producing as output a single set containing all the data. However, for very distinct datasets, its application can be compute-intensive. Another approach is the generation of synthetic data, which uses statistical [Kosta et al. 2012] and machine learning methods to generate data that is similar to the real behavior. Although it demands a precise modeling of real data, this technique has as benefit the capacity of generating synthetic data on demand and in large-scale.

2.4. Data Storage

This section will discuss concepts, techniques and existing tools for storing geospatial data, which is fundamental for dealing with massive amounts of data. It is important to discuss this topic, given that in its majority, traditional relational databases are not fit for efficiently storing geospatial data. This is due to the different forms of representing geospatial data that may not be compatible with tabular storage. Moreover, we must consider data manipulation, i.e., inserting and querying data, and the need to perform geospatial filters.

2.4.1. Spatial components structure

The real world is too complex to be fully represented by a data structure, and thus we must select the relevant characteristics (e.g., roads, buildings) for each scenario. To digitally represent geospatial data, there are two primary structures: vector and raster. Vector structure is based on using points, lines, and polygons to define the location and limits of an object. By its turn, raster structure uses a regular cell grid to define objects. Each structure has its own advantages and drawbacks in data modeling. Moreover, we can combine the two approaches in a single project aiming to get the best of both. Table 2.3 presents a comparison between the two data structures.

The characteristics of each structure impact directly on the details they are capable of capturing. Figure 2.6 shows the transformations of a real world representation into vector and raster structures. The vector one represents the existing entities considering their dimensions and shapes. To do so, we must select which geometric shapes will be used in the representation. On the other hand, the raster structure reduces every feature contained in a single cell into a basic identification according to a codification criteria. In the example shown in Figure 2.6, we used the dominant area criterion, in which the label corresponds to the feature that occupies the majority of the cell. Another possible criterion is using the feature located at the center of the cell.

Taking into consideration the characteristics mentioned above, it is clear that there is not a superior structure. Raster structure has the advantage of being simpler to store,

Table 2.3. Comparison between vector and raster structures

Characteristic	Vector	Raster
Data structure	Generally complex	Generally simple
Storage requirements	Small, for most data	Big, for most uncompressed data
Coordinate systems conversion	Simple	Can be slow due to the volume, may require resampling
Positional precision	Limited	Depends on the resolution adopted
Accessibility	Often complex	Easy to modify by using specific programs
Visualization and output	Similar to maps, with continuous curves; poor for images	Good for images, but can produce jagged effects
Spatial relations between objects	Topological relationships between available objects	Spatial relationships must be inferred
Modeling and analysis	Map algebra is limited	Easier superposition and modeling

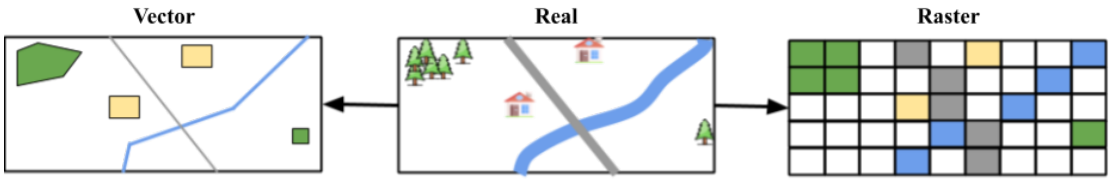


Figure 2.6. Vector and raster structures (Adapted from [Lisboa Filho and Iochpe 2001])

specially when dealing with digital images, such as aerial pictures and satellite images. On its turn, vector structure tends to be more accurate, providing better visualizations and efficient calculations of topological operations. At last, vector structure stores only the essential elements, reducing the amount of storage needed, while raster codifies the entire grid, which can be unnecessary.

2.4.2. Data Compression

As it can be noted, geospatial datasets are used to represent large amounts of information, demanding considerable storage capacity. Just as in traditional datasets, compression algorithms can be applied in geospatial datasets, resulting in a more efficient storage. Compression algorithms can be classified into lossy and lossless: while the former obtain high compression levels at the cost of reducing the data quality, the latter preserves its quality but obtains lower compression levels. Although for certain applications lossy compression is acceptable, when dealing with geospatial data it can severely impact the analysis' results. Thus, using lossy compression algorithms when processing or analyzing geospatial data is not recommended.

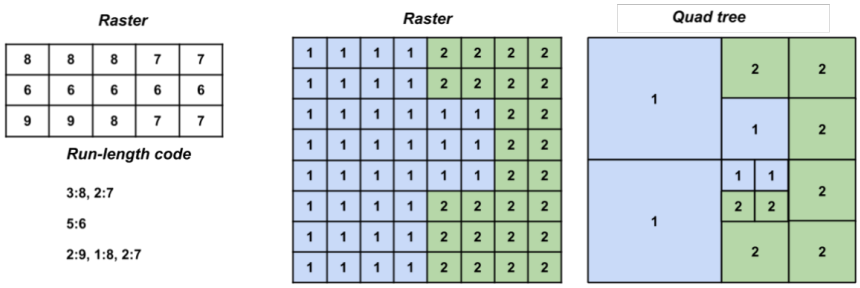


Figure 2.7. Run-length and Quad Tree compression of raster data

Given their representation characteristics, most compression algorithms for geospatial data are focused on raster structured data. A common method for compressing raster data is Run-length code. This compression technique is based on codifying a sequence of cells to optimize space when there are large sequences of cells with the same value. This coding is represented by two numbers, with the first indicating the amount of cells with the same identification and the second being the identification itself. Another well-known coding is a bi-dimensional version of Run-length code called Quad Tree [Finkel and Bentley 1974]. In this method, areas with same values are represented with a single identifier. To do this, the grid is divided recursively into increasing square blocks until the division is not possible anymore, resulting in squares where all identifiers inside them are equal. An application example of both methods can be seen on Figure 2.7.

