Fundamentos da Engenharia de Features

> *"Dados não são apenas números, são histórias esperando para serem contadas. A engenharia de features é como aprender a linguagem dessas histórias."*

O que é Engenharia de Features?

A engenharia de features é o processo de usar conhecimento de domínio e técnicas de transformação para extrair características (features) dos dados brutos que melhor representam os padrões subjacentes que um modelo de machine learning precisa aprender. É uma etapa crítica que conecta os dados brutos aos algoritmos de aprendizado.

Em termos simples, a engenharia de features transforma variáveis de entrada em um formato que permite que algoritmos de machine learning funcionem de maneira mais eficaz. Isso pode envolver:

- Criação de novas variáveis a partir das existentes
- Transformação de variáveis para melhor capturar relações
- Codificação de variáveis categóricas em formatos numéricos
- Seleção das features mais relevantes para o problema

Por que a Engenharia de Features é Crucial?

A importância da engenharia de features não pode ser subestimada. Aqui estão algumas razões pelas quais ela é considerada uma das etapas mais valiosas no desenvolvimento de modelos:

1. **Melhora o desempenho do modelo**: Features bem projetadas podem capturar relações complexas que algoritmos simples não conseguiriam detectar sozinhos.
2. **Reduz a complexidade computacional**: Com features melhores, você pode frequentemente usar modelos mais simples e ainda obter resultados excelentes.
3. **Incorpora conhecimento de domínio**: Permite que especialistas no assunto contribuam diretamente para o processo de machine learning.
4. **Lida com dados problemáticos**: Ajuda a tratar valores ausentes, outliers e outras anomalias nos dados.
5. **Aumenta a interpretabilidade**: Features bem projetadas podem tornar os modelos mais fáceis de entender e explicar.

```
# Exemplo: Comparação de desempenho com e sem engenharia de features
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error
```

```
import numpy as np

# Dados sem engenharia de features
X_raw = df[['temp', 'humidity', 'wind_speed']]
y = df['target']

# Dados com engenharia de features
X_engineered = df[['temp', 'humidity', 'wind_speed',
                   'temp_squared', 'humidity_squared',
                   'temp_humidity_interaction', 'is_weekend']]

# Treinar modelos
model_raw = RandomForestRegressor(random_state=42)
model_engineered = RandomForestRegressor(random_state=42)

model_raw.fit(X_raw, y)
model_engineered.fit(X_engineered, y)

# Avaliar desempenho
mse_raw = mean_squared_error(y, model_raw.predict(X_raw))
mse_engineered = mean_squared_error(y, model_engineered.predict(X_engineered))

print(f"MSE sem engenharia de features: {mse_raw:.4f}")
print(f"MSE com engenharia de features: {mse_engineered:.4f}")
print(f"Melhoria: {(1 - mse_engineered/mse_raw) * 100:.2f}%")
```

O Processo de Engenharia de Features

O processo de engenharia de features geralmente segue estas etapas:

1. **Exploração e compreensão dos dados**: Antes de criar features, você precisa entender profundamente seus dados.
2. **Limpeza de dados**: Tratar valores ausentes, outliers e erros nos dados.
3. **Transformação de features**: Aplicar operações matemáticas para criar novas features.
4. **Codificação de variáveis categóricas**: Converter variáveis categóricas em representações numéricas.
5. **Seleção de features**: Escolher as features mais relevantes para o modelo.
6. **Validação**: Testar se as features melhoram o desempenho do modelo.
7. **Iteração**: Refinar as features com base nos resultados.

> ***Dica**: A engenharia de features é um processo iterativo. Não espere acertar na primeira tentativa. Experimente diferentes abordagens e avalie seu impacto no desempenho do modelo.*

Técnicas de Engenharia de Features com scikit-learn

Nesta seção, exploraremos técnicas práticas de engenharia de features usando a biblioteca scikit-learn. Utilizaremos dados climáticos como exemplo, mas as técnicas podem ser aplicadas a diversos domínios.

