

RODRIGO VIANNINI

POSTECH

MACHINE LEARNING ENGINEERING

PYTHON PARA ML E IA

AULA 05

SUMÁRIO

O QUE VEM POR AÍ?	3
HANDS ON	4
SAIBA MAIS.....	6
O QUE VOCÊ VIU NESTA AULA?	28
REFERÊNCIAS.....	29

EMSE

O QUE VEM POR AÍ?

Agora que já sabemos programar, vamos entender como criar uma interface de programação de aplicações, também conhecida como API. Iremos abordar os dois frameworks mais utilizados para o desenvolvimento destas aplicações: Flask e FastAPI.

Lembrando que as APIs conectam soluções e serviços sem a necessidade de saber como os elementos internos foram implementados.

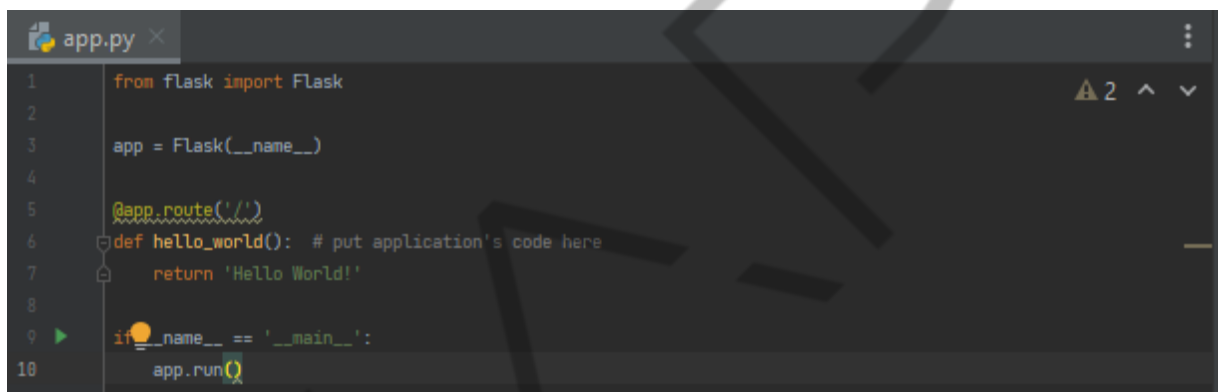
Ao final desta aula, ainda iremos introduzir os conceitos básicos dos principais frameworks de machine learning.

HANDS ON

Flask e FastAPI são dois frameworks populares em Python, cada um com suas características distintas no desenvolvimento de aplicações web. Enquanto o Flask destaca-se por sua simplicidade e flexibilidade, o FastAPI oferece uma abordagem moderna, com ênfase em tipagem de dados, automação de tarefas e performance.

Ambos os frameworks são escolhas comuns para especialistas em desenvolvimento, dependendo das necessidades específicas do projeto.

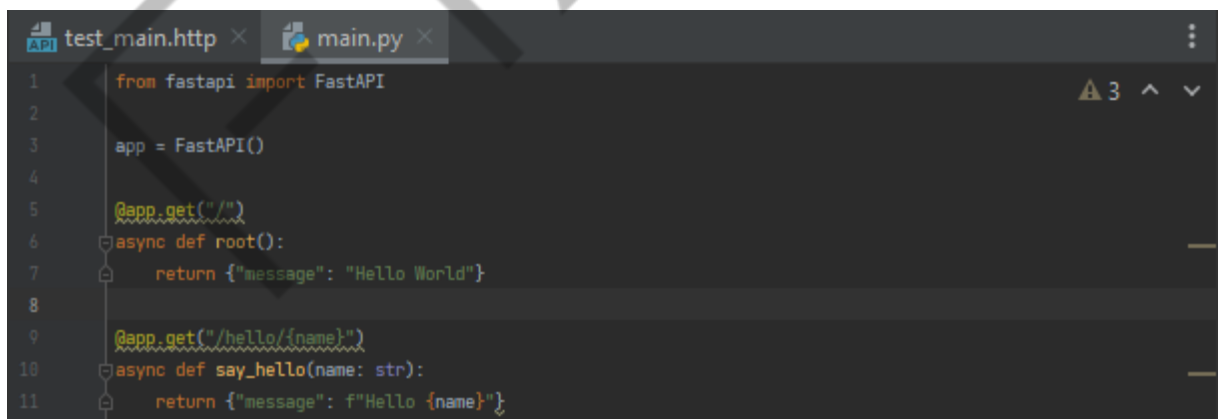
Exemplo:



```
1 from flask import Flask
2
3 app = Flask(__name__)
4
5 @app.route('/')
6 def hello_world(): # put application's code here
7     return 'Hello World!'
8
9 if __name__ == '__main__':
10     app.run()
```

Figura 1 – Exemplo de código-fonte Python (API em Flask)
Fonte: Elaborado pelo autor (2024)

Exemplo:



```
1 from fastapi import FastAPI
2
3 app = FastAPI()
4
5 @app.get("/")
6 async def root():
7     return {"message": "Hello World"}
8
9 @app.get("/hello/{name}")
10 async def say_hello(name: str):
11     return {"message": f"Hello {name}"}
```

Figura 2 – Exemplo de código-fonte Python (API em FastAPI)
Fonte: Elaborado pelo autor (2024)

Quando se trata de frameworks de Machine Learning em Python, Sklearn, Keras, TensorFlow e PyTorch são líderes de mercado. Eles fornecem ferramentas robustas para construir e treinar modelos de machine learning.

Exemplo:

```
1  from sklearn.linear_model import LinearRegression
2  import numpy as np
3
4  # Dados de exemplo para regressão linear
5  X = np.array([[1], [2], [3]])
6  y = np.array([2, 4, 5])
7
8  # Criando o modelo de regressão linear
9  model = LinearRegression()
10
11 # Treinando o modelo
12 model.fit(X, y)
13
14 # Realizando uma predição
15 prediction = model.predict([[4]])
16
17 print(f'Regressão Linear Prediction: {prediction[0]}')
18
```

Figura 3 – Exemplo de código-fonte Python (machine learning - regressão linear)
Fonte: Elaborado pelo autor (2024)

SAIBA MAIS

Agora que passamos pelos conceitos básicos do desenvolvimento de APIs, iremos nos aprofundar nos conceitos avançados que poderão ser úteis no seu dia-a-dia.

