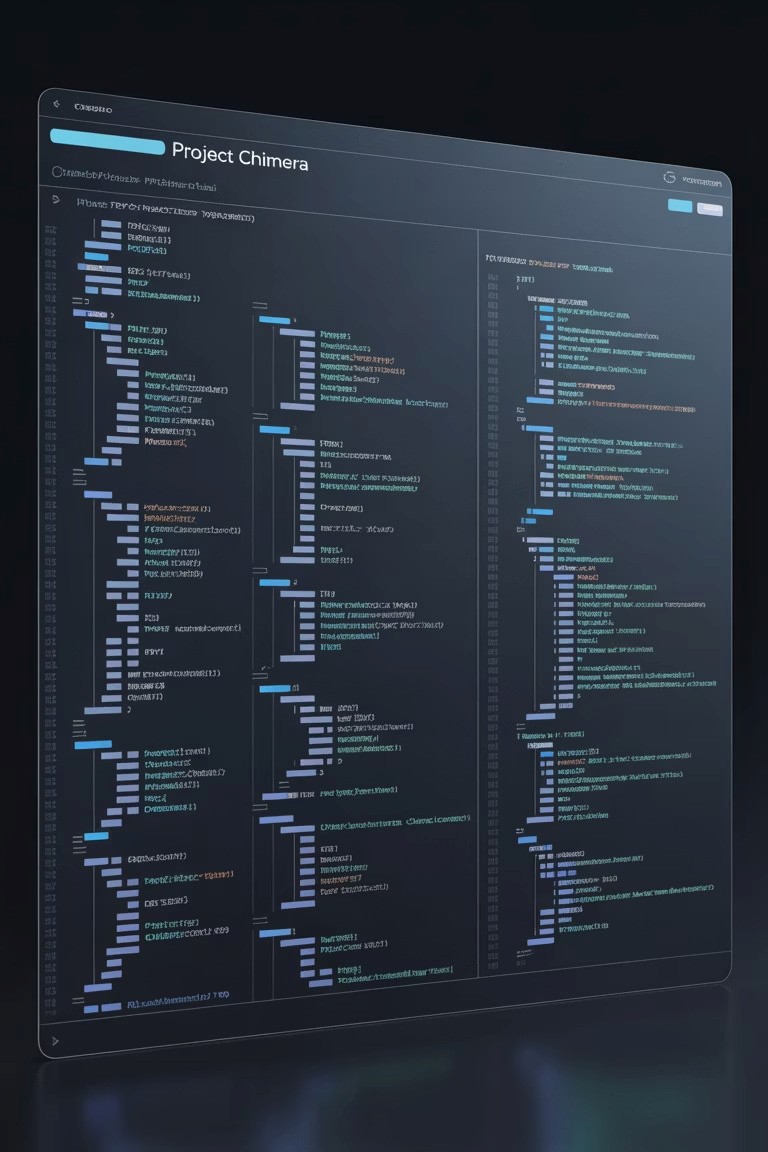


Arquiteturas,

Processamento e ML em

Python



Arquitetura de Repositório

Conceito e

Motivação

Estrutura organizada para

garantir manutenibilidade e

escalabilidade do código.

Separação de

Responsabilidades

Cada componente com

função específica e bem

definida.

Organização em Python

Estruturas de diretórios e pacotes seguindo boas práticas.