2.4.3. Databases

One of the most important components for geospatial data analysis tools, such as Geographic Information Systems (GIS), are Spatial Database Management Systems (Spatial DBMS). Besides having the conventional functionalities found in traditional database management systems, Spatial DBMS accept different geospatial reference systems, providing functions for querying and manipulating this type of data. Additionally, they are capable of indexing geospatial data both using coordinates as well as using polygons, improving efficiency. However, without indexing, both querying locations and filtering regions are inefficiently, mainly when dealing with large volumes of data.

Some examples of commonly used DBMS for storing geospatial data are *MySQL*, *PostgreeSQL* with *PostGis* extension, and *Oracle Spatial*. They use the same spatial data standard, called Simple Feature Specification - Structured Query Language (SFS-SQL), which describes a common storage and access model for geometries (points, lines, and polygons). SFS-SQL is a standard defined by Open Geospatial Consortium (OGC) [ogc 2023] that, besides describing the geometries used by GIS, present the definition of operations between geometries.

2.4.4. Indexing

Finally, indexes are data structures used to increase the performance of queries in database systems, allowing data to be retrieved in more efficient ways than linear searches. Indexing (i.e., the creation of indexes) in geospatial databases is useful not only to efficiently query data, but also to perform many spatial operations. We can cite the identification

of k-nearest neighbors, geocoding (obtaining coordinates from an address), and reverse geocoding (obtaining an address from coordinates).

The most applied structure for indexing geospatial data is called R-Tree [Guttman 1984], that indexes geometries by using a balanced tree structure. This strategy has as advantage the capacity of querying data with logarithmic time complexity ($O(log_{|M|}|N|)$), justifying its usage in diverse databases and geospatial analysis tools, such as *PostGis*, *Oracle* and *GeoPandas*. On the other hand, we must take into consideration the processing costs for generating the tree, which may limit its usage.

Besides R-Tree, we can use grid-based systems, which are easier to implement, such as *Geohash* [Morton 1966] and *H3* [Uber 2015]. Grid-based systems can analyze massive amounts of geospatial data through the division of larger areas into uniquely identifiable cells. *H3* provides an hierarchical spatial index based on hexagons, grouping points in hexagons of different sizes according to the precision needed in the analysis. *H3* index levels determine the area of hexagons and its selection is essential for a better precision during indexing. On the one hand, hexagons too big will group distant points in a same cell. On the other hand, hexagons too small will result in a very large number of indexes, decreasing performance. *Geohash*, on its turn, uses rectangles instead of hexagons. However, the indexing approach is the same as H3, with the disadvantage that each rectangle has eight neighbors while each hexagon has six – more neighbors means more locations to search.

2.5. Data Transformation and Knowledge Extraction

This section presents the core of the process of geospatial data analysis, which comprehends the steps of data transformation and knowledge extraction.

2.5.1. Data Transformation

The transformation of geospatial data involves five steps: formatting, sampling, cleaning, filtering, and aggregating. Although they are all essential in preparing data to analyze, the need to apply each one is defined by the initial conditions of the input data and the characteristics of the analysis. Moreover, multiple transformation iterations can occur with the aim to refine and validate the obtained results.

2.5.1.1. Formatting

When working with geospatial data, it is essential to observe formatting, since it can be represented in different forms. When dealing with geographical coordinates (latitude and longitude), for example, there are three basic formats in which they can be found: degrees, seconds, and minutes; degrees, seconds, and decimal seconds; and degrees and decimal degrees. The choice of which format to use will depend of the application, given that a larger number of decimal places allows representing locations with higher precision. Tools such as Geospatial Databases, GIS software, and sources of official data often use the last format, that represents coordinates as degrees and decimal degrees (e.g., 38.8897°). Another form of representation is the set of coordinates, creating polygons or lines. Platforms such as OpenStreetMaps represent polygons as a sequence of coordinates; The *Shapely* library, on the other hand, adopts the Well-Known Text (WKT)

format to represent sequences of coordinates. Other approaches are Shapefiles, sheets, and JSON. Besides certifying that all data are with the same format, it is important to verify if all records are under the same projection and datum, since different projections will lead to wrongful analysis.

Other data sources, such as check-in and wireless networks, may require data formatting as well. For the former, different LBSNs may refer to a same location with different names (by abbreviating words, for example), requiring the identification of all the possible formats and the conversion to a unique identifier. Additionally, during the collection time span, locations can change names or addresses, which also must be considered during formatting. Similarly, for the latter, access points and towers can change their identifiers over time, as well as each device can reproduce this identifier in a particular way, according to different technologies and software specifications.

2.5.1.2. Cleaning

Besides undesired locations, the usage of data collected from urban areas can bring some challenges, such as the imprecision of geographical coordinates. This occurs due to the large number of obstacles, such as buildings, that obstruct the line of sight of satellites, a phenomenon called urban canyons [Johnson and Watson 1984], reducing their capacity to attribute a location with the required accuracy. For this reason, it is important to analyze the impact of the affected locations in the collected data, and if needed, remove them. The main consequence of using erroneous data (or data with low location accuracy) is reducing the significance of geospatial analysis, specially when dealing with distances between points and users' density.