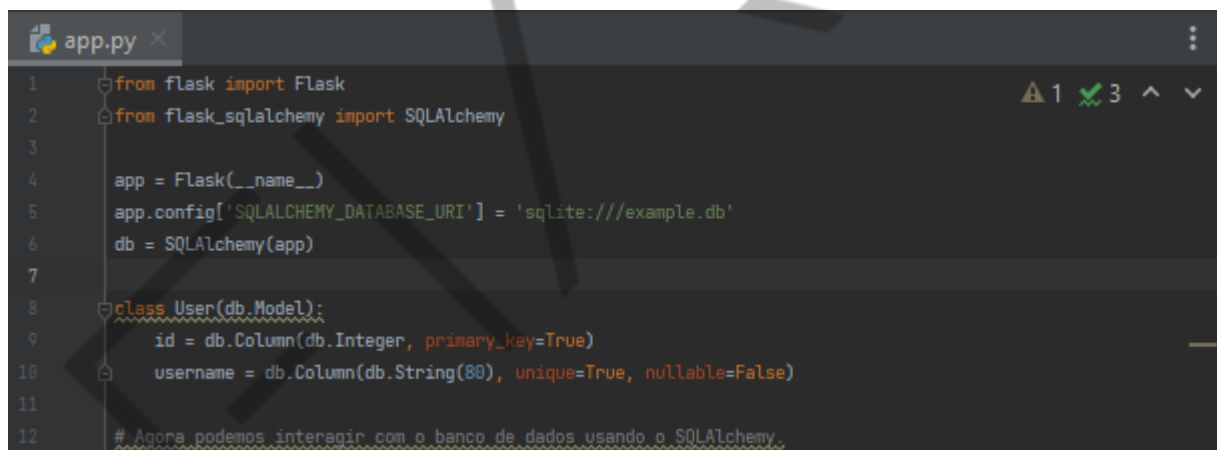
Flask: Micro-Framework Versátil

O Flask é conhecido por sua simplicidade e flexibilidade, mas também oferece recursos avançados para devs experientes. Vamos explorar alguns desses conceitos.

Extensões Flask

As extensões no Flask são módulos adicionais que estendem as funcionalidades do framework. Vamos considerar uma extensão popular, o “Flask_SQLAlchemy”, que facilita a integração com bancos de dados.

Exemplo:

A screenshot of a code editor window titled 'app.py'. The code is as follows:

```
1 from flask import Flask
2 from flask_sqlalchemy import SQLAlchemy
3
4 app = Flask(__name__)
5 app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///example.db'
6 db = SQLAlchemy(app)
7
8 class User(db.Model):
9     id = db.Column(db.Integer, primary_key=True)
10    username = db.Column(db.String(80), unique=True, nullable=False)
11
12 # Agora podemos interagir com o banco de dados usando o SQLAlchemy.
```

Figura 4 – Exemplo de código-fonte Python (Flask integração com banco de dados)
Fonte: Elaborado pelo autor (2024)

Interpretação

No código anterior importamos a classe “Flask” do Flask e a classe “SQLAlchemy” do Flask_SQLAlchemy. O Flask_SQLAlchemy é uma extensão que simplifica a integração do SQLAlchemy com o Flask, facilitando a interação com bancos de dados.

Um objeto Flask chamado app é criado. O primeiro argumento passado para o construtor é “__name__”, que representa o nome do módulo ou pacote principal. Isso é necessário para que o Flask saiba onde procurar por templates e arquivos estáticos.

A configuração do aplicativo é ajustada para definir a URL do banco de dados. Neste caso, estamos usando o “SQLite” e o banco de dados será um arquivo chamado “example.db” na mesma pasta do script.

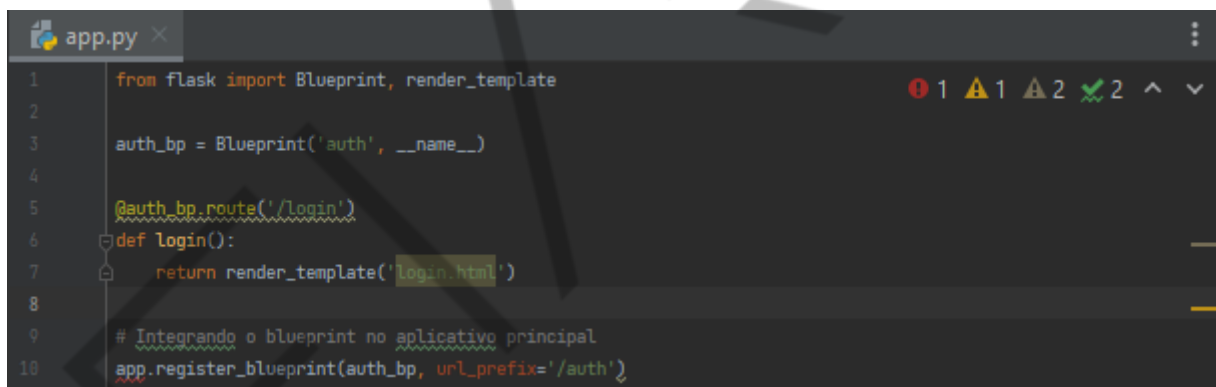
Um objeto chamado “db” é criado e associado ao aplicativo Flask. Isso configura a extensão para que ela saiba como interagir com o aplicativo Flask.

Uma classe “User” é definida, que herda da classe “db.Model” do SQLAlchemy. Esta classe representa uma tabela no banco de dados. No exemplo, a tabela terá uma coluna “id” como chave primária e uma coluna “username” que deve ser única e não pode ser nula.

Blueprints e Organização de Código

Para projetos maiores, organizar o código de maneira eficiente é crucial. Os blueprints no Flask ajudam a estruturar o aplicativo.

Exemplo:



```
1 from flask import Blueprint, render_template
2
3 auth_bp = Blueprint('auth', __name__)
4
5 @auth_bp.route('/login')
6 def login():
7     return render_template('login.html')
8
9 # Integrando o blueprint no aplicativo principal
10 app.register_blueprint(auth_bp, url_prefix='/auth')
```

Figura 5 – Exemplo de código-fonte Python (Flask com blueprints)
Fonte: Elaborado pelo autor (2024)

Interpretação

O código importa duas classes do Flask: “Blueprint” e “render_template”. Assim, Blueprint é uma maneira de organizar um conjunto de rotas relacionadas e views em um aplicativo Flask.