Estrutura Básica de um Projeto Python

README e Arquivos de Configuração

Documentação e configurações do projeto

Diretórios Principais

src/, tests/, docs/

3

Estrutura de Módulos

Organização lógica dos componentes

Exemplo: Estruturando um Repositório Python

Arquivos de Configuração

README.md para documentação

setup.py para instalação

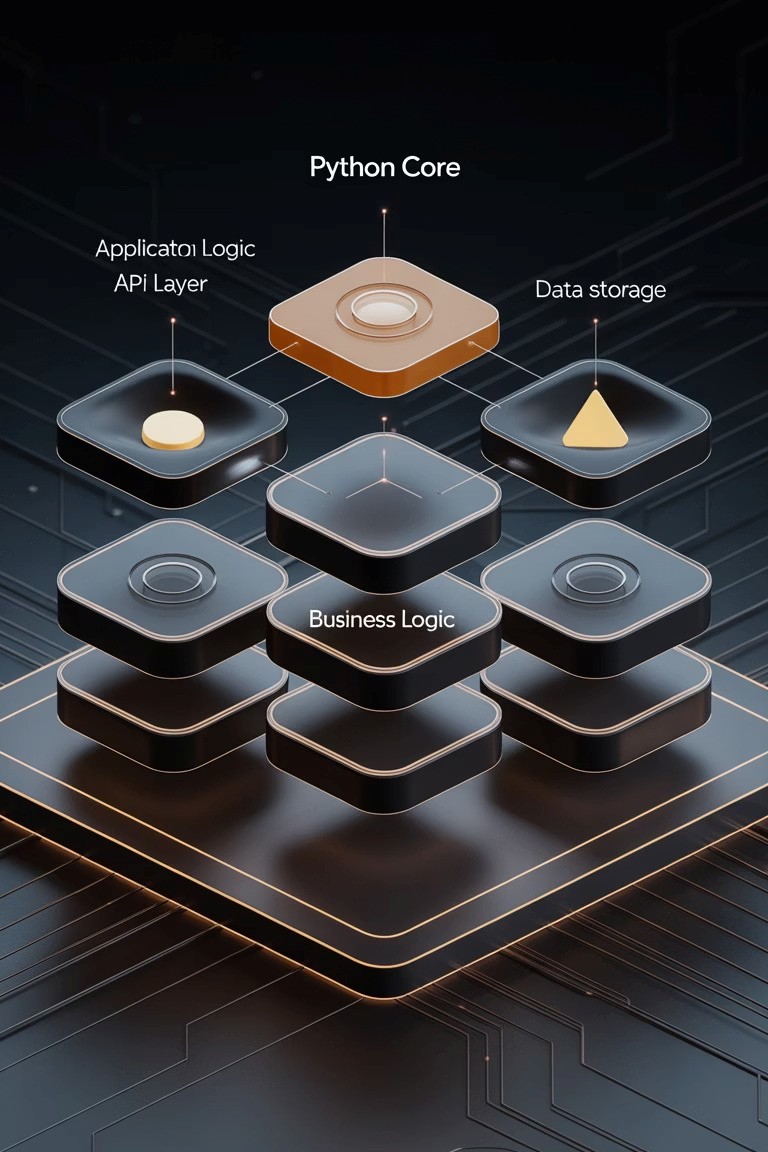
requirements.txt para dependências

Estrutura de Diretórios

src/ para código fonte

tests/ para testes automatizados

docs/ para documentação completa



Principais Padrões de

Estrutura

Modularização

Separar código em módulos

coesos

Organizar helpers, core e

interfaces

Evitar arquivos muito grandes

Isolamento de

Dependências

Usar ambientes virtuais

Separar dependências de

desenvolvimento

Versionar requirements.txt

Convenções de Código

Seguir PEP 8

Usar nomes descritivos

Documentar funções e classes

Clean Architecture em Python

4

Camada de Domínio

Entidades e regras de negócio

Camada de Aplicação

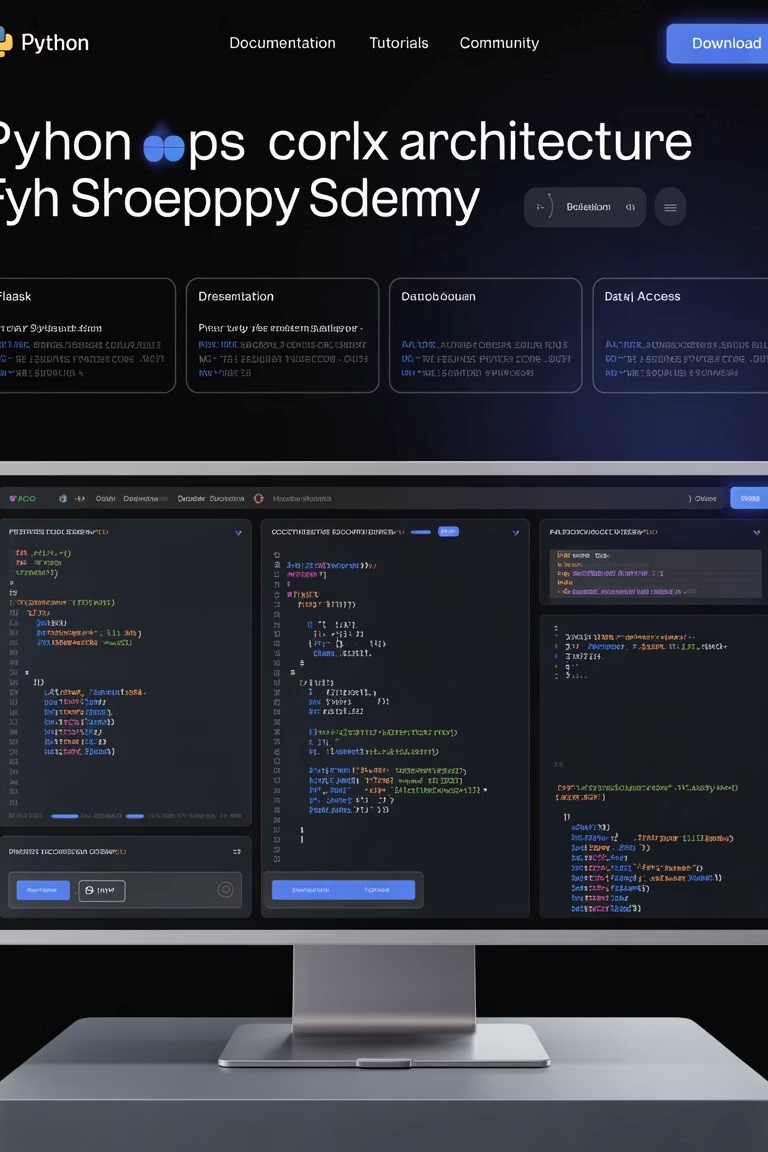
Casos de uso e serviços

Camada de Adaptadores

Controllers e gateways

Camada de Infraestrutura

Frameworks e drivers



Exemplo prático de Clean

Architecture

Interface do Usuário

Templates Flask e formulários

Controllers

Rotas Flask que recebem requisições

Casos de Uso

Lógica de negócio independente

4

Persistência

SQLAlchemy e PostgreSQL

Código: Interface e Implementação

import abc

class RepositorioABC(abc.ABC):

@abc.abstractmethod

def salvar(self, entidade):

pass

@abc.abstractmethod

def buscar\_por\_id(self, id):

pass

Interface Abstrata

class ProdutoRepositorio(RepositorioABC):

def \_\_init\_\_(self, session):

self.session = session

def salvar(self, produto):

self.session.add(produto)

self.session.commit()

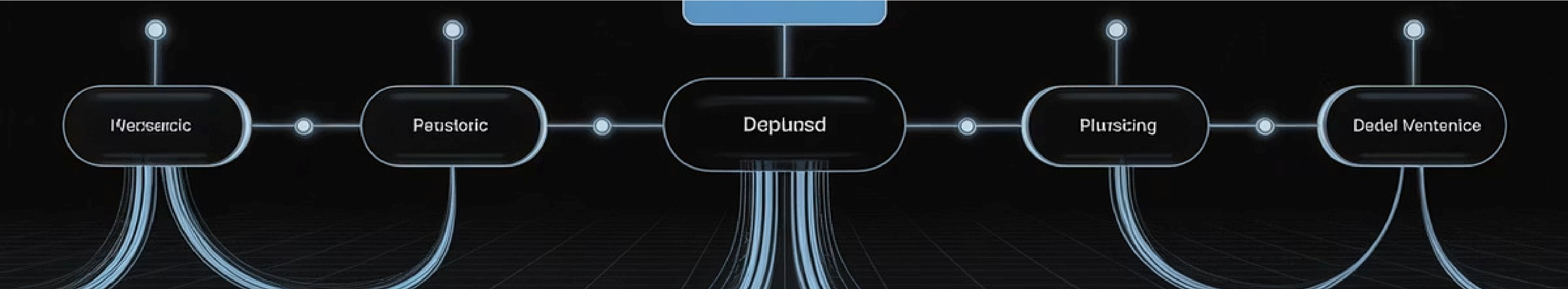
return produto

def buscar\_por\_id(self, id):

return self.session.query(

Produto).filter\_by(id=id).first()