Even if they do not refer to geospatial data, other data dimensions must be considered during cleaning. These, besides not being affected by the issues presented above, can also contain irregularities such as null values or outliers. Therefore, cleaning them is crucial to guarantee trustful results.

2.5.1.3. Filtering

While the cleaning step aims to remove wrongfully sensed information, the filtering step aims to select, given a cleaned dataset, a subset that meets the specified rules for analysis. While here we specifically refer to rules applied to geospatial data, it is valid to point out that filtering can include other data dimensions as well. Thus, according to the application, we can remove data located at unwanted regions, e.g., outside the city limits where we want to focus our analysis.

It is important to highlight that records from geospatial data are frequently represented as a single point, that is, a single latitude and longitude pair. To filter records within a location of interest – which can be a street, a place, or even a city – we can consider the geometric representation of this area (i.e., a polygon or a set of polygons encompassing the whole area). To do this, various geographic operations can be applied to evaluate the relationship between geometries. Such operations can be found in GIS software and databases with support for geospatial data, as well as in libraries such as *Shapely*

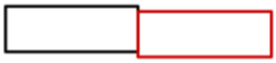
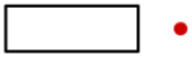
Operation	Description	False	True
CONTAINS	Verifies if one geometry contains the other completely		
CROSSES	Verifies if one geometry overlaps the other at any point, but not all (does not contains it)		
DISJOINT	Verifies if the geometries are disjoint, i.e., they do not share any point in common		
EQUALS	Verifies if the two geometries are the same		
INTERSECTS	Verifies if the geometries intersect at any point		
OVERLAPS	Verifies if two geometries of equal dimension overlap, but are not contained		
RELATE	Verifies if two geometries relate through intersections on interior or exterior limits		
TOUCHES	Verifies if the geometries intersect at their limits, but their interiors do not intersect		
WITHIN	Verifies if one geometry is contained inside the other. It is the inverse relation of the operation CONTAINS.		

Figure 2.8. Examples of operations that verify a relation between two geometries

Operation	Description	Base geometry	Resulting geometry
BUFFER	Creates a buffer geometry around the input geometry at a distance specified by the user		
CONVEX HULL	Returns the convex hull of the specified geometry		
DIFFERENCE	Return a geometry containing all the points in the base geometry but not in the comparing geometry		
INTERSECTION	Return a geometry containing the points observed in both base and comparing geometries		
SYMDIFFERENCE	Returns a geometry containing all the points that not intersect (inverse of INTERSECTION)		
UNION	Returns a geometry obtained from the union of all the input geometries		

Figure 2.9. Examples of spatial operations on geometries

and *GeoPandas*. Figure 2.8 details the main operations, with each returning a Boolean value (true or false) by comparing two geometries. Additionally, there are operations that do not analyze the relationship between two geometries, but perform spatial operations, returning values or new geometries as output, as it can be seen in Figure 2.9.

2.5.1.4. Sampling

When analyzing large amounts of data, we may not possess the computational capacity or even the interest in the whole dataset. Frequently, a small sample is enough to understand and to generate knowledge about all the data. Once data is formatted, cleaned and filtered, we can consider using samples of data. Sometimes, data sampling is the first step in the data transformation process, making it easier to clean and filter the dataset, since we are dealing with a reduced volume. For this, we must guarantee that the sample is representative of the population, otherwise it will introduce bias, invalidating the results. In a valid sample, records are selected randomly to guarantee that no biased record gets picked on purpose. In the resulting sample, the distribution of the data inside it must originate from the same distribution of the population.

Regarding geospatial data, we must also observe if the sample represents the existing variations regarding space and time. For the former, entities in different regions may behave differently, and thus it is important to capture all this variability. For the latter, the original dataset can cover a time interval with comprising many weekdays, holidays, and even multiple seasons. Considering the change in the users' routine caused by these periods, we must consider splitting the sampling process or creating a stratified sampling, generating one or more samples to meet the needs of our analysis.

2.5.1.5. Aggregation

Finally, with the data ready to be used, we must analyze if its current granularity is enough for the desired analysis. If not, then we must aggregate our data, according to defined aggregating rules and aggregating regions. For example, if our analysis is focused on populational metrics by neighborhood, then aggregating records allows us to proceed while also reducing the processing needed by decreasing the amount of data. Aggregation can also happen due to privacy considerations: in certain scenarios, analyzing the raw data can reveal personal information. By aggregating, individual traces can disappear or become indistinguishable from one another. Finally, aggregation can also occur to enable fusing with other data sources. Weather forecast, safety, and traffic indicators are examples of data sources frequently used in geospatial data analysis, each one with different granularity. In order to combine these sources in an efficient way, aggregation can be used with few or no information loss.

2.5.2. Knowledge Extraction

Next, we showcase applications of knowledge extraction from geospatial data. For each one, we give examples of its functioning and highlight its importance in data analysis.

2.5.2.1. Radius of gyration

When working with geospatial data, we frequently assume that the analyzed users have a reference location, which can be their home, work place or any other point of interest. Based on that, we can calculate a users' radius of gyration, that can be defined as the maximum distance between its reference location and the other locations visited by them.

The radius of gyration provides information about the mobility of users, allowing their classification and evaluation according to their radius' value, for example.