Um objeto Blueprint chamado “auth_bp” é criado. O primeiro argumento é o nome do blueprint e o segundo é o nome do módulo ou pacote. O blueprint é utilizado para agrupar rotas e views relacionadas.

Aqui, uma rota é definida para o caminho “/login”. Quando alguém acessa essa rota no navegador, a função login é executada. Neste caso, a função simplesmente

retorna o resultado da função “render_template”, que renderiza o modelo HTML chamado “login.html”. Isso geralmente significa que, ao acessar a rota “/login”, será exibida uma página HTML contida no arquivo “login.html”.

Aqui, o blueprint “auth_bp” é registrado no aplicativo Flask principal (app). A opção url_prefix=’/auth’ significa que todas as rotas definidas no blueprint terão o prefixo “/auth”. Portanto, a rota “/login” do blueprint será acessível através de “/auth/login” no aplicativo principal.

FastApi: eficiência e tipo de dados

O FastAPI, construído sobre o Starlette e o Pydantic, oferece uma abordagem moderna para o desenvolvimento de APIs. Vamos explorar alguns conceitos avançados.

Declaração de Modelos com Pydantic

O Pydantic é integrado ao FastAPI para a validação automática de dados.

Exemplo:



```
1 from fastapi import FastAPI
2 from pydantic import BaseModel
3
4 app = FastAPI()
5
6 class Item(BaseModel):
7     name: str
8     description: str = None
9     price: float
10    tax: float = None
```

Figura 6 – Exemplo de código-fonte Python (FastAPI com PyDantic)

Fonte: Elaborado pelo autor (2024)

Interpretação

Importamos as classes “FastAPI” e “BaseModel” do FastAPI e Pydantic, respectivamente.

Há a criação de uma instância do aplicativo FastAPI chamada “app” e a definição de uma classe chamada “Item” que herda de BaseModel do Pydantic.

Esta classe é usada para representar a estrutura de dados esperada para os itens manipulados pela API. Cada atributo da classe (name, description, price, tax) corresponde a um campo de dados associado a um item:

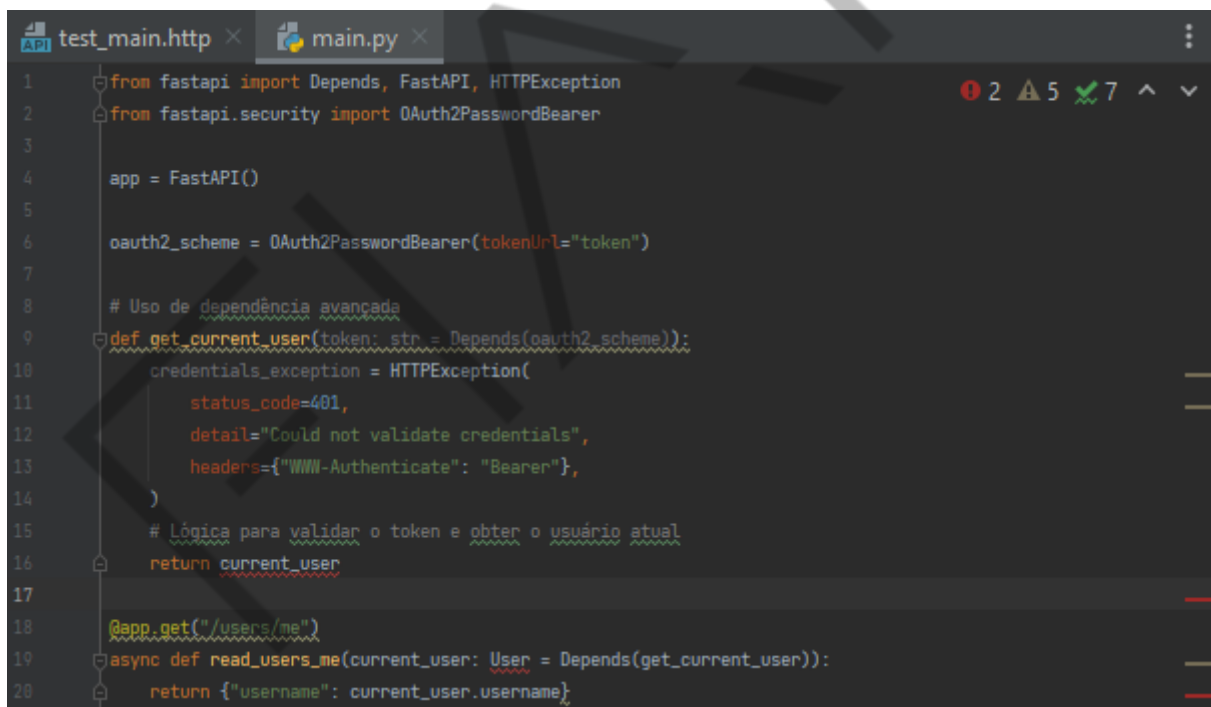
- **name:** um campo obrigatório do tipo string.
- **description:** um campo opcional do tipo string (o padrão é None se não fornecido).
- **price:** um campo obrigatório do tipo float.
- **tax:** um campo opcional do tipo float (o padrão é None se não fornecido).

Essa classe é usada para definir automaticamente a validação de dados para as entradas da API.

Dependências Avançadas

O FastAPI permite o uso de dependências avançadas para organizar o código e adicionar funcionalidades.

Exemplo:



```
1 from fastapi import Depends, FastAPI, HTTPException
2 from fastapi.security import OAuth2PasswordBearer
3
4 app = FastAPI()
5
6 oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")
7
8 # Uso de dependência avançada
9 def get_current_user(token: str = Depends(oauth2_scheme)):
10     credentials_exception = HTTPException(
11         status_code=401,
12         detail="Could not validate credentials",
13         headers={"WWW-Authenticate": "Bearer"},
14     )
15     # lógica para validar o token e obter o usuário atual
16     return current_user
17
18 @app.get("/users/me")
19 async def read_users_me(current_user: User = Depends(get_current_user)):
20     return {"username": current_user.username}
```

Figura 7 – Exemplo de código-fonte Python (FastAPI com autenticação por token)

Fonte: Elaborado pelo autor (2024)

Interpretação

No código anterior, importamos as classes necessárias do FastAPI, incluindo “Depends” para dependências, FastAPI para criar o aplicativo e “HTTPException” para

lidar com exceções HTTP. Também importa “OAuth2PasswordBearer” para autenticação via “OAuth2”.