Implementação Concreta



Aplicação em Machine Learning: MLOps

Estruturação do Repositório

Organização modular

Separação de código, dados e modelos

Documentação clara

Versionamento de Modelos

MLflow para tracking de experimentos

DVC para versionamento de dados

Git para código

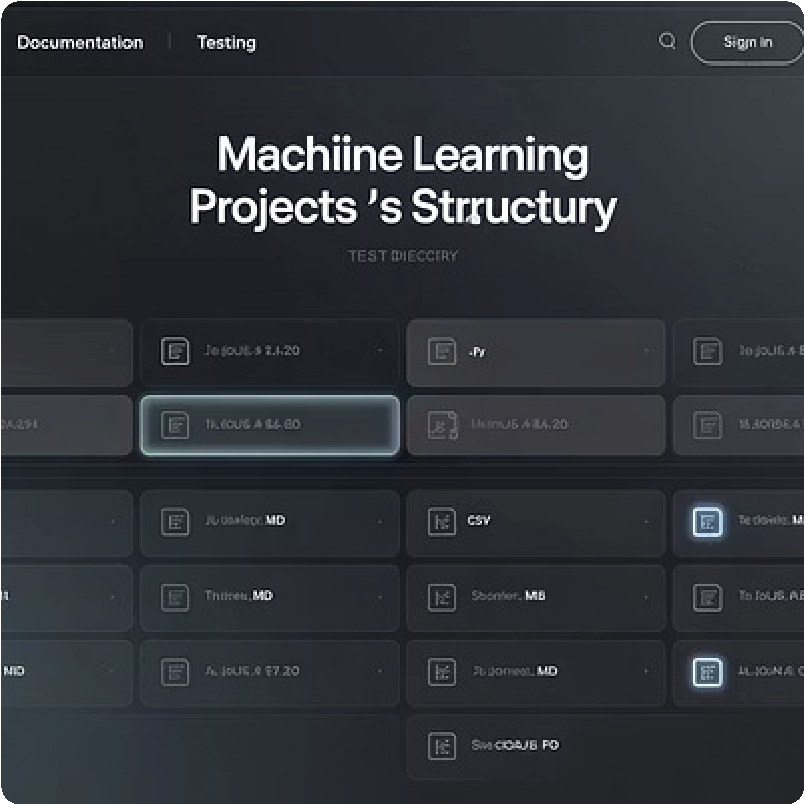
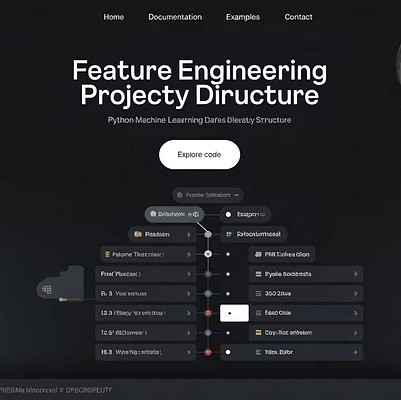
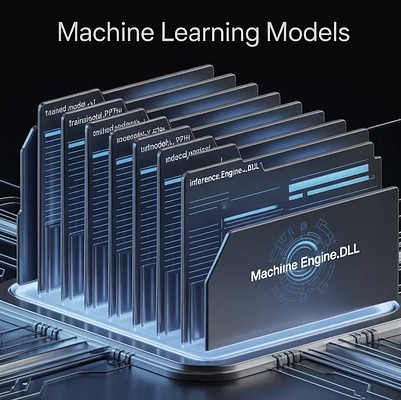
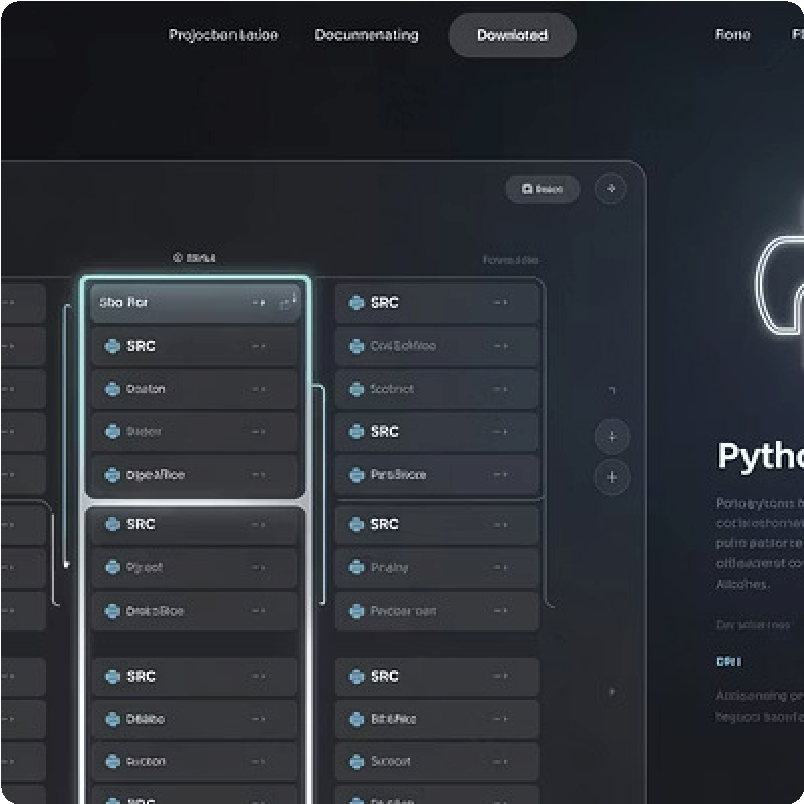
Automatização de Pipeline

CI/CD para treino e deploy

Testes automatizados

Monitoramento em produção

Exemplo: Estrutura para Projeto ML



Automação de Testes

com Pytest

Estrutura de Testes

Organizar testes em diretórios espelhando a estrutura do código.

Tipos de Testes

Implementar testes unitários, integração e end-to-end.

Execução Automatizada

Integrar com CI/CD para validação contínua.

Cobertura de Código

Monitorar métricas de cobertura para garantir qualidade.

Organização de Dados no Repositório

Dados Brutos

Armazenados em data/raw/ sem modificações

Processamento

Pipeline transforma dados em data/processed/

Resultados

Modelos e análises em output/

Exercício Prático: Criando um Esqueleto de Projeto

projeto\_ml/

=

%%

README.md

=

%%

requirements.txt

=

%%

setup.py

=

%%

src/

'

%%

=

\_\_init\_\_.py

'

=

%%

data/

'

%%

=

features/

'

=

%%

models/

'

5

%%

visualization/

%%

=

tests/

=

%%

data/

'

%%

=

raw/

'

%%

5

processed/

5

%%

notebooks/

Estrutura a Criar

# Criar estrutura

mkdir -p projeto\_ml/src/{data,features,models,visualization}

mkdir -p projeto\_ml/tests

mkdir -p projeto\_ml/data/{raw,processed}

mkdir -p projeto\_ml/notebooks

# Inicializar arquivos

touch projeto\_ml/README.md

touch projeto\_ml/requirements.txt

touch projeto\_ml/setup.py

touch projeto\_ml/src/\_\_init\_\_.py

Comandos para Inicialização



Arquitetura Cliente-Servidor

Cliente (Front-end)

Interface com usuário que solicita serviços

Servidor (Back-end)

Processa solicitações e fornece recursos

Comunicação

Protocolos HTTP, WebSocket, etc.