To compute a users' radius of gyration, we must first define the criteria to identify their reference location through geospatial data. Some approaches found in the literature are using a random point [Kosta et al. 2012], the average of all points, the most visited location, and the first point reported in a day [Ekman et al. 2008]. We must consider the characteristics of the dataset, such as its granularity and sensors used, to select a criteria that identifies reasonable reference locations. Additionally, we must select a distance function (haversine or euclidean, for example) to perform the measurements.

Knowing the radius of gyration of users has contributed to research about urban mobility. Previous studies have used this information to argue that users tend to explore ever increasing regions over time [González et al. 2008], as well as shown that some users tend to explore more than others [Pappalardo et al. 2015]. Additionally, radius of gyration analysis has been used to investigate locations visited by social networks users [Jurdak et al. 2015]. Finally, it has aided in understanding escape routes during natural disasters [Wang and Taylor 2014] and understanding how communication in social networks helps disseminating actions in global scale [Morales et al. 2017].

2.5.2.2. Spatial Clustering

Other common activity when dealing with geospatial data is the need of identifying groups that present similar behavior patterns. Such patterns can be represented by users that frequent a same region [Sakai et al. 2014], providing information to identify regions with higher demands for services, locations where diseases can be spread easier, among others. To identify these regions or groupings, we use a technique called spatial clustering.

There are several clustering algorithms found in the literature, using different approaches to obtain groupings, such as algorithms based on distance (K-means), based on density (DBScan), and based on distribution (GMM). Although they are algorithms frequently used in common datasets, using them for clustering geospatial data can lead to errors. Algorithms based on distance frequently work with euclidean distance, that fails to compute the correct distance between points on the Earth's surface [Ingole and Nichat 2013]. While algorithms based on distribution can also work, the most popular algorithms for spatial clustering are the ones based on density, since they can be applied for most types of data. However, parameters must be well-calibrated so the groupings produced are representative.

There are many different applications of spatial clustering: adjusting network resources to accommodate demand of a region, feeding recommender systems for indicating similar locations, and personalizing marketing campaigns according to the visited location [Tran et al. 2013] are some examples.

2.5.2.3. Social relationships

Inferring social relationships between individuals in a dataset can help us better understand daily activities, such as mobility patterns that change due to others [Cho et al. 2011], or even the projection of disease propagation in a pandemic [Firestone et al. 2011]. For this to happen, correctly identifying social links between two users is essential. Social links derived from an analysis can indicate if two individuals are friends, acquaintances, neighbors, workmates or housemates, for example. However, inferring theses links is not a simple task: we must define rules – using the data – to affirm that two individuals were in contact with each other. To define a contact, one may use the spatial proximity between users' reported positions (in case of GNSS data) or the presence at a same location (in case of LBSN data). In both scenarios, the temporal proximity must also be considered, to guarantee that users were in the same area at the same time. Moreover, commonly used rules to identify social links are high encounter frequency, the intersection between users' social links sets, and the detection of communities.

Regarding the relation of social links and geospatial data, there are works in the literature to identify the probability of users having similar tastes by the occurrence of similar routes [Hung et al. 2009]. Other works also analyze the occurrence of communities from social links obtained from geospatial data [de Melo et al. 2015], as well as using these links to disseminate data in opportunistic networks [Domingues et al. 2022].

2.6. Visualization

Visualizing data is essential in all steps during analysis. We start visualizing data right after collecting it, with the objective to validate and verify, identifying issues that can be corrected through new collections or processing steps. Next, different visualizations are created to depict the distribution of the data, allowing the detection of outliers and the discovery of the population characteristics. During analysis, visualizations aid in decision making and presenting partial results. Finally, the end results are also presented through visualizations, that ease the understanding and illustrate the proposed ideas.

In this way, it is fundamental to create easy to read graphics with no considerable time or effort. These two factors guarantee that the usage of graphics become a tool for the process, and not another problem to be solved. For this to happen, we must know the different forms of visualization of geospatial data and the results they provide. Different from numerical and categorical data, in which graphics such as barplots and lineplots are enough to transmit the information, the visualization of geospatial data generally involves the need of a map over which the location records are drawn. Additionally, we need to consider aggregation strategies due to the large volume of data. Drawing massive amounts of data can be inefficient in terms of computing resources, and also produce polluted results. Finally, we are frequently more interested in visualizing existing patterns than the behavior of unique individuals.

To aid in understanding geospatial data graphics, we draw maps from the represented regions to approximate the figure to the real environment. To draw a map, first we define its type and projection. Map types define the characteristics it represents, such as terrain, territorial borders, streets and roads. Projection, on its turn, refers to representing

data in two or three dimensions. Both factors must be selected in a way to increase the understanding of the generated graphic, while also non essential characteristics must be unconsidered to avoid pollution. For example, using a three-dimension projection for two dimensional data will only add uncertainty to a figure.

Zoom levels refer to magnify the region covered by the figure, approximating (zooming in) or distancing (zooming out). A graphic must cover the whole area where the geospatial data are located, unless the objective is to highlight a specific region. However, in the occurrence of outliers, we may generate visualizations that are distant from the main region of interest, and thus, zooming in may be considered.