Um objeto FastAPI chamado “app” é criado. Este é o aplicativo principal que irá conter as rotas e funcionalidades da API. Além disso, um esquema OAuth2 é configurado usando a classe OAuth2PasswordBearer.

Este esquema é usado para autenticação por senha, em que clientes podem obter um token de acesso usando um nome de usuário e senha, e o URL para obter o token é definido como “/token”.

Uma dependência avançada chamada “get_current_user” é definida. Esta dependência usa o esquema OAuth2 definido anteriormente para obter o token de autenticação. A lógica para validar o token e obter o usuário atual deve ser implementada.

Uma rota é definida em “/users/me”, que depende da função get_current_user. Esta rota está protegida por autenticação e somente usuários autenticados podem acessá-la. O usuário autenticado é passado como um parâmetro chamado “current_user”, que é do tipo “User” (presumidamente, uma classe de modelo definida em outro lugar no código).

Frameworks de Machine Learning

Scikit-Learn

O scikit-learn é conhecido por sua simplicidade e eficácia, mas também oferece uma variedade de ferramentas avançadas para desenvolvedoras e desenvolvedores experientes.

Pipelines para Pré-Processamento de Dados

Pipelines são uma maneira eficiente de encadear várias etapas de pré-processamento de dados e modelagem. Isso é crucial para manter o código organizado e facilitar a reprodução de experimentos.

Exemplo:

```
1 from sklearn.pipeline import Pipeline
2 from sklearn.preprocessing import StandardScaler
3 from sklearn.decomposition import PCA
4 from sklearn.ensemble import RandomForestClassifier
5
6 # Pipeline com escalonamento, redução de dimensionalidade e classificação
7 pipeline = Pipeline([
8     ('scaler', StandardScaler()),
9     ('pca', PCA(n_components=3)),
10    ('classifier', RandomForestClassifier())
11 ])
12
13 # Treinando o modelo e fazendo previsões
14 pipeline.fit(X_train, y_train)
15 predictions = pipeline.predict(X_test)
16
```

Figura 8 – Exemplo de código-fonte Python (pipeline com sklearn)
Fonte: Elaborado pelo autor (2024)

Interpretação

O código exposto importa classes específicas do “scikit-learn” necessárias para construir o pipeline. “Pipeline” é usado para criar um pipeline de processamento de dados e modelagem; “StandardScaler” é uma técnica de pré-processamento para escalonar os dados; “PCA” (Análise de Componentes Principais) é uma técnica de redução de dimensionalidade; e “RandomForestClassifier” é um algoritmo de classificação baseado em “ensemble” que utiliza árvores de decisão.

Um objeto Pipeline é criado com uma lista de tuplas, em que cada uma contém o nome de uma etapa no pipeline e a instância da classe correspondente. As etapas são:

- **scaler:** usa StandardScaler para escalonar os dados.
- **pca:** usa PCA para redução de dimensionalidade para três componentes.
- **classifier:** usa RandomForestClassifier como o classificador.

O pipeline é treinado com dados de treinamento (X_train, y_train) usando o método “fit”. Isso aplica as transformações de pré-processamento (escalonamento e redução de dimensionalidade) e treina o classificador.

Posteriormente, o modelo treinado é usado para fazer previsões nos dados de teste (X_test) usando o método “predict”. As previsões são armazenadas na variável predictions.

O uso de um pipeline é uma prática comum em aprendizado de máquina para organizar e automatizar o fluxo de trabalho, especialmente quando há etapas de pré-

processamento envolvidas. O pipeline garante que todas as etapas sejam aplicadas de maneira consistente tanto nos dados de treinamento quanto nos dados de teste.

Otimização de Hiperparâmetros com Grid Search

O Grid Search é uma técnica poderosa para encontrar a combinação ideal de hiperparâmetros para um modelo, o que resulta em um desempenho otimizado.

Exemplo:

```
1 from sklearn.model_selection import GridSearchCV
2
3 # Definindo os parâmetros a serem testados
4 param_grid = {
5     'classifier__n_estimators': [50, 100, 200],
6     'classifier__max_depth': [None, 10, 20]
7 }
8
9 # Criando um objeto GridSearchCV
10 grid_search = GridSearchCV(estimator=pipeline,
11                             param_grid=param_grid, cv=5)
12
13 # Executando a busca em grade
14 grid_search.fit(X_train, y_train)
15
16 # Obtendo os melhores hiperparâmetros
17 best_params = grid_search.best_params_
18
```

Figura 9 – Exemplo de código-fonte Python (otimização com sklearn)
Fonte: Elaborado pelo autor (2024)

Interpretação

Ele importa a classe “GridSearchCV”, que é usada para realizar busca em grade, e define um dicionário “param_grid” que especifica os hiperparâmetros a serem testados durante a busca em grade. Neste exemplo, dois hiperparâmetros do classificador RandomForest são considerados: “n_estimators” e “max_depth”. Serão testados diferentes valores para cada um desses hiperparâmetros.

Além disso, há a criação de um objeto GridSearchCV que será responsável pela busca em grade. Os argumentos são:

- **estimator:** o estimador (modelo) para o qual queremos encontrar os melhores hiperparâmetros. Neste caso, é o pipeline (pipeline) que foi definido anteriormente.
- **param_grid:** a grade de hiperparâmetros a ser testada.

- **cv**: o número de dobras (folds) a serem usadas na validação cruzada. Neste caso é 5, o que significa que o conjunto de dados será dividido em 5 partes e o treinamento/validação será realizado 5 vezes.

Ele ainda executa a busca em grade utilizando os dados de treinamento. O GridSearchCV treinará o modelo para todas as combinações possíveis de hiperparâmetros definidos em “param_grid” e usará a validação cruzada para avaliar o desempenho de cada configuração.

Obtemos os melhores hiperparâmetros encontrados durante a busca em grade. Esses hiperparâmetros são os que resultaram no melhor desempenho de acordo com a métrica de avaliação configurada (geralmente a acurácia, mas pode ser outra dependendo do problema).

Integração com Outros Frameworks e Bibliotecas

O scikit-learn é projetado para ser interoperável com outros frameworks e bibliotecas de machine learning, como TensorFlow e PyTorch. Isso permite a construção de modelos mais complexos e especializados.