Exemplo: Servidor Flask em Python

from flask import Flask, request, jsonify

app = Flask(\_\_name\_\_)

@app.route('/api/dados', methods=['GET'])

def obter\_dados():

return jsonify({

'mensagem': 'Dados recuperados',

'status': 'sucesso',

'dados': [1, 2, 3, 4, 5]

})

@app.route('/api/dados', methods=['POST'])

def inserir\_dados():

dados = request.json

# processamento aqui...

return jsonify({

'mensagem': 'Dados recebidos',

'status': 'sucesso'

})

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

Código do Servidor

Características

API RESTful simples

Suporte a métodos GET e POST

Respostas em formato JSON

Facilmente extensível

Exemplo: Cliente Consumindo API

import requests

# Consumindo endpoint GET

resposta = requests.get('http://localhost:5000/api/dados')

if resposta.status\_code == 200:

dados = resposta.json()

print(f"Mensagem: {dados['mensagem']}")

print(f"Dados: {dados['dados']}")

else:

print(f"Erro: {resposta.status\_code}")

# Enviando dados via POST

novos\_dados = {'nome': 'Produto', 'valor': 29.99}

resposta = requests.post(

'http://localhost:5000/api/dados',

json=novos\_dados

)

print(f"Resposta: {resposta.json()}")

Código do Cliente

Características

Biblioteca requests para HTTP

Tratamento de códigos de status

Envio e recebimento de JSON

Estrutura simples e reutilizável

Modelo de Comunicação: Stateless e Stateful

# Exemplo REST (Stateless)

@app.route('/api/produto/')

def obter\_produto(id):

# Busca produto pelo ID

return jsonify(produto)

Stateless

Cada requisição é independente e contém todas as informações

necessárias.

# Exemplo com sessão (Stateful)

@app.route('/login')

def login():

# Autentica usuário

session['usuario\_id'] = 123

return redirect('/dashboard')

@app.route('/dashboard')

def dashboard():

if 'usuario\_id' not in session:

return redirect('/login')

# Mostra dashboard

Stateful

O servidor mantém informações sobre o cliente entre requisições.



Deploy de APIs para Machine

Learning

Treinar Modelo

Desenvolver e treinar modelo ML em Python

Serializar

Salvar modelo treinado com pickle/joblib

Criar API

Desenvolver endpoint /predict com Flask/FastAPI

Implantar

Deploy em servidor ou contêiner Docker

Demonstração: Deploy de modelo ML como API

import joblib

from flask import Flask, request, jsonify

app = Flask(\_\_name\_\_)

# Carregar modelo pré-treinado

modelo = joblib.load('modelo\_ml.joblib')

# Rota para previsões

@app.route('/predict', methods=['POST'])

def predict():

# Obter dados da requisição

dados = request.json

# Fazer previsão

resultado = modelo.predict([dados['features']])

# Retornar resultado

return jsonify({

'previsao': resultado[0].tolist()

})

if \_\_name\_\_ == '\_\_main\_\_':

app.run(host='0.0.0.0', port=5000)

Carregamento do Modelo

import requests

import json

# Dados para previsão

dados = {

'features': [5.1, 3.5, 1.4, 0.2]

}

# Enviar requisição

resposta = requests.post(

'http://localhost:5000/predict',

json=dados

)

# Imprimir resultado

print(json.dumps(resposta.json(), indent=2))

Testando a API

Arquiteturas Scaláveis: Load Balancer e

Microserviços

4

Clientes

Acessam aplicação via HTTP/API

Load Balancer

Distribui carga entre servidores

Microserviços

Serviços independentes e

especializados

Banco de Dados

Armazenamento persistente

distribuído

Exemplo: API Gateway Python

from flask import Flask, request

import requests

app = Flask(\_\_name\_\_)

SERVIÇOS = {

'usuarios': 'http://serviço-usuarios:5000',

'produtos': 'http://serviço-produtos:5001',

'pedidos': 'http://serviço-pedidos:5002'

}

@app.route('//', methods=['GET', 'POST'])

def gateway(serviço, caminho):

if serviço not in SERVIÇOS:

return {'erro': 'Serviço não encontrado'}, 404

# Obter URL do serviço

url\_serviço = f"{SERVIÇOS[serviço]}/{caminho}"

# Encaminhar requisição

if request.method == 'GET':

resp = requests.get(url\_serviço, params=request.args)

elif request.method == 'POST':

resp = requests.post(url\_serviço, json=request.json)

return resp.json(), resp.status\_code

if \_\_name\_\_ == '\_\_main\_\_':

app.run(host='0.0.0.0', port=8000)

Benefícios

Implementação de Gateway

Ponto único de acesso

Roteamento de requisições

Abstração da complexidade

Possibilidade de implementar:

-

Autenticação centralizada

Rate limiting

-

Logging

-

-

Transformação de dados



Monitoramento de Serviços

em Produção

Logging Centralizado

Utilizar biblioteca logging

Integrar com ELK Stack

Estruturar logs em JSON

Níveis: DEBUG, INFO, WARN,

ERROR

Métricas de Aplicação

Prometheus para coleta

Grafana para visualização

Métricas de latência e

throughput

Alertas para anomalias

Health Checks

Endpoints /health e /ready

Monitoramento de dependências

Verificação de conectividade

Auto-recuperação



Exercício: Rodar Cliente e

Servidor Localmente

Criar Servidor Flask

Implementar endpoint /dados

Salvar como servidor.py

Executar com python servidor.py

Implementar Cliente

Criar script para consumir API

Implementar GET e POST

Salvar como cliente.py

Testar Interação

Executar cliente em terminal separado

Verificar logs do servidor

Analisar respostas recebidas

Seção 3: Arquitetura de Aplicações

Monolítica

Aplicação única e integrada

Multicamadas

Separação em camadas

funcionais

Microserviços

Componentes independentes

Serverless

Funções como serviço

Padrão MVC em Python

1

Model

Manipula dados e regras de negócio

View

Apresenta dados ao usuário

Controller

Processa requisições e coordena fluxo

Camadas de Aplicação

Camada de Apresentação

Interface e interação com usuário

2

Camada de Negócio

Regras e processamento de dados

3

Camada de Persistência

Armazenamento e recuperação de dados

Organização em Pacotes e Módulos

Pacote Principal

Ponto de entrada da aplicação

Subpacotes Funcionais

Agrupamento por funcionalidade

3

Módulos

Arquivos Python com funções e classes

Exemplo Prático: App Flask Modularizado

minha\_app/

=

%%

\_\_init\_\_.py # Inicialização

=

%%

config.py # Configurações

=

%%

models/ # Modelos de dados

'