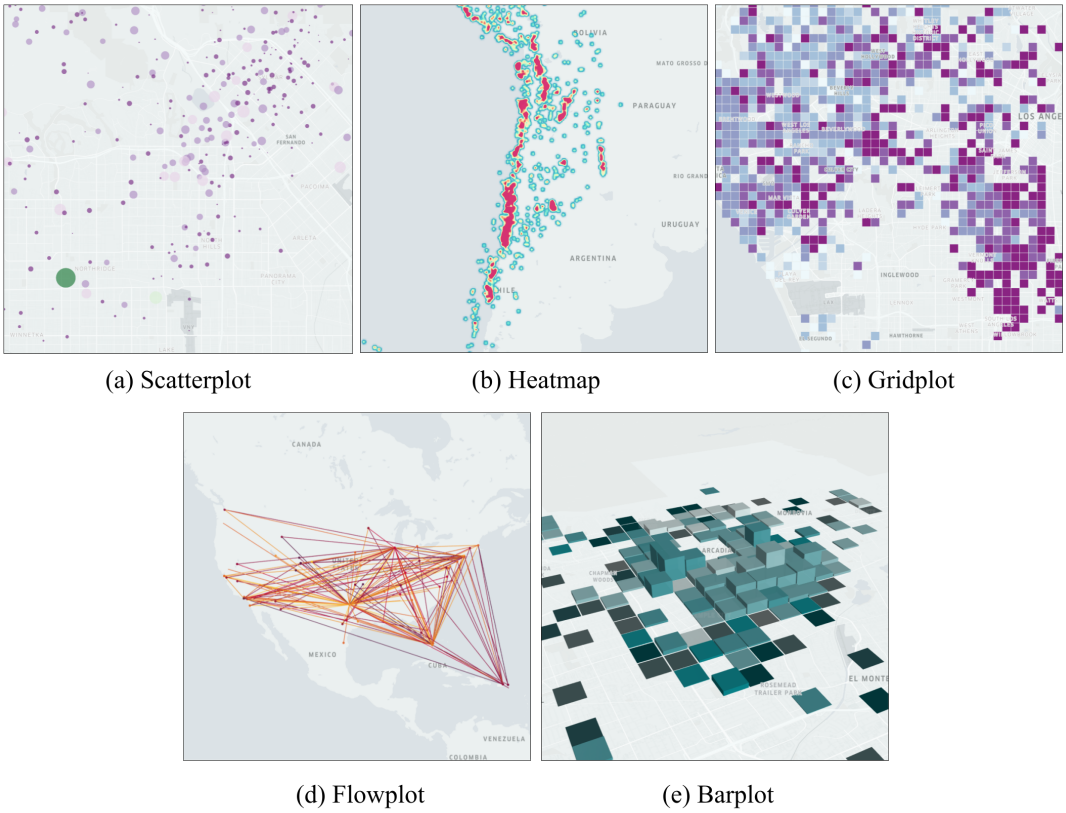
Finally, beyond aesthetic reasons, color scales in a graphic can be used to indicate intensity levels (such as altitude) and to separate categories found in data (such as device type or PoI category). Whenever possible, we must opt for color scales with high contrast (considering that the figure may also be visualized in grayscale) and that refer to the categories being represented (e.g., using a red and blue scale to indicate temperature). In the presence of an elevated number of categories, colorscales can produce confusing visualizations, and in this case, they can be replaced by textual (e.g., written values) and visual (e.g., different symbols for each category) subtitles.

2.6.1. Visualization Types

The visualization types presented next are the basic structure to construct graphics for analyzing geospatial data. From these approaches, it is possible to develop new visualizations, adapting and adding characteristics to meet the requirements of the desired result. By creating the first sketches, the reader will notice that modifications are needed to make the proposed graphic as clear as possible, which is essential for its understanding. Adjusting the map format, zoom levels, color scale, and visual and textual subtitles are some examples of modifications. Figure 2.10 shows an example for each of the visualizations presented next.

Scatterplot. Scatterplots constitute the simplest form of visualizing geospatial data. In it, locations are drawn as points over the map according to their coordinates. Points can have different sizes, colors and formats to depict different characteristics found in data. Scatterplots are easy to comprehend, to implement and modify. On the other hand, visualizing large amounts of data in scatterplots will lead to point superposition, causing loss of details. Additionally, in this scenario their construction will be compute-intensive. With this in mind, they should be used to visualize the dispersion of the collected data, but we should avoid them when we want to explore the details in the population.

Heatmap. Heatmaps are used to represent the density of a variable by means of intensity curves and color scales. When combined with geospatial data, both the variable's density and its spatial dispersion (i.e., how density changes according to space) can be shown in a single visual. In this combination, high-density regions can be formed, that can represent a topology or PoI, for example. Heatmaps are recommended when data does not present a uniform location distribution, leading to the existence of regions with higher densities. Because they are base in density regions, heatmaps can produce poor results for datasets with sparse locations, that is, when points are too far away from each other, with no aggregation.



(a) Scatterplot (b) Heatmap (c) Gridplot

(d) Flowplot (e) Barplot

Figure 2.10. Examples of geospatial data visualizations

Gridplot. Gridplots are created by dividing the analyzed region into a grid with defined dimensions, where the minimum location unit are the cells that compose the grid. For each cell, the data located in the coordinates within it are aggregated through an aggregation function (e.g., summation, average, minimum or maximum value) and the result of this function represents the cell. Besides the aggregation function, another parameter that must be defined is the cell size, that is, the area covered by it. Smaller cells are capable of capturing more details, while larger cells can lose relevant information. On the other hand, it is easier to have an empty cell (with no data inside it) as its size reduces. Eventually, a compromise between the level of details and the minimum number of records inside each cell may be needed.