Exemplo:

```
1 from sklearn.ensemble import RandomForestClassifier
2 from sklearn.preprocessing import StandardScaler
3 from sklearn.pipeline import make_pipeline
4 from sklearn2pmml import sklearn2pmml
5
6 # Criando um pipeline com scikit-learn
7 pipeline = make_pipeline(StandardScaler(), RandomForestClassifier())
8
9 # Treinando o modelo
10 pipeline.fit(X_train, y_train)
11
12 # Convertendo o modelo para PMML (Predictive Model Markup Language)
13 sklearn2pmml(pipeline, "model.pmml")
14
```

Figura 10 – Exemplo de código-fonte Python (pipeline: StandardScaler com RandomForestClassifier)
Fonte: Elaborado pelo autor (2024)

Interpretação

Ele importa as classes necessárias do scikit-learn para criar o pipeline, bem como “make_pipeline” para criar um pipeline de forma mais concisa e “sklearn2pmml” para converter o modelo para PMML.

Também utiliza “make_pipeline” para criar um pipeline composto por um StandardScaler (para escalonamento) seguido por um RandomForestClassifier (um classificador de árvores de decisão).

Ele treina o modelo usando os dados de treinamento (X_train, y_train) com o método fit do pipeline. O StandardScaler ajustará seus parâmetros com base nos dados de treinamento e o RandomForestClassifier será treinado nos dados escalonados.

Por fim, ele utiliza a função “sklearn2pmml” para converter o modelo treinado para o formato PMML. O primeiro argumento é o modelo (pipeline) e o segundo argumento é o caminho do arquivo em que o modelo PMML será salvo (neste caso, “model.pmml”).

O PMML é um padrão aberto que permite a interoperabilidade entre diferentes plataformas de software para modelos de machine learning. A conversão para PMML é útil quando você deseja implantar seu modelo em um ambiente que suporta esse formato, permitindo que o modelo treinado seja executado em diferentes sistemas sem a necessidade de recriar o modelo original.

Keras

O Keras é conhecido por sua interface de alto nível e versatilidade para desenvolvimento de modelos de deep learning. Vamos explorar alguns conceitos avançados.

Construção de Modelos com Camadas Personalizadas

O Keras permite a criação de modelos personalizados usando camadas customizadas, proporcionando maior flexibilidade para a construção de arquiteturas específicas.

Exemplo:

```
1 from keras.models import Model
2 from keras.layers import Input, Dense, concatenate
3
4 # Definindo camadas personalizadas
5 input_layer = Input(shape=(input_size,))
6 dense_layer1 = Dense(64, activation='relu')(input_layer)
7 dense_layer2 = Dense(32, activation='relu')(input_layer)
8
9 # Concatenando camadas
10 concatenated_layer = concatenate([dense_layer1, dense_layer2])
11
12 # Camada de saída
13 output_layer = Dense(output_size,
14                       activation='softmax')(concatenated_layer)
15
16 # Criando o modelo
17 custom_model = Model(inputs=[input_layer], outputs=output_layer)
18
```

Figura 11 – Exemplo de código-fonte Python (Keras)
Fonte: Elaborado pelo autor (2024)

Interpretação

Essa parte do código importa os módulos necessários da biblioteca Keras, que é uma biblioteca de alto nível para construção e treinamento de redes neurais em Python.

Aqui, são definidas as camadas do modelo. “Input” representa a camada de entrada e “Dense” representa uma camada densa (com ativação ReLU), concatenando as saídas dessas camadas (dense_layer1 e dense_layer2), ambas conectadas à camada de entrada.

As saídas das duas camadas densas são concatenadas usando a função “concatenate”. Isso significa que a saída final do modelo será uma combinação das características aprendidas por ambas as camadas densas.

Uma camada densa de saída é adicionada usando a saída concatenada como entrada e a ativação “softmax” é comumente usada em problemas de classificação para gerar probabilidades.

O modelo é criado usando “Model”, especificando as camadas de entrada e saída. Os inputs recebem a camada de entrada e os outputs recebem a camada de saída.

Esse tipo de arquitetura é útil em casos em que diferentes características da entrada devem ser processadas separadamente e, em seguida, combinadas para formar a saída final.

Treinamento de Modelos com Callbacks

Os callbacks no Keras permitem realizar ações específicas durante o treinamento do modelo, como salvamento de pesos e ajuste dinâmico de taxa de aprendizado, entre outros.

Exemplo:

```
1  from keras.callbacks import ModelCheckpoint, EarlyStopping
2
3  # Definindo callbacks
4  checkpoint = ModelCheckpoint('best_model.h5',
5                               save_best_only=True)
6
7  early_stopping = EarlyStopping(patience=3)
8
9  # Compilando e treinando o modelo com callbacks
10 custom_model.compile(optimizer='adam',
11                       loss='categorical_crossentropy',
12                       metrics=['accuracy'])
13 custom_model.fit(X_train,
14                 y_train,
15                 epochs=10,
16                 validation_split=0.2,
17                 callbacks=[
18                     checkpoint,
19                     early_stopping
20                 ]
21             )
22
```

Figura 12 – Exemplo de código-fonte Python (Keras com Callback)
Fonte: Elaborado pelo autor (2024)

Intepretação

Os módulos ModelCheckpoint e EarlyStopping são importados da biblioteca Keras. Esses são utilizados para definir callbacks durante o treinamento do modelo.

Definição de callbacks:

- **ModelCheckpoint:** este callback salva o modelo em um arquivo ('best_model.h5', neste caso) sempre que a métrica monitorada (por padrão, a perda) melhora. O argumento "save_best_only=True" garante que apenas o melhor modelo seja salvo.
- **EarlyStopping:** este callback é utilizado para o treinamento se a métrica monitorada não melhorar após um número específico de épocas (patience=3 significa que o treinamento será interrompido após 3 épocas sem melhoria).

Aqui, o modelo é compilado com um otimizador (“adam”), uma função de perda (“categorical_crossentropy”, comum em problemas de classificação multiclasse) e a métrica a ser monitorada durante o treinamento (“accuracy”).

Este modelo é treinado usando o método “fit”, em que:

- **X_train e y_train:** são os dados de treinamento.
- **epochs=10:** define o número de épocas de treinamento.
- **validation_split=0.2:** reserva 20% dos dados de treinamento para validação.
- **Call-backs:** inclui os callbacks definidos anteriormente (checkpoint e early_stopping).