Configuração do Ambiente

Antes de começarmos, vamos configurar nosso ambiente com as bibliotecas necessárias:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import PolynomialFeatures, OneHotEncoder, StandardScaler
from sklearn.feature_selection import VarianceThreshold
from datetime import datetime, timedelta

# Configurações de visualização
plt.style.use('ggplot')
sns.set_palette('viridis')
```

Dados de Exemplo: Informações Climáticas

Para este capítulo, trabalharemos com dados climáticos simulados. Em um ambiente de produção, você poderia obter esses dados de APIs como OpenWeatherMap ou de conjuntos de dados históricos.

```
def simulate_weather_data(cities, days=10):
    """Simula dados climáticos para uma lista de cidades."""
    np.random.seed(42)  # Para reprodutibilidade
    weather_conditions = ['Clear', 'Clouds', 'Rain', 'Snow', 'Thunderstorm', 'Drizzle',

    data = []
    for city in cities:
        for day in range(days):
            data.append({
```

```
                'city': city,
                'date': pd.Timestamp('2023-10-01') + pd.Timedelta(days=day),
                'temp': np.random.uniform(5, 35),  # Temperatura em Celsius
                'feels_like': np.random.uniform(3, 37),  # Sensação térmica
                'temp_min': np.random.uniform(2, 30),  # Temperatura mínima
                'temp_max': np.random.uniform(10, 40),  # Temperatura máxima
                'pressure': np.random.uniform(990, 1030),  # Pressão atmosférica
                'humidity': np.random.uniform(30, 100),  # Umidade
                'wind_speed': np.random.uniform(0, 20),  # Velocidade do vento
                'wind_deg': np.random.uniform(0, 360),  # Direção do vento
                'clouds': np.random.uniform(0, 100),  # Cobertura de nuvens
                'weather_condition': np.random.choice(weather_conditions)  # Condição c
            })

    return pd.DataFrame(data)

# Lista de cidades para simular dados
cities = ['New York', 'London', 'Tokyo', 'Sydney', 'Rio de Janeiro', 'Paris', 'Moscow',

# Gerar dados simulados
weather_df = simulate_weather_data(cities)
```

Vamos examinar os primeiros registros do nosso conjunto de dados:

```
weather_df.head()
```

Tabela 7-1. Primeiras linhas do DataFrame de dados climáticos

Explorando os Dados

Antes de aplicar técnicas de engenharia de features, é essencial entender os dados com os quais estamos trabalhando:

```
# Informações básicas sobre o DataFrame
print("Informações do DataFrame:")
weather_df.info()

print("\nEstatísticas descritivas:")
weather_df.describe()
```

Visualizar a distribuição das variáveis numéricas pode fornecer insights valiosos:

```
fig, axes = plt.subplots(2, 2, figsize=(12, 10))

sns.histplot(weather_df['temp'], kde=True, ax=axes[0, 0])
axes[0, 0].set_title('Distribuição de Temperatura')

sns.histplot(weather_df['humidity'], kde=True, ax=axes[0, 1])
axes[0, 1].set_title('Distribuição de Umidade')

sns.histplot(weather_df['wind_speed'], kde=True, ax=axes[1, 0])
axes[1, 0].set_title('Distribuição de Velocidade do Vento')

sns.histplot(weather_df['pressure'], kde=True, ax=axes[1, 1])
axes[1, 1].set_title('Distribuição de Pressão Atmosférica')

plt.tight_layout()
plt.show()
```

Figura 7-1. Distribuição das variáveis climáticas

Técnica 1: Extração de Features Temporais

Dados temporais frequentemente contêm padrões sazonais e cíclicos que podem ser capturados através da extração de componentes de data e hora:

```
# Extrair features temporais
weather_df['day_of_week'] = weather_df['date'].dt.dayofweek
weather_df['is_weekend'] = weather_df['day_of_week'].apply(lambda x: 1 if x >= 5 else 0
weather_df['month'] = weather_df['date'].dt.month
weather_df['day'] = weather_df['date'].dt.day

# Definir estação do ano (simplificado para hemisfério norte)
def get_season(month):
    if month in [12, 1, 2]:
        return 'Winter'
    elif month in [3, 4, 5]:
        return 'Spring'
    elif month in [6, 7, 8]:
        return 'Summer'
    else:  # 9, 10, 11
        return 'Fall'

weather_df['season'] = weather_df['month'].apply(get_season)
```

> ***Nota**: Em aplicações reais, considere a localização geográfica ao definir estações do ano. O código acima é simplificado para o hemisfério norte.*

Técnica 2: Features Polinomiais

As features polinomiais são úteis para capturar relações não-lineares nos dados. O scikit-learn fornece a classe `PolynomialFeatures` para criar essas transformações automaticamente:

```
# Selecionar colunas numéricas para aplicar polynomial features
numerical_cols = ['temp', 'humidity', 'wind_speed', 'pressure']
X_numerical = weather_df[numerical_cols]

# Aplicar polynomial features (grau 2)
poly = PolynomialFeatures(degree=2, include_bias=False)
poly_features = poly.fit_transform(X_numerical)

# Obter os nomes das features polinomiais
poly_feature_names = poly.get_feature_names_out(numerical_cols)

# Criar DataFrame com as features polinomiais
poly_df = pd.DataFrame(poly_features, columns=poly_feature_names)

# Exibir as primeiras linhas
poly_df.head()
```

Tabela 7-2. Features polinomiais geradas

As features polinomiais incluem: - Features originais (temp, humidity, etc.) - Termos quadráticos ($temp^2$, $humidity^2$, etc.) - Termos de interação (temp × humidity, temp × wind_speed, etc.)

> ***Atenção**: Features polinomiais podem aumentar significativamente a dimensionalidade dos dados. Para grau d e n features originais, você terá aproximadamente n^d features. Use com cautela para evitar overfitting.*

Técnica 3: One-Hot Encoding

Variáveis categóricas precisam ser convertidas em formato numérico antes de serem usadas em modelos de machine learning. O one-hot encoding é uma técnica comum para isso:

```
# Selecionar colunas categóricas para aplicar one-hot encoding
categorical_cols = ['weather_condition', 'season', 'city']
X_categorical = weather_df[categorical_cols]

# Aplicar one-hot encoding
encoder = OneHotEncoder(sparse=False, drop='first')  # drop='first' para evitar multico
encoded_features = encoder.fit_transform(X_categorical)

# Obter os nomes das features codificadas
encoded_feature_names = encoder.get_feature_names_out(categorical_cols)

# Criar DataFrame com as features codificadas
encoded_df = pd.DataFrame(encoded_features, columns=encoded_feature_names)

# Exibir as primeiras linhas
encoded_df.head()
```

Tabela 7-3. Features categóricas após one-hot encoding

> ***Dica**: Use* `drop='first'` *para evitar a "armadilha da variável dummy", que pode causar multicolinearidade. Isso remove uma categoria de cada variável categórica.*

Técnica 4: Features de Interação Personalizadas

Além das interações automáticas geradas por `PolynomialFeatures`, podemos criar features de interação personalizadas baseadas em conhecimento de domínio:

```
# Criar features de interação manualmente
weather_df['temp_humidity_ratio'] = weather_df['temp'] / weather_df['humidity']
weather_df['heat_index'] = weather_df['temp'] * (1 + 0.01 * weather_df['humidity'])  #