Flow plot. Flow plots are used to represent the movement flow of entities (that can be people, vehicles, drones, or others) between two or more regions. This displacement is represented by arcs that connect the origin and destination regions, together with an intensity indicator, which can be done using the arc's dimensions (e.g., a bolder arc indicates a bigger flow), or through a color scale. Arcs may not be symmetric, that is, the flow's intensity from point A to point B can be different of the one from point B to point A. In this way, multiple graphics can be drawn to cover all cases and avoid superposition that leads to confusion. Other forms of representing flows between regions can be found in the literature, such as transition graphs and transition matrices. However, they do not communicate the spatial distribution as well as flow plots.

Barplot. Barplots for geospatial data project bars over a spatial region, where each bar represent the impact of a variable over that specific location. Like gridplots, they are useful when geospatial data have information sensitive to location. By its turn, these graphics allow readers to observe regions of interest easily, due to the projection of the bars in a third dimension. However, we must consider the limits of representing data in third dimension through non-interactive means, such as in printing: the projection may cover some regions, causing loss of information. Therefore, barplots for geospatial data should be preferred for digital and interactive means.

2.6.2. Visualization Tools

Next, we discuss some libraries and tools commonly used to create geospatial data visualizations. While most plotting tools (e.g., gnuplot and matplotlib) can be used to draw geospatial data, our aim here is to focus on those that provide specific resources for this kind of data, thus reducing the amount of work needed and speeding up the process.

Bokeh. Bokeh is a library to build and visualize graphics. Built with Python, it is capable of generating interactive visualizations, allowing users to change parameters and scales and add new data to an existing graphic in real time. This makes Bokeh an interesting alternative for publishing results in websites. Additionally, it is capable of rendering large amounts of data. Lastly, users can find an extensive collection of visualizations available for use, including the ones discussed above, as well as more complex ones.

Kepler. Kepler is a geospatial data analysis tool developed by *Uber*. It can be used through a Web interface that allows loading data, performing aggregations, filtering, and projecting over a detailed map of the Earth's surface using many different visualizations, such as scatterplots, heatmaps, flowplots and barplots. Besides that, the user can select the map type and projection, draw additional geometries to complement the visualization, observe data over time (when a time variable is available) and export the results to different formats. However, Kepler can only be used to visualize geospatial data with locations based on latitude and longitude coordinates.

OSMNx. OSMNx is a Python library for building geospatial data visualizations focused on road maps. Using data from OpenStreetMaps, it is capable of generating personalized visualizations of road mesh of a determined region, over which the user can draw additional geospatial data. Besides that, it is capable of constructing the road network by using graphs, allowing the mapping of geospatial locations to their closest roads, that in turn allows computing distances, road-based shortest paths, as well as metrics and complex networks algorithms.

QGis. QGis is a multiplatform software to visualize and edit geospatial data for analysis. It is a robust system, capable of loading large amounts of data and that supports different input types. Due to its advanced capacities, it can produce high-quality visualization. On the other hand, it demands a stepper learning curve. Finally, by being a standalone tool, its interaction with other geospatial data analysis (e.g., Python scripts for processing) can be complex, making it less preferable for building quick visualizations.

2.7. Conclusion

This chapter presented an in-depth study about geospatial data and its applications in knowledge extraction, generating new products and services and allowing the capture

of new sources of revenue, as well as advancing the state-of-the-art in areas related to mobility, Internet of Things (IoT), urban computing, among others. For this, we presented theoretical concepts and the main techniques and tools applied in the steps of collection, storage, transformation, knowledge extraction, and visualization of geospatial data.

We highlighted the importance of mobility to the development of new techniques and technologies applied to issues such as traffic flow prediction and control, contagion models, network resource optimization, among others. In this scenario, the interest for studying and applying geospatial data has become bigger, due to its capacity to represent the entities' mobility behavior, specially in the case of humans and vehicles. However, although there is an increasing demand for research involving this type of data, still there is not a consensus regarding the adequate methodologies for analyzing it, due to the lack of references that introduce in a clear way the concepts and techniques to be applied.

Aiming to fill this gap, Section 2.2 introduced the main concepts related to geospatial data, such as geographic characteristics, reference systems, and spatial projections. Section 2.3 discussed the process of data collection, highlighting the most commonly used existing sources (GNSS devices, networks and LBSNs). Additionally, characteristics from the collected data such as accuracy and precision, granularity, and entities' privacy were discussed. Next, Section 2.4 presented the particularities of storing geospatial data, discussing spatial database management systems, compression methods for geospatial data, and spatial indexing structures, required for an efficient storage. Section 2.5 presented the steps of data transformation and knowledge extraction. The former is composed by the sub tasks of formatting, cleaning, filtering, sampling, and data aggregation. For the latter, we introduced some examples of applications, such as radius of gyration and spatial clustering. Lastly, data visualization techniques and the tools used to create them were discussed in Section 2.6.

References

(2023). The Home of Location Technology Innovation and Collaboration | OGC. [Online; accessed 7. Aug. 2023].

Barbosa, H., Barthelemy, M., Ghoshal, G., James, C. R., Lenormand, M., Louail, T., Menezes, R., Ramasco, J. J., Simini, F., e Tomasini, M. (2018). Human mobility: Models and applications. *Physics Reports*, 734:1–74.

Bolstad, P. (2016). *GIS Fundamentals: A First Text on Geographic Information Systems*. Eider Press, 5 edition.

Castro, P. S., Zhang, D., e Li, S. (2012). Urban traffic modelling and prediction using large scale taxi gps traces. In *International Conference on Pervasive Computing*, pages 57–72. Springer.