5

%%

\_\_init\_\_.py

%%

=

controllers/ # Lógica de controle

'

5

%%

\_\_init\_\_.py

%%

=

views/ # Blueprints e rotas

'

=

%%

\_\_init\_\_.py

'

%%

=

auth.py # Blueprint de autenticação

'

%%

5

main.py # Blueprint principal

%%

=

templates/ # Templates HTML

5

%%

static/ # Arquivos estáticos

Estrutura de Diretórios

# minha\_app/\_\_init\_\_.py

from flask import Flask

from .config import Config

def create\_app(config\_class=Config):

app = Flask(\_\_name\_\_)

app.config.from\_object(config\_class)

# Registrar blueprints

from .views.main import main\_bp

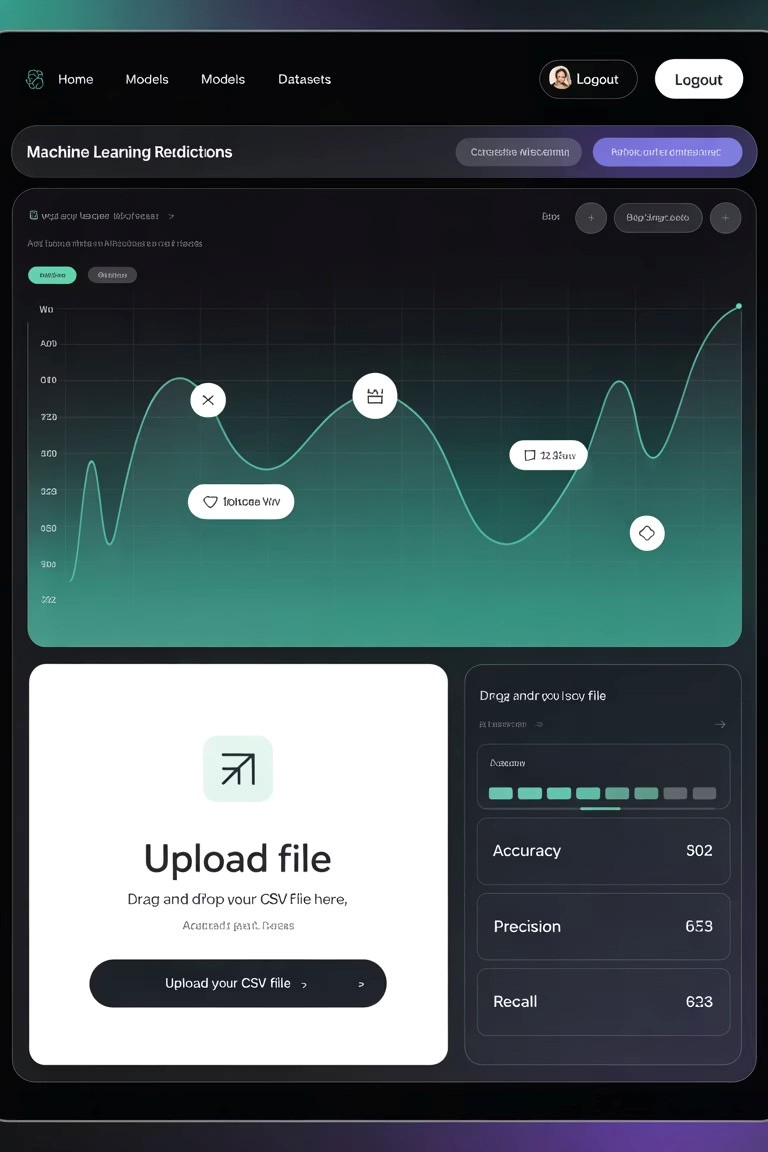
from .views.auth import auth\_bp

app.register\_blueprint(main\_bp)

app.register\_blueprint(auth\_bp, url\_prefix='/auth')

return app

Inicialização com Blueprints



Aplicações para Machine

Learning

Upload de Dados

Interface para receber arquivos ou valores

Pré-processamento

Transformação e preparação dos dados

Inferência

Aplicação do modelo ML aos dados

Visualização

Apresentação de resultados e insights

Demonstração: Front-end Simples com Streamlit

import streamlit as st

import pandas as pd

import numpy as np

import joblib

import matplotlib.pyplot as plt

# Carregar modelo

modelo = joblib.load('modelo\_regressao.joblib')

# Título da aplicação

st.title('Previsão de Preços de Imóveis')

# Input widgets

area = st.slider('Área (m²)', 30, 300, 100)

quartos = st.slider('Número de Quartos', 1, 5, 2)

banheiros = st.slider('Número de Banheiros', 1, 4, 1)

idade = st.slider('Idade do Imóvel (anos)', 0, 50, 10)

# Fazer predição

if st.button('Calcular Preço Estimado'):

features = np.array([[area, quartos, banheiros, idade]])

preco = modelo.predict(features)[0]

st.success(f'Preço estimado: R$ {preco:.2f}')

# Gráfico de importância das features

fig, ax = plt.subplots()

importancias = modelo.feature\_importances\_

nomes = ['Área', 'Quartos', 'Banheiros', 'Idade']

ax.barh(nomes, importancias)

st.pyplot(fig)

Vantagens do Streamlit

Código Streamlit

Desenvolvimento rápido

Widgets interativos

Integração com visualizações

Deploy simplificado

Ideal para protótipos

Focado em ciência de dados

Integração Backend e Frontend

@app.route('/api/predict', methods=['POST'])

def predict():

data = request.json

features = [

data['area'],

data['quartos'],

data['banheiros'],

data['idade']

]

prediction = model.predict([features])[0]

return jsonify({

'prediction': prediction,

'unit': 'BRL'

})

Backend Flask (API)

async function getPrediction() {

const area = document.getElementById('area').value;

const quartos = document.getElementById('quartos').value;

const banheiros = document.getElementById('banheiros').value;

const idade = document.getElementById('idade').value;

const response = await fetch('/api/predict', {

method: 'POST',

headers: {

'Content-Type': 'application/json'

},

body: JSON.stringify({

area, quartos, banheiros, idade

})

});

const result = await response.json();

document.getElementById('result').textContent =

`Preço estimado: R$ ${result.prediction.toFixed(2)}`;

}

Frontend (JavaScript)