Esses callbacks são estratégias comuns para melhorar o treinamento de modelos de machine learning, ajudando a evitar overfitting e a salvar o melhor modelo durante o treinamento.

Transferência de Aprendizagem com Modelos Pré-treinados

A transferência de aprendizagem é uma técnica poderosa. O Keras facilita a utilização de modelos pré-treinados para tarefas específicas.

Exemplo:

```
1 from keras.applications import VGG16
2 from keras.layers import Flatten
3
4 # Carregando modelo pré-treinado (VGG16 neste exemplo)
5 base_model = VGG16(weights='imagenet',
6                     include_top=False,
7                     input_shape=(
8                         224,
9                         224,
10                        3
11                    )
12                )
13
14 # Adicionando camadas personalizadas no topo do modelo pré-treinado
15 top_model = Flatten()(base_model.output)
16 output_layer = Dense(output_size,
17                      activation='softmax')(top_model)
18
19 # Criando o modelo final
20 transfer_model = Model(inputs=base_model.input,
21                       outputs=output_layer)
22
```

Figura 13 – Exemplo de código-fonte Python (Keras com modelos pré-treinados)
Fonte: Elaborado pelo autor (2024)

Interpretação

Importamos os módulos necessários da biblioteca Keras, incluindo a arquitetura pré-treinada “VGG16” e a camada “Flatten” que será adicionada ao topo.

Carregamos o modelo VGG16 pré-treinado nos pesos da “ImageNet” (o modelo VGG16 é pré-treinado nas imagens da “ImageNet”). O argumento “include_top=False” indica que a camada totalmente conectada no topo do modelo original não será incluída, visto que planejamos adicionar nossas próprias camadas personalizadas.

Adição de Camadas Personalizadas:

- **Flatten()**: adiciona uma camada de achatamento que transforma os dados de saída do modelo VGG16 em um vetor unidimensional.
- **Dense(output_size, activation='softmax')**: adiciona uma camada densa de saída com ativação “softmax”, com o “output_size” representando o número de classes no problema de classificação.

Há também a criação do modelo final utilizando a classe Model, especificando as camadas de entrada (base_model.input) e saída (output_layer).

Nesta exploração de conceitos avançados em Keras, abordamos a construção de modelos com camadas personalizadas, treinamento com callbacks e transferência de aprendizagem com modelos pré-treinados.

TensorFlow

O TensorFlow é um poderoso framework de machine learning e deep learning desenvolvido pela Google. É conhecido por sua versatilidade e escalabilidade, sendo utilizado em uma variedade de cenários de machine learning. Vamos explorar alguns conceitos avançados.

Construção de Grafos Computacionais com TensorFlow 2.X

O TensorFlow 2.x introduziu o modo de execução eager, tornando a construção de modelos mais intuitiva e flexível.

Exemplo:

```
1 import tensorflow as tf
2 from tensorflow.keras import layers
3
4 # Construindo um modelo sequencial
5 model = tf.keras.Sequential([
6     layers.Dense(64, activation='relu', input_shape=(input_size,)),
7     layers.Dense(32, activation='relu'),
8     layers.Dense(output_size, activation='softmax')
9 ])
10
```

Figura 14 – Exemplo de código-fonte Python (TensorFlow com camadas de ativação)
Fonte: Elaborado pelo autor (2024)

Interpretação

Importa os módulos necessários da biblioteca TensorFlow, incluindo a submódulo layers que contém várias camadas Keras.

Construção do Modelo Sequencial:

- **tf.keras.Sequential:** cria um modelo sequencial, em que as camadas são empilhadas uma sobre a outra.
- **layers.Dense(64, activation='relu', input_shape=(input_size,)):** adiciona uma camada densa (totalmente conectada) com 64 unidades e ativação ReLU. A camada de entrada é especificada com input_shape=(input_size,).
- **layers.Dense(32, activation='relu'):** adiciona uma segunda camada densa com 32 unidades e ativação ReLU.
- **layers.Dense(output_size, activation='softmax'):** adiciona a camada de saída com output_size unidades (representando o número de classes em um problema de classificação) e ativação softmax, comumente usada para problemas de classificação multiclasse.

Este código cria um modelo sequencial simples com duas camadas densas ocultas e uma camada de saída softmax. Esse tipo de modelo é comumente utilizado para tarefas de classificação, em que a entrada é transformada por camadas densas com ativação ReLU e a saída é mapeada para probabilidades usando a ativação softmax.

Treinamento de Modelos com TensorFlow

O TensorFlow facilita o treinamento de modelos com otimizadores customizados e loops de treinamento personalizados.

Exemplo:

```
1 # Compilando o modelo
2 model.compile(optimizer='adam',
3               loss='categorical_crossentropy',
4               metrics=['accuracy'])
5
6 # Treinando o modelo
7 model.fit(X_train,
8         y_train,
9         epochs=10,
10        batch_size=32
11        )
12
```

Figura 15 – Exemplo de código-fonte Python (TensorFlow - compilando o modelo)

Fonte: Elaborado pelo autor (2024)

Intepretação

Compilação do Modelo:

- **optimizer='adam':** especifica o otimizador a ser usado durante o treinamento. Neste caso, é o otimizador Adam, que é uma escolha comum devido à sua eficácia em muitos cenários.
- **loss='categorical_crossentropy':** define a função de perda a ser otimizada durante o treinamento. No contexto de problemas de classificação multiclasse, a entropia cruzada categórica é frequentemente utilizada.
- **metrics=['accuracy']:** indica a métrica a ser avaliada durante o treinamento. Neste caso, é a precisão (accuracy), que é uma medida comum para problemas de classificação.

Treinamento do Modelo:

- **X_train e y_train:** representam os dados de treinamento e os rótulos correspondentes.
- **epochs=10:** especifica o número de épocas de treinamento, ou seja, quantas vezes o modelo passará por todos os dados de treinamento.
- **batch_size=32:** indica o tamanho do lote (batch) utilizado durante cada atualização dos pesos do modelo. O treinamento é realizado em lotes para melhorar a eficiência computacional.

Tensorflow Extendido: Tensorboard e Salvamento De Modelos

O TensorBoard é uma ferramenta de visualização poderosa integrada ao TensorFlow para monitorar o treinamento do modelo.