Cebrian, M. (2021). The past, present and future of digital contact tracing. *Nature Electronics*, 4(1):2–4.

Celes, C., Silva, F. A., Boukerche, A., d. C. Andrade, R. M., e Loureiro, A. A. F. (2017). Improving vanet simulation with calibrated vehicular mobility traces. *IEEE Transactions on Mobile Computing*, 16(12):3376–3389.

Chen, G., Viana, A. C., e Sarraute, C. (2017). Towards an adaptive completion of sparse call detail records for mobility analysis. In *2017 IEEE international conference on pervasive computing and communications workshops (PerCom workshops)*, pages 302–305. IEEE.

Cho, E., Myers, S. A., e Leskovec, J. (2011). Friendship and mobility: user movement in location-based social networks. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1082–1090.

de Mattos, E. P., Domingues, A. C., e Loureiro, A. A. (2019). Give me two points and i'll tell you who you are. In *2019 IEEE Intelligent Vehicles Symposium (IV)*, pages 1081–1087. IEEE.

de Melo, P. O. V., Viana, A. C., Fiore, M., Jaffrès-Runser, K., Le Mouël, F., Loureiro, A. A., Addepalli, L., e Guangshuo, C. (2015). Recast: Telling apart social and random relationships in dynamic networks. *Performance Evaluation*, 87:19–36.

Domingues, A. C., de Souza Santana, H., Silva, F. A., de Melo, P. O. V., e Loureiro, A. A. (2022). Socialroute: A low-cost opportunistic routing strategy based on social contacts. *Ad Hoc Networks*, 135:102949.

Duckham, M. e Kulik, L. (2005a). A formal model of obfuscation and negotiation for location privacy. In *International conference on pervasive computing*, pages 152–170. Springer.

Duckham, M. e Kulik, L. (2005b). Simulation of obfuscation and negotiation for location privacy. In *International conference on spatial information theory*, pages 31–48. Springer.

Ekman, F., Keränen, A., Karvo, J., e Ott, J. (2008). Working day movement model. In *Proceedings of the 1st ACM SIGMOBILE workshop on Mobility models*, pages 33–40.

Finkel, R. A. e Bentley, J. L. (1974). Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4(1):1–9.

Firestone, S. M., Ward, M. P., Christley, R. M., e Dhand, N. K. (2011). The importance of location in contact networks: Describing early epidemic spread using spatial social network analysis. *Preventive Veterinary Medicine*, 102(3):185 – 195. Special Issue: GEOVET 2010.

González, M. C., Hidalgo, C. A., e Barabási, A.-L. (2008). Understanding individual human mobility patterns. *Nature*, 453(7196):779–782.

Gu, Y., Yao, Y., Liu, W., e Song, J. (2016). We know where you are: Home location identification in location-based social networks. In *2016 25th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9. IEEE.

Guttman, A. (1984). R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57.

Hess, A., Hummel, K. A., Gansterer, W. N., e Haring, G. (2015). Data-driven human mobility modeling: a survey and engineering guidance for mobile networking. *ACM Computing Surveys (CSUR)*, 48(3):1–39.

Hoteit, S., Chen, G., Viana, A., e Fiore, M. (2016). Filling the gaps: On the completion of sparse call detail records for mobility analysis. In *Proceedings of the Eleventh ACM Workshop on Challenged Networks*, pages 45–50. ACM.

Hung, C.-C., Chang, C.-W., e Peng, W.-C. (2009). Mining trajectory profiles for discovering user communities. In *Proceedings of the 2009 International Workshop on Location Based Social Networks*, pages 1–8.

Ingole, P. e Nichat, M. M. K. (2013). Landmark based shortest path detection by using dijkestra algorithm and haversine formula. *International Journal of Engineering Research and Applications (IJERA)*, 3(3):162–165.

Johnson, G. T. e Watson, I. D. (1984). The determination of view-factors in urban canyons. *Journal of Climate and Applied Meteorology*, 23(2):329–335.

Jurdak, R., Zhao, K., Liu, J., AbouJaoude, M., Cameron, M., e Newth, D. (2015). Understanding human mobility from twitter. *PloS one*, 10(7):e0131469–e0131469.

Kang, J. H., Welbourne, W., Stewart, B., e Borriello, G. (2004). Extracting places from traces of locations. In *Proceedings of the 2nd ACM international workshop on Wireless mobile applications and services on WLAN hotspots*, pages 110–118. ACM.

Kosta, S., Mei, A., e Stefa, J. (2012). Large-scale synthetic social mobile networks with swim. *IEEE Transactions on Mobile Computing*, 13(1):116–129.

Krumm, J. (2009). A survey of computational location privacy. *Personal and Ubiquitous Computing*, 13(6):391–399.

Lisboa Filho, J. e Iochpe, C. (2001). Modelagem de bancos de dados geográficos. In *Apostila do XX Congresso Brasileiro de Cartografia, Porto Alegre*.

Maouche, M., Mokhtar, S. B., e Bouchenak, S. (2017). Ap-attack: a novel user re-identification attack on mobility datasets. In *Proceedings of the 14th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, pages 48–57. ACM.

Marques-Neto, H. T., Xavier, F. H., Xavier, W. Z., Malab, C. H. S., Ziviani, A., Silveira, L. M., e Almeida, J. M. (2018). Understanding human mobility and workload dynamics due to different large-scale events using mobile phone data. *Journal of Network and Systems Management*, 26(4):1079–1100.