# Fórmula simplificada para sensação térmica (wind chill)
weather_df['wind_chill'] = 13.12 + 0.6215 * weather_df['temp'] - 11.37 * (weather_df['w
```

Estas features personalizadas incorporam conhecimento de domínio sobre meteorologia: - `temp_humidity_ratio`: Captura a relação entre temperatura e umidade - `heat_index`: Aproximação de como a temperatura é percebida considerando a umidade -

`wind_chill`
: Aproximação de como a temperatura é percebida considerando o vento

Técnica 5: Seleção de Features

Após criar muitas features, é importante selecionar as mais relevantes para evitar overfitting e reduzir a dimensionalidade:

```
# Preparar dados para seleção de features
# Primeiro, precisamos converter todas as colunas para numéricas
numeric_columns = weather_df.select_dtypes(include=['number']).columns
X_for_selection = weather_df[numeric_columns].copy()

# Normalizar os dados para que a variância seja comparável
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_for_selection)
X_scaled_df = pd.DataFrame(X_scaled, columns=X_for_selection.columns)

# Aplicar seleção por limiar de variância
selector = VarianceThreshold(threshold=0.01)  # Remover features com variância < 0.01
X_selected = selector.fit_transform(X_scaled_df)

# Obter os nomes das features selecionadas
selected_features = X_for_selection.columns[selector.get_support()]
print(f"Features originais: {len(X_for_selection.columns)}")
print(f"Features selecionadas: {len(selected_features)}")
print("\nFeatures selecionadas:")
print(selected_features.tolist())
```

> ***Nota**: Existem muitas outras técnicas de seleção de features, como SelectKBest, RFE (Recursive Feature Elimination) e métodos baseados em modelos como feature importance de Random Forests.*

Automatizando a Engenharia de Features com Apache Airflow

Até agora, exploramos técnicas de engenharia de features em um ambiente interativo. No entanto, em um contexto de MLOps, precisamos automatizar esse processo para que seja executado regularmente, de forma confiável e escalável. É aqui que o Apache Airflow entra em cena.

O que é Apache Airflow?

Apache Airflow é uma plataforma de código aberto para criar, agendar e monitorar fluxos de trabalho programaticamente. Com o Airflow, você define seus pipelines como código, o que facilita o versionamento, teste e manutenção.

Os principais componentes do Airflow são:

- **DAG (Directed Acyclic Graph)**: Representa um fluxo de trabalho como um grafo acíclico direcionado de tarefas.
- **Operators**: Definem o que realmente é executado em cada tarefa (ex: PythonOperator, BashOperator).
- **Tasks**: Instâncias parametrizadas de operadores.
- **Task Dependencies**: Definem a ordem de execução das tarefas.

Projetando um Pipeline ETL para Engenharia de Features

Vamos projetar um pipeline ETL (Extract, Transform, Load) para automatizar nosso processo de engenharia de features:

1. **Extract**: Obter dados climáticos da API OpenWeatherMap.
2. **Transform**: Aplicar técnicas de engenharia de features aos dados.
3. **Load**: Armazenar os resultados em um banco de dados MongoDB.

Figura 7-2. Arquitetura do pipeline ETL com Apache Airflow

Implementando o DAG do Airflow

Aqui está a implementação do nosso pipeline como um DAG do Airflow:

```
import json
import pandas as pd
from airflow import DAG
from airflow.operators.python import PythonOperator
from airflow.providers.http.operators.http import SimpleHttpOperator
from airflow.models import Variable
from datetime import datetime, timedelta
from sklearn.preprocessing import PolynomialFeatures, OneHotEncoder
from sklearn.feature_selection import VarianceThreshold
import numpy as np
from pymongo import MongoClient