Exemplo:

```
1 # Configurando o uso do TensorBoard
2 tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir='./logs')
3
4 # Treinando o modelo com TensorBoard
5 model.fit(X_train,
6           y_train,
7           epochs=10,
8           batch_size=32,
9           callbacks=[
10              tensorboard_callback
11           ])
12
13
```

Figura 16 – Exemplo de código-fonte Python (TensorFlow - TensorBoard)
Fonte: Elaborado pelo autor (2024)

Interpretação

Configuração do Callback do TensorBoard:

- **tf.keras.callbacks.TensorBoard:** cria um callback para o TensorBoard, que é uma ferramenta de visualização incluída no TensorFlow para monitorar o treinamento de modelos.
- **log_dir='./logs':** especifica o diretório em que os logs do TensorBoard serão armazenados. Você pode substituir ./logs pelo caminho desejado.

Treinamento do Modelo com o Callback do TensorBoard:

- **callbacks=[tensorboard_callback]:** adiciona o callback do TensorBoard durante o treinamento. Isso permite que o TensorBoard capture métricas e visualize o progresso do treinamento.

Execução do TensorBoard:

- Após o treinamento, você pode executar o TensorBoard no terminal usando o seguinte comando:

```
tensorboard -- logdir=./logs
```

Figura 17 – Execução do TensorBoard para observar os logs
Fonte: Elaborado pelo autor (2024)

Isso iniciará o TensorBoard e você poderá acessá-lo pelo navegador no endereço <http://localhost:6006/>. Lá, você poderá visualizar gráficos e métricas relacionadas ao treinamento do modelo.

O TensorBoard é uma ferramenta poderosa para análise visual durante o treinamento de modelos, permitindo a inspeção de métricas, gráficos de arquitetura, distribuição de parâmetros e muito mais. O callback do TensorBoard no treinamento do modelo facilita a geração dessas informações para avaliação e ajuste do desempenho do modelo.

Integração com GPUs E TPUs

O TensorFlow oferece suporte eficiente para treinamento distribuído em GPUs e TPUs, permitindo escalabilidade em larga escala.

Exemplo:

```
1 # Configurando para treinamento em GPU
2 gpus = tf.config.experimental.list_physical_devices('GPU')
3 if gpus:
4     tf.config.experimental.set_memory_growth(gpus[0], True)
5
6 # Configurando para treinamento em TPU (caso disponível)
7 resolver = tf.distribute.cluster_resolver.TPUClusterResolver(
8     tpu='grpc://10.0.0.2:8470'
9 )
10 tf.tpu.experimental.initialize_tpu_system(resolver)
11 strategy = tf.distribute.experimental.TPUStrategy(resolver)
12
```

Figura 18 – Exemplo de código-fonte Python (GPUs e TPUs)
Fonte: Elaborado pelo autor (2024)

Interpretação

Configuração para Treinamento em GPU:

- **`tf.config.experimental.list_physical_devices('GPU')`**: obtém a lista de GPUs físicas disponíveis no sistema.
- **`tf.config.experimental.set_memory_growth(gpus[0], True)`**: configura o crescimento dinâmico da memória da GPU. Isso permite que a GPU aloque mais memória conforme necessário durante o treinamento, evitando a alocação estática que pode levar à falta de memória.

Configuração para Treinamento em TPU (caso disponível):

- **tf.distribute.cluster_resolver.TPUClusterResolver:** cria um resolvidor de cluster para a TPU, especificando o endereço da TPU no formato 'grpc://[endereço]:[porta]'.
- **tf.tpu.experimental.initialize_tpu_system(resolver):** inicializa o sistema TPU com base no resolvidor configurado.
- **tf.distribute.experimental.TPUStrategy(resolver):** cria uma estratégia de treinamento distribuído para a TPU, permitindo a distribuição eficiente do treinamento em múltiplos núcleos da TPU.

Essas configurações são úteis para aproveitar os recursos computacionais disponíveis, seja em GPU ou TPU, otimizando o treinamento do modelo. É importante notar que a execução dessas configurações requer a presença dos recursos correspondentes (GPU ou TPU) no sistema. O treinamento em GPU ou TPU pode acelerar significativamente o processo de treinamento, especialmente para modelos mais complexos ou conjuntos de dados grandes.

Pytorch

O PyTorch é conhecido por sua flexibilidade e expressividade, sendo escolhido por muitos indivíduos pesquisadores e desenvolvedores. Vamos explorar alguns conceitos avançados.

Construção Dinâmica de Grafos Computacionais

A característica dinâmica de construção de grafos computacionais no PyTorch permite uma abordagem mais flexível na definição e modificação de modelos.

Exemplo:

```
1 import torch
2 import torch.nn as nn
3
4 # Definindo um modelo com construção dinâmica
5 class DynamicModel(nn.Module):
6     def __init__(self, input_size, hidden_size, output_size):
7         super(DynamicModel, self).__init__()
8         self.layer1 = nn.Linear(input_size, hidden_size)
9         self.layer2 = nn.Linear(hidden_size, output_size)
10
11     def forward(self, x):
12         x = torch.relu(self.layer1(x))
13         return self.layer2(x)
14
15 # Criando uma instância do modelo
16 model = DynamicModel(input_size=10, hidden_size=64, output_size=5)
17
```

Figura 19 – Exemplo de código-fonte Python (PyTorch)
Fonte: Elaborado pelo autor (2024)

Intepretação

Importamos os módulos necessários do PyTorch, incluindo o módulo “nn”, que contém as ferramentas para a construção de redes neurais.

Definição da Classe do Modelo:

- **nn.Module:** a classe base para todos os modelos PyTorch. A classe `DynamicModel` herda dela.
- **__init__:** é o método de inicialização em que as camadas do modelo são definidas. Neste caso, são duas camadas lineares (`nn.Linear`) representando uma camada oculta (`layer1`) e uma camada de saída (`layer2`).
- **forward:** o método que define a passagem direta (forward pass) do modelo. Define como os dados de entrada `x` passam através das camadas. Neste caso, a ativação ReLU é aplicada após a primeira camada.

Cria uma instância do modelo `DynamicModel` com dimensões de entrada de 10, uma camada oculta de tamanho 64 e uma camada de saída de tamanho 5.

Autograd e Otimizadores Flexíveis

O mecanismo de Autograd do PyTorch permite o cálculo automático de gradientes, facilitando o treinamento de modelos customizados.