Morales, A. J., Vavilala, V., Benito, R. M., e Bar-Yam, Y. (2017). Global patterns of synchronization in human communications. *Journal of the Royal Society Interface*, 14(128):20161048.

Morton, G. M. (1966). A computer oriented geodetic data base and a new technique in file sequencing.

Motlagh, N. H., Taleb, T., e Arouk, O. (2016). Low-altitude unmanned aerial vehicles-based internet of things services: Comprehensive survey and future perspectives. *IEEE Internet of Things Journal*, 3(6):899–922.

Naboulsi, D., Fiore, M., Ribot, S., e Stanica, R. (2016). Large-scale mobile traffic analysis: a survey. *IEEE Communications Surveys & Tutorials*, 18(1):124–161.

Newson, P. e Krumm, J. (2009). Hidden markov map matching through noise and sparseness. In *Proceedings of the 17th ACM SIGSPATIAL international conference on advances in geographic information systems*, pages 336–343.

Pappalardo, L., Simini, F., Rinzivillo, S., Pedreschi, D., Giannotti, F., e Barabási, A.-L. (2015). Returners and explorers dichotomy in human mobility. *Nature communications*, 6(1):8166.

Rettore, P. H., Santos, B. P., Lopes, R. R. F., Maia, G., Villas, L. A., e Loureiro, A. A. (2020). Road data enrichment framework based on heterogeneous data fusion for its. *IEEE Transactions on Intelligent Transportation Systems*, 21(4):1751–1766.

Sakai, T., Tamura, K., e Kitakami, H. (2014). Extracting attractive local-area topics in georeferenced documents using a new density-based spatial clustering algorithm. *IAENG International Journal of Computer Science*, 41(3):185–192.

Silva, F. A., Celes, C., Boukerche, A., Ruiz, L. B., e Loureiro, A. A. (2015). Filling the gaps of vehicular mobility traces. In *Proceedings of the 18th ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, pages 47–54.

Tran, K. A., Barbeau, S. J., e Labrador, M. A. (2013). Automatic identification of points of interest in global navigation satellite system data: A spatial temporal approach. In *Proceedings of the 4th ACM SIGSPATIAL international workshop on geostreaming*, pages 33–42.

Uber (2015). *H3: A hexagonal hierarchical geospatial indexing system*.

Wang, Q. e Taylor, J. E. (2014). Quantifying human mobility perturbation and resilience in hurricane sandy. *PLoS one*, 9(11).

Zheng, Y., Capra, L., Wolfson, O., e Yang, H. (2014). Urban computing: concepts, methodologies, and applications. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 5(3):1–55.

Sobre os autores

Augusto C.S.A. Domingues é bacharel (2016) em Ciência da Computação pela Universidade Federal de Viçosa e mestre (2018) em Ciência da Computação pela Universidade Federal de Minas Gerais. Atualmente, é doutorando em Ciência da Computação pela UFMG, onde realiza pesquisas na grande área de Redes de Computadores, com ênfase em Computação Ubíqua, Computação Urbana, Redes Móveis e Mobilidade Humana. Durante o mestrado, trabalhou com pesquisas relacionadas a caracterização da escolha de rotas em *traces* de mobilidade veicular, propondo um algoritmo para gerar trajetórias de grão fino, enriquecendo dados existentes na literatura e permitindo novas oportunidades de pesquisa na área. Também trabalhou com a caracterização de *traces* de mobilidade em geral, propondo uma ferramenta que permite a extração de um conjunto de métricas sociais, espaciais e temporais. Como resultado, publicou trabalhos em diversas conferências nacionais e internacionais e em periódicos. No doutorado, além dos tópicos já descritos acima, também realiza pesquisas relacionadas a mecanismos de proteção de privacidade de localização de usuários em redes.

Fabrício A. Silva é doutor (2015), mestre (2006) e bacharel (2004) em Ciência da Computação pela UFMG. Durante a graduação e mestrado, trabalhou em projetos de pesquisa na área de redes de sensores sem fio. Entre 2006 e 2010, trabalhou em uma Startup desenvolvendo projetos inovadores na área de processamento de linguagem natural e aprendizagem de máquina. Desde 2010 é professor na UFV-Campus Florestal. Em 2015, obteve o grau de doutor, tendo desenvolvido sua tese na área de redes veiculares. Durante o doutorado, atuou como pesquisador visitante na Universidade de Ottawa, Canadá, sob a orientação do professor Azzedine Boukerche. Sua tese foi escolhida como a segunda melhor do Brasil no Concurso de Teses e Dissertações da Sociedade Brasileira de Computação. Atualmente, é Bolsista de Produtividade em Pesquisa 2 e tem trabalhado com pesquisa nas áreas de sistemas distribuídos e computação ubíqua, principalmente no estudo e caracterização de grandes volumes de dados desses sistemas.

Antonio A. F. Loureiro possui graduação em Ciência da Computação pela Universidade Federal de Minas Gerais (1983), mestrado em Ciência da Computação pela Universidade Federal de Minas Gerais (1987) e doutorado em Ciência da Computação pela University of British Columbia, Canadá (1995). Atualmente é Professor Titular do Departamento de Ciência da Computação da Universidade Federal de Minas Gerais. Tem experiência na área de Ciência da Computação, com ênfase em sistemas distribuídos, atuando principalmente nos seguintes temas: algoritmos distribuídos, computação móvel/ubíqua, comunicação sem fio, gerenciamento de redes, redes de computadores, redes de sensores sem fio. Atualmente, é Bolsista de Produtividade em Pesquisa 1A.