Exercício: Criar Módulo Web

Simples que Faz Predição ML

Treinar Modelo Simples

Usar scikit-learn

Dataset de exemplo (Iris, Boston)

Salvar com joblib

Criar API com Flask

Endpoint /predict

Método POST para receber dados

Resposta em JSON

Implementar Interface

Formulário HTML simples

JavaScript para requisição AJAX

Exibição do resultado



Sistemas de Processamento

de Transações

Atomicidade

Transação é tudo ou nada

Consistência

Dados mantêm integridade

Isolamento

Transações não interferem entre si

Durabilidade

Alterações são permanentes

Transações em Python com SQLAlchemy

from sqlalchemy import create\_engine

from sqlalchemy.orm import sessionmaker

from models import Cliente, Conta

# Criar conexão

engine = create\_engine('postgresql://user:pass@localhost/banco')

Session = sessionmaker(bind=engine)

# Iniciar transação

session = Session()

try:

# Obter contas

conta\_origem = session.query(Conta).filter\_by(id=1).one()

conta\_destino = session.query(Conta).filter\_by(id=2).one()

# Transferir valor

valor = 100.00

conta\_origem.saldo -= valor

conta\_destino.saldo += valor

# Registrar transação

transacao = Transacao(

origem\_id=conta\_origem.id,

destino\_id=conta\_destino.id,

valor=valor

)

session.add(transacao)

# Confirmar transação

session.commit()

print("Transferência concluída com sucesso!")

except Exception as e:

# Desfazer em caso de erro

session.rollback()

print(f"Erro na transferência: {str(e)}")

finally:

# Fechar sessão

session.close()

Características

Exemplo de Transação

Controle explícito de transação

Tratamento de exceções

Rollback automático em caso de erro

Commit apenas se tudo funcionar

Garante as propriedades ACID

Exemplo: Inserção Atômica de Dados

class GerenciadorTransacao:

def \_\_init\_\_(self, session):

self.session = session

def \_\_enter\_\_(self):

return self

def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):

if exc\_type is not None:

self.session.rollback()

return False

else:

self.session.commit()

return True

# Uso com context manager

with GerenciadorTransacao(session) as tx:

cliente = Cliente(nome='João Silva', cpf='123.456.789-00')

session.add(cliente)

endereco = Endereco(

cliente\_id=cliente.id,

rua='Av. Principal',

cidade='São Paulo'

)

session.add(endereco)

# Se algo falhar, tudo é revertido automaticamente

# Se tudo funcionar, o commit é feito no final

Benefícios

Classe de Gerenciamento

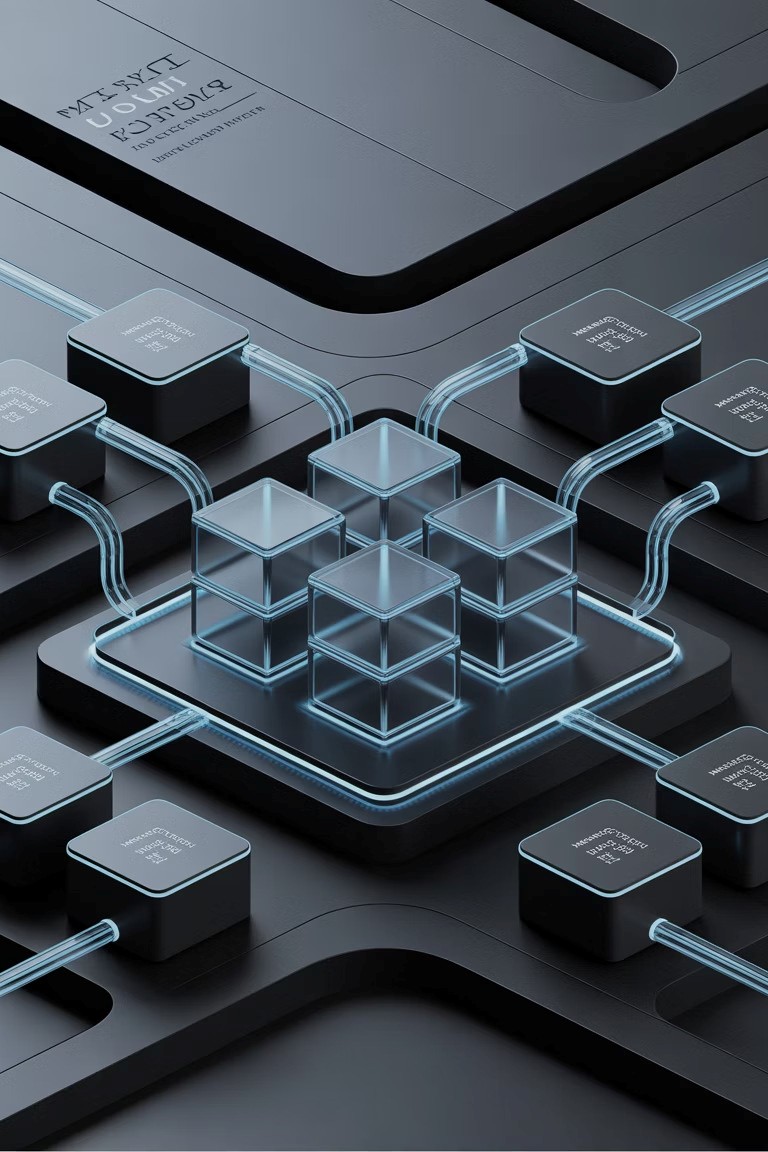
Sintaxe limpa com context manager

Isolamento de responsabilidades

Tratamento automático de erros

Fácil integração no código

Garantia de atomicidade



Filas e Mensageria para

Processamento Assíncrono

Produtor

Envia mensagens para a fila

Fila de Mensagens

Armazena mensagens (RabbitMQ, Redis)

Consumidor

Processa mensagens da fila

4

Armazenamento

Persiste resultados do processamento

Exemplo: Python Celery para Processamento de Lotes

# tasks.py

from celery import Celery

# Criar aplicação Celery

app = Celery(

'tarefas',

broker='redis://localhost:6379/0',

backend='redis://localhost:6379/1'

)

# Definir tarefa

@app.task

def processar\_dados(dataset\_id):

# Simular processamento pesado

print(f"Processando dataset {dataset\_id}")

# Carregar dados

dados = carregar\_dataset(dataset\_id)

# Processar

resultado = aplicar\_transformacoes(dados)

# Salvar resultado

salvar\_resultado(dataset\_id, resultado)

return {

'status': 'concluído',

'dataset\_id': dataset\_id,

'registros\_processados': len(dados)

}

Configuração do Celery

# main.py

from tasks import processar\_dados

# Iniciar processamento assíncrono

resultado = processar\_dados.delay(42)