# Define functions outside the DAG block

def parse_weather_response(**context):
```

```
    """Parse the HTTP response into a DataFrame."""
    response = context["task_instance"].xcom_pull(task_ids="fetch_weather")
    data = json.loads(response)
    weather_data = {
        "temp": data["main"]["temp"],
        "humidity": data["main"]["humidity"],
        "wind_speed": data["wind"]["speed"],
        "weather_condition": data["weather"][0]["main"]
    }
    df = pd.DataFrame([weather_data])
    context["task_instance"].xcom_push(key="weather_df", value=df.to_json())

def apply_polynomial_features(**context):
    """Apply polynomial features to numerical columns."""
    df_json = context["task_instance"].xcom_pull(task_ids="parse_weather", key="weather
    df = pd.read_json(df_json)
    poly_degree = context["dag_run"].conf.get("poly_degree", 2)
    numerical_cols = ["temp", "humidity", "wind_speed"]
    poly = PolynomialFeatures(degree=poly_degree, include_bias=False)
    poly_features = poly.fit_transform(df[numerical_cols])
    feature_names = poly.get_feature_names_out(numerical_cols)
    df_poly = pd.DataFrame(poly_features, columns=feature_names)
    context["task_instance"].xcom_push(key="poly_features", value=df_poly.to_json())

def apply_one_hot_encoding(**context):
    """Apply one-hot encoding to categorical columns."""
    df_json = context["task_instance"].xcom_pull(task_ids="parse_weather", key="weather
    df = pd.read_json(df_json)
    categorical_cols = ["weather_condition"]
    encoder = OneHotEncoder(sparse=False, drop="first")
    encoded_features = encoder.fit_transform(df[categorical_cols])
    feature_names = encoder.get_feature_names_out(categorical_cols)
    df_encoded = pd.DataFrame(encoded_features, columns=feature_names)
    context["task_instance"].xcom_push(key="encoded_features", value=df_encoded.to_json

def combine_features(**context):
    """Combine polynomial and encoded features into a single DataFrame."""
    poly_json = context["task_instance"].xcom_pull(task_ids="apply_polynomial", key="po
    encoded_json = context["task_instance"].xcom_pull(task_ids="apply_encoding", key="e
    df_poly = pd.read_json(poly_json)
    df_encoded = pd.read_json(encoded_json)
    df_combined = pd.concat([df_poly, df_encoded], axis=1)
    context["task_instance"].xcom_push(key="combined_features", value=df_combined.to_js

def feature_selection(**context):
```

```
    """Perform variance-based feature selection."""
    df_json = context["task_instance"].xcom_pull(task_ids="combine_features", key="comb
    df = pd.read_json(df_json)
    selector = VarianceThreshold(threshold=0.01)  # Remove features with variance < 0.0
    selected_features = selector.fit_transform(df)
    feature_names = df.columns[selector.get_support()]
    df_selected = pd.DataFrame(selected_features, columns=feature_names)
    context["task_instance"].xcom_push(key="selected_features", value=df_selected.to_js

def load_to_mongodb(**context):
    """Load the transformed data into MongoDB."""
    df_json = context["task_instance"].xcom_pull(task_ids="feature_selection", key="sel
    df = pd.read_json(df_json)
    client = MongoClient("mongodb://localhost:27017/")  # Update if using MongoDB Atlas
    db = client["weather_db"]
    collection = db["daily_weather"]
    collection.insert_many(df.to_dict("records"))

# Define the DAG and operators inside the "with" block
with DAG(
    "daily_weather_etl",
    default_args={
        "owner": "airflow",
        "start_date": datetime(2023, 10, 1),
        "retries": 1,
        "retry_delay": timedelta(minutes=5),
    },
    schedule_interval="@daily",
    catchup=False,
    params={"city": "New York", "poly_degree": 2}  # Default parameters
) as dag:

    # Task 1: Fetch weather data from OpenWeatherMap API
    fetch_weather = SimpleHttpOperator(
        task_id="fetch_weather",
        http_conn_id="openweather_conn",  # Connection set in Airflow UI
        endpoint="data/2.5/weather?q={{ dag_run.conf['city'] }}&appid={{ var.value.weat
        method="GET",
        xcom_push=True,  # Push response to XCom
    )