Exemplo:


```
1 # Definindo dados e parâmetros
2 inputs = torch.randn(10, requires_grad=True)
3 weights = torch.randn(10, requires_grad=True)
4 target = torch.randn(10)
5
6 # Computando a perda e realizando a retropropagação
7 loss = torch.nn.functional.mse_loss(inputs * weights, target)
8 loss.backward()
9
10 # Atualizando os pesos usando um otimizador
11 optimizer = torch.optim.SGD([weights], lr=0.01)
12 optimizer.step()
13
```

Figura 20 – Exemplo de código-fonte Python (otimizadores do PyTorch)

Fonte: Elaborado pelo autor (2024)

Interpretação

Definindo Dados e Parâmetros:

- **inputs:** tensor de entrada com valores aleatórios. O argumento `requires_grad=True` indica que os gradientes em relação a este tensor serão calculados durante a retropropagação.
- **weights:** tensor de pesos com valores aleatórios, também configurado para calcular gradientes.
- **target:** tensor de alvo com valores aleatórios.

Computando a Perda e Realizando a Retropropagação:

- **torch.nn.functional.mse_loss:** calcula a perda usando a função de erro médio quadrático (MSE) entre a multiplicação dos tensores `inputs` e `weights` e o tensor `target`.
- **loss.backward():** inicia a retropropagação, calculando os gradientes dos tensores com relação à perda.

Atualizando os Pesos Usando um Otimizador:

- **torch.optim.SGD:** cria um otimizador Stochastic Gradient Descent (SGD) para atualizar os pesos. O argumento `lr=0.01` define a taxa de aprendizado.
- **optimizer.step():** atualiza os parâmetros do modelo com base nos gradientes calculados durante a retropropagação.

Treinamento Distribuído e Torch.Dataparallel

O PyTorch oferece suporte para treinamento distribuído, permitindo a escala para várias GPUs.

Exemplo:

```
1 # Configurando treinamento distribuído
2 import torch.distributed as dist
3 dist.init_process_group(backend='nccl')
4
5 # Envolvendo o modelo com o Torch DataParallel
6 model = nn.DataParallel(model)
7
```

Figura 21 – Exemplo de código-fonte Python (escalando GPUs com PyTorch)
Fonte: Elaborado pelo autor (2024)

Interpretação

Configuração do Treinamento Distribuído:

- **torch.distributed:** módulo que fornece suporte para treinamento distribuído no PyTorch.
- **dist.init_process_group(backend='nccl'):** inicializa o grupo de processos para treinamento distribuído. O argumento `backend='nccl'` indica o uso do backend NCCL (NVIDIA Collective Communication Library) para comunicação entre GPUs.

Envolvimento do Modelo com o Torch DataParallel:

- **nn.DataParallel:** envolvendo o modelo com `DataParallel`, permite o treinamento paralelo em várias GPUs. Cada GPU processará uma porção dos dados e contribuirá para o cálculo dos gradientes durante a retropropagação.

O treinamento distribuído é útil para acelerar o treinamento em conjuntos de dados grandes ou modelos complexos, distribuindo o trabalho entre várias GPUs ou dispositivos. O backend NCCL é comumente usado quando se trabalha com GPUs da NVIDIA.

É importante notar que para usar `DataParallel`, o modelo e os dados precisam ser distribuídos entre as GPUs. O treinamento distribuído geralmente é usado em ambientes com várias GPUs e a inicialização do grupo de processos

(`init_process_group`) é uma etapa crucial para garantir a comunicação eficiente entre eles.

Integração com Libras e Frameworks Externos

PyTorch é conhecido por sua integração eficiente com outras bibliotecas e frameworks, como LIBRAS (Libraries for Deep Learning Research) e ONNX (Open Neural Network Exchange).

Exemplo:

```
1 # Exportando modelo PyTorch para o formato ONNX
2 dummy_input = torch.randn(1, input_size)
3 onnx_path = "model.onnx"
4 torch.onnx.export(model, dummy_input, onnx_path)
5
```

Figura 22 – Exemplo de código-fonte Python (exportando modelo do PyTorch)
Fonte: Elaborado pelo autor (2024)

Interpretação

Criando Dados de Entrada de Exemplo:

- **`torch.randn(1, input_size)`:** gera dados de entrada de exemplo (`dummy_input`) com uma forma de `(1, input_size)`. Esses dados são usados para criar um exemplo de entrada durante a exportação do modelo ONNX.

Exportando o Modelo para ONNX:

- **`onnx_path = "model.onnx"`:** define o caminho e o nome do arquivo ONNX em que o modelo será salvo.
- **`torch.onnx.export(model, dummy_input, onnx_path)`:** exporta o modelo para o formato ONNX. O primeiro argumento é o modelo PyTorch (`model`), o segundo é o exemplo de entrada (`dummy_input`) e o terceiro é o caminho do arquivo ONNX (`onnx_path`).

O QUE VOCÊ VIU NESTA AULA?

Nessa aula, conhecemos dois frameworks muito utilizados na construção de APIs (Flask e FastAPI), alguns conceitos avançados, como a integração com banco de dados, e exploramos os conceitos introdutórios aos algoritmos de machine learning.

Apresentamos exemplos de estrutura de código utilizando os frameworks mais populares de machine learning acompanhados das interpretações do código.

Esta abordagem permitiu-nos desenvolver habilidades na leitura e compreensão de códigos, além de fornecer insights sobre a funcionalidade subjacente.

REFERÊNCIAS

BROWNLEE, J. **Deep Learning with Keras**: Implementing deep learning models and neural networks with the power of Python. [s.l.]: Machine Learning Mastery, 2017.

GRINBERG, M. **Flask Web Development**: Developing Web Applications with Python. [s.l.]: O'Reilly Media, 2018.

GULLI, A.; KAPOOR, A.; PAL, S. **Deep Learning with TensorFlow 2 and Keras**. [s.l.]: Packt Publishing, 2019.

PEDREGOSA, F. et al. **Scikit-learn**: Machine Learning in Python. [s.l.]: Journal of Machine Learning Research, 2011.

RAO, K.; R. and M. KELLEHER. **Deep Learning with PyTorch**. [s.l.]: Manning Publications, 2018.

TITUS, S. **FastAPI**: The complete guide. [s.l.]: Packt Publishing, 2021.

PALAVRAS-CHAVE

Palavras-chave: API. Flask. FastAPI. Framework de Machine Learning.

EMSE



POSTECH