# Verificar status (não bloqueante)

print(f"Tarefa iniciada: {resultado.id}")

print(f"Pronta? {resultado.ready()}")

# Obter resultado (bloqueante)

resultado\_final = resultado.get(timeout=10)

print(f"Resultado: {resultado\_final}")

# Processamento em lote

jobs = []

for i in range(1, 100):

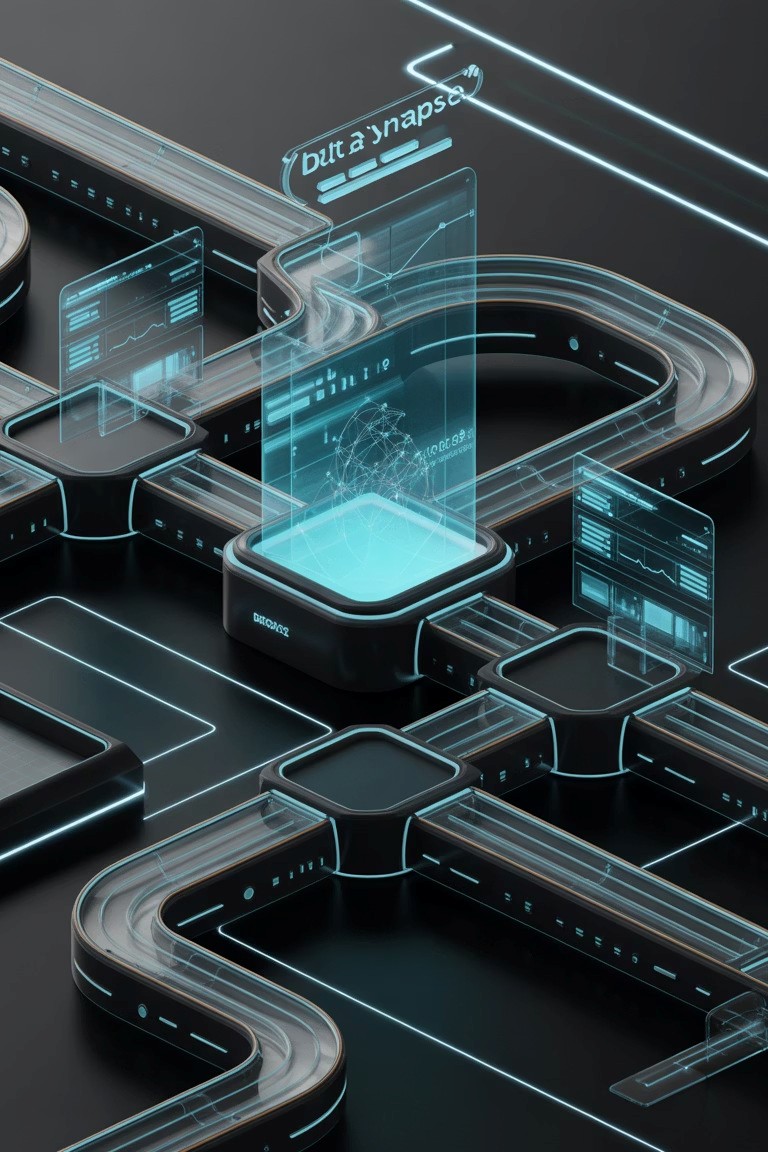
job = processar\_dados.delay(i)

jobs.append(job)

# Coletar todos os resultados

resultados = [job.get() for job in jobs]

Executando Tarefas



Aplicação em ML: Pipeline de

Treinamento Lote

Ingestão de Dados

Coleta e validação dos dados brutos

Pré-processamento

Limpeza e transformação de features

3

Treinamento do Modelo

Ajuste de parâmetros e validação cruzada

Persistência

Armazenamento do modelo treinado e métricas



Logs e Auditoria de

Transações

Registro de Transações

Timestamp preciso

Identificação do usuário

Detalhes da operação

Estado anterior e posterior

Armazenamento Seguro

Logs imutáveis

Criptografia de dados sensíveis

Redundância e backup

Segregação de acesso

Consulta e Análise

Ferramentas de busca

Alertas para anomalias

Relatórios periódicos

Conformidade regulatória

Exercício: Simular Lançamentos Bancários Atômicos

Desafio

1.

Criar duas contas bancárias

2.

Implementar transferência entre contas

3.

Garantir atomicidade da operação

4.

Simular cenários de erro

Registrar auditoria das transações

5.

# models.py - Definição de modelos

class Conta:

def \_\_init\_\_(self, id, saldo):

self.id = id

self.saldo = saldo

class Transacao:

def \_\_init\_\_(self, origem, destino, valor):

self.origem = origem

self.destino = destino

self.valor = valor

self.timestamp = datetime.now()

self.status = "pendente"

# transaction.py - Lógica de negócio

def transferir(origem, destino, valor):

# Implementar com tratamento de erros

# e garantir atomicidade

Estrutura do Código

Sistemas de Processamento de Linguagens

Entrada de Texto

Texto bruto para análise

Pré-processamento

Tokenização, limpeza, normalização

Análise Linguística

Parsing, POS tagging, entidades

Processamento Semântico

Extração de significado e relações

Exemplo: Compilador Simples em Python

import re

def tokenize(code):

# Definir padrões para tokens

token\_spec = [

('NUMBER', r'\d+'),

('PLUS', r'\+'),

('MINUS', r'-'),

('MULT', r'\\*'),

('DIV', r'/'),

('LPAREN', r'\('),

('RPAREN', r'\)'),

('WHITESPACE', r'\s+'),

]

# Combinar todos os padrões

token\_regex = '|'.join('(?P<%s>%s)' % pair for pair in token\_spec)

# Tokenizar o código

tokens = []

for match in re.finditer(token\_regex, code):

token\_type = match.lastgroup

token\_value = match.group()

if token\_type != 'WHITESPACE':

tokens.append((token\_type, token\_value))

return tokens

Analisador Léxico

class Parser:

def \_\_init\_\_(self, tokens):

self.tokens = tokens

self.pos = 0

def parse(self):

return self.expr()

def current\_token(self):

if self.pos < len(self.tokens):

return self.tokens[self.pos]

return None

def consume(self):

token = self.current\_token()

self.pos += 1

return token

def expr(self):