    # Task 2: Parse the weather response
    parse_task = PythonOperator(
        task_id="parse_weather",
        python_callable=parse_weather_response,
```

```
        provide_context=True,
    )

    # Task 3: Apply polynomial features
    poly_task = PythonOperator(
        task_id="apply_polynomial",
        python_callable=apply_polynomial_features,
        provide_context=True,
    )

    # Task 4: Apply one-hot encoding
    encode_task = PythonOperator(
        task_id="apply_encoding",
        python_callable=apply_one_hot_encoding,
        provide_context=True,
    )

    # Task 5: Combine features
    combine_task = PythonOperator(
        task_id="combine_features",
        python_callable=combine_features,
        provide_context=True,
    )

    # Task 6: Perform feature selection
    selection_task = PythonOperator(
        task_id="feature_selection",
        python_callable=feature_selection,
        provide_context=True,
    )

    # Task 7: Load data into MongoDB
    load_task = PythonOperator(
        task_id="load_to_db",
        python_callable=load_to_mongodb,
        provide_context=True,
    )

    # Set task dependencies
    fetch_weather >> parse_task
    parse_task >> [poly_task, encode_task]  # Parallel execution
    [poly_task, encode_task] >> combine_task
    combine_task >> selection_task >> load_task
```

Explicação do Pipeline ETL

Vamos analisar cada componente do nosso pipeline:

1. Extração (Extract)

- Utilizamos o `SimpleHttpOperator` para fazer uma requisição à API OpenWeatherMap.
- Os parâmetros da requisição (cidade, chave de API) são configurados no Airflow.
- A resposta da API é armazenada usando o sistema XCom do Airflow para compartilhamento entre tarefas.

2. Transformação (Transform)

- **Parsing**: Convertemos a resposta JSON em um DataFrame pandas.
- **Feature Engineering**:
- Aplicamos polynomial features às variáveis numéricas.
- Aplicamos one-hot encoding às variáveis categóricas.
- Combinamos as features transformadas.
- Realizamos seleção de features baseada em variância.

3. Carregamento (Load)

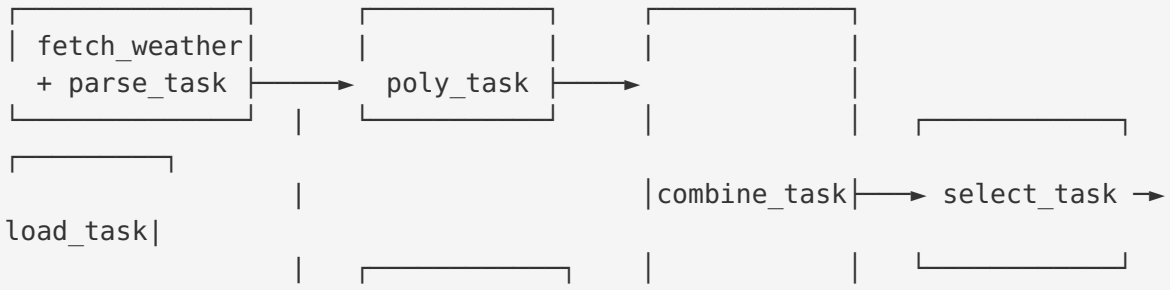
- Armazenamos as features processadas em um banco de dados MongoDB.

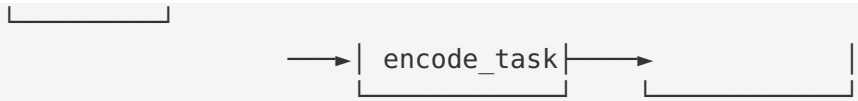
Fluxo de Execução do DAG

O fluxo de execução do DAG é definido pelas dependências entre as tarefas:

```
fetch_weather >> parse_task
parse_task >> [poly_task, encode_task]  # Execução paralela
[poly_task, encode_task] >> combine_task
combine_task >> selection_task >> load_task
```

Este fluxo pode ser visualizado como:





Configuração do Airflow

Para executar este DAG, você precisa configurar o Airflow:

1. **Instalar o Airflow**:
   ```
   bash
      pip install apache-airflow
      pip install apache-airflow-providers-http
      pip install pymongo
   ```
2. **Inicializar o banco de dados do Airflow**:
   ```
   bash
      airflow db init
   ```
3. **Criar um usuário administrador**:
   ```
   bash
      airflow users create \
          --username admin \
          --firstname Admin \
          --lastname User \
          --role Admin \
          --email admin@example.com \
          --password admin
   ```
4. **Configurar a conexão com a API OpenWeatherMap**:
5. Na interface do Airflow, vá para Admin > Connections
6. Adicione uma nova conexão:
   - Conn Id: openweather_conn
   - Conn Type: HTTP
   - Host: api.openweathermap.org
   - Schema: https
7. **Configurar a variável para a chave de API**:
8. Na interface do Airflow, vá para Admin > Variables
9. Adicione uma nova variável:
   - Key: weather_api_key
   - Value: sua_chave_api_openweathermap
10. **Iniciar o servidor web e o scheduler do Airflow**:

```
bash
   airflow webserver --port 8080
   airflow scheduler
```

Vantagens do Apache Airflow para Engenharia de Features

O uso do Apache Airflow para automatizar pipelines de engenharia de features traz várias vantagens:

1. **Automação**: Executa o pipeline automaticamente conforme o agendamento definido.
2. **Monitoramento**: Fornece uma interface visual para monitorar a execução do pipeline.
3. **Escalabilidade**: Pode lidar com grandes volumes de dados e pipelines complexos.
4. **Flexibilidade**: Permite definir fluxos de trabalho complexos com execução paralela e dependências.
5. **Recuperação de Falhas**: Oferece mecanismos para lidar com falhas e retentativas.
6. **Versionamento**: O pipeline é definido como código, facilitando o versionamento e a colaboração.

Desafio Avançado: Integração com Feast Feature Store

Para equipes que buscam escalar suas operações de MLOps, um feature store pode ser uma adição valiosa à infraestrutura. Nesta seção, exploraremos como integrar nosso pipeline com Feast, um feature store de código aberto.

> ***Nota**: Esta seção é opcional e destinada a usuários avançados que desejam explorar ferramentas adicionais de MLOps.*

O que é um Feature Store?

Um feature store é um sistema especializado para armazenar, gerenciar e servir features para modelos de machine learning. Ele resolve problemas comuns em pipelines de ML, como:

1. **Consistência entre treinamento e inferência**: Garante que as mesmas transformações sejam aplicadas aos dados de treinamento e produção.
2. **Reutilização de features**: Permite que features sejam compartilhadas entre diferentes modelos e equipes.
3. **Monitoramento**: Facilita o monitoramento de drift e qualidade das features.
4. **Governança**: Fornece linhagem e documentação para features.

Introdução ao Feast

Feast (Feature Store) é uma ferramenta de código aberto para gerenciar, servir e descobrir features para machine learning. Ele fornece:

- Um registro centralizado de definições de features
- Armazenamento online e offline para features
- APIs para recuperação de features para treinamento e inferência
- Integração com várias fontes de dados e plataformas de ML

Integrando Feast ao Pipeline Airflow

Aqui está um exemplo de como integrar o Feast ao nosso pipeline:

```
# Primeiro, definimos as entidades e features no Feast
# Arquivo: feature_repo/feature_definitions.py

from datetime import timedelta
from feast import Entity, Feature, FeatureView, ValueType
from feast.data_source import FileSource

# Define a entidade para nossos dados climáticos
location = Entity(name="location", value_type=ValueType.STRING, description="Localizaçã

# Define a fonte de dados para as features climáticas
weather_source = FileSource(
    path="/path/to/weather_features.parquet",  # Caminho para o arquivo de features
    event_timestamp_column="date",
)