# Implementar parser recursivo

# para expressões matemáticas

term = self.term()

while self.current\_token() and self.current\_token()[0] in ['PLUS', 'MINUS']:

op = self.consume()[0]

right = self.term()

if op == 'PLUS':

term += right

else: # MINUS

term -= right

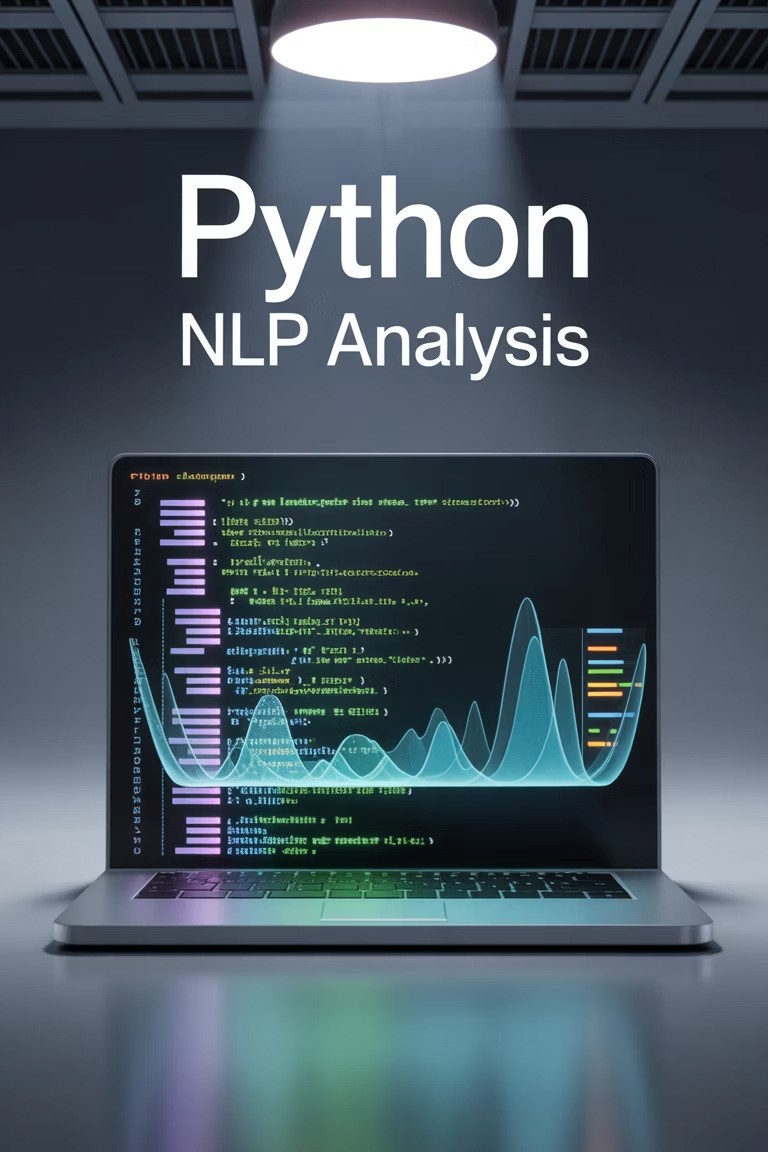
return term

def term(self):

# Implementar multiplicação e divisão

# ...

Parser Simples



Aplicação em Machine Learning:

NLP com SpaCy/NLTK

Pré-processamento

Tokenização

Remoção de stopwords

Normalização

Stemming/Lemmatization

Extração de Features

Bag-of-words

TF-IDF

Word embeddings

N-grams

Modelagem

Classificação de texto

Reconhecimento de entidades

Análise de sentimento

Sumarização

Exemplo: Pré-processamento Textual com SpaCy

import spacy

# Carregar modelo em português

nlp = spacy.load('pt\_core\_news\_sm')

def processar\_texto(texto):

# Processar o texto

doc = nlp(texto)

# Extrair tokens, removendo stopwords e pontuação

tokens = [token.lemma\_.lower() for token in doc

if not token.is\_stop and not token.is\_punct]

# Entidades nomeadas

entidades = [(ent.text, ent.label\_) for ent in doc.ents]

# Análise sintática

deps = [(token.text, token.dep\_, token.head.text)

for token in doc]

return {

'tokens': tokens,

'entidades': entidades,

'deps': deps

}

# Exemplo de uso

texto = "O presidente Lula visitou São Paulo na segunda-feira."

resultado = processar\_texto(texto)

print(f"Tokens: {resultado['tokens']}")

print(f"Entidades: {resultado['entidades']}")

Resultados

Código de Processamento

Para o texto:

"O presidente Lula visitou São Paulo na segunda-feira."

Tokens:

presidente

lula

visitar

são

paulo

segunda-feira

Entidades:

)

Lula, PER

(

)

São Paulo, LOC

(

segunda-feira, DATE

)

(

Exemplo: Geração de Texto com Modelos ML

from transformers import pipeline, set\_seed

# Carregar gerador de texto em português

gerador = pipeline('text-generation',

model='pierreguillou/gpt2-small-portuguese')

set\_seed(42)

# Gerar texto a partir de prompt

def gerar\_texto(prompt, max\_length=100):

resultados = gerador(prompt,

max\_length=max\_length,

num\_return\_sequences=1,

temperature=0.7)

texto\_gerado = resultados[0]['generated\_text']

return texto\_gerado

# Exemplo

prompt = "A inteligência artificial está transformando"

texto = gerar\_texto(prompt)

print(texto)

Aplicações Práticas

Código com Transformers

Geração de relatórios automáticos

Chatbots inteligentes

Sumarização de documentos

Tradução de idiomas

Assistentes virtuais

Criação de conteúdo

Os modelos transformers como BERT, GPT e T5 revolucionaram o processamento de linguagem

natural nos últimos anos.

Análise de Sentimento: Case prático com Python

0

30

60

90

Positivo

Neutro

Negativo

Exercício: Construir Pipeline NLP de Fim a Fim

1

Coleta de Dados

Extrair comentários de produtos de um site

Pré-processamento

Limpar e normalizar o texto usando SpaCy

Vetorização

Transformar texto em features usando TF-IDF

Modelagem

Treinar classificador para análise de sentimento

Avaliação

Medir desempenho e melhorar o modelo



Conclusão e Recomendações

Principais Aprendizados

Estruturação adequada de repositórios

Implementação de padrões cliente-

servidor

Organização de arquitetura de

aplicações

Garantia de transações confiáveis

Processamento avançado de linguagem

natural

Projetos Sugeridos

API de predição com modelo ML

Sistema distribuído com microserviços

Processador de transações bancárias

Aplicação de análise de sentimento

Pipeline completa de MLOps

Recursos Adicionais

Livros recomendados

Cursos online complementares

Comunidades para dúvidas

Ferramentas para prática

Repositórios de exemplo