# Define a view de features para os dados climáticos
weather_features_view = FeatureView(
    name="weather_features",
    entities=[location],
    ttl=timedelta(days=3),  # Time-to-live para as features
    features=[
        Feature(name="temp", dtype=ValueType.FLOAT),
        Feature(name="humidity", dtype=ValueType.FLOAT),
        Feature(name="wind_speed", dtype=ValueType.FLOAT),
        Feature(name="temp_squared", dtype=ValueType.FLOAT),
        Feature(name="humidity_squared", dtype=ValueType.FLOAT),
        Feature(name="temp_humidity", dtype=ValueType.FLOAT),
        Feature(name="is_rainy", dtype=ValueType.INT32),
        Feature(name="is_sunny", dtype=ValueType.INT32),
```

```
    ],
    online=True,
    input=weather_source,
    tags={"team": "weather_analytics"},
)
```

Agora, modificamos a função `load_to_mongodb` no nosso DAG para incluir a integração com Feast:

```
def load_to_feast(**context):
    """Load the transformed data into Feast feature store."""
    from feast import Client
    import pandas as pd
    from datetime import datetime

    # Obter as features processadas
    df_json = context["task_instance"].xcom_pull(task_ids="feature_selection", key="sel
    df = pd.read_json(df_json)

    # Adicionar colunas necessárias para o Feast
    df["location"] = context["dag_run"].conf.get("city", "New York")  # Entidade
    df["date"] = datetime.now()  # Timestamp do evento

    # Salvar o DataFrame em um formato que o Feast possa ler
    parquet_path = "/path/to/weather_features.parquet"
    df.to_parquet(parquet_path)

    # Conectar ao Feast e aplicar as definições de features
    feast_client = Client(repo_path="/path/to/feature_repo")
    feast_client.apply()  # Aplica as definições de features

    # Ingerir os dados no feature store
    feast_client.materialize_incremental(datetime.now() - timedelta(days=1), datetime.n

    return "Features loaded to Feast successfully"

# Adicionar a tarefa ao DAG
load_feast_task = PythonOperator(
    task_id="load_to_feast",
    python_callable=load_to_feast,
    provide_context=True,
)
```

```
# Modificar as dependências
selection_task >> load_feast_task
```

Benefícios da Integração com Feast

Integrar nosso pipeline de engenharia de features com o Feast traz vários benefícios:

1. **Consistência**: Garante que as mesmas transformações sejam aplicadas aos dados de treinamento e produção.
2. **Reutilização**: Permite que as features sejam facilmente reutilizadas em diferentes modelos.
3. **Escalabilidade**: Facilita o gerenciamento de features em projetos de grande escala.
4. **Monitoramento**: Fornece ferramentas para monitorar a qualidade e o drift das features.
5. **Documentação**: Mantém um registro centralizado de metadados sobre as features.

Resumo

Neste capítulo, exploramos o processo completo de engenharia de features, desde a compreensão dos conceitos básicos até a implementação de um pipeline ETL automatizado com Apache Airflow.

Conceitos-chave abordados:

1. **Fundamentos de Engenharia de Features**:
2. Importância da engenharia de features para modelos de machine learning
3. Processo de transformação de dados brutos em features úteis
4. **Técnicas de Engenharia de Features**:
5. Extração de features temporais
6. Features polinomiais
7. One-hot encoding
8. Features de interação personalizadas
9. Seleção de features
10. **Automatização com Apache Airflow**:
11. Estrutura de um DAG no Airflow
12. Implementação de um pipeline ETL completo
13. Configuração e execução do Airflow

14. **Integração com Feast Feature Store (opcional)**:

15. Conceito de feature store
16. Benefícios da utilização de um feature store
17. Implementação básica com Feast

Próximos Passos

Para aprofundar seu conhecimento em engenharia de features e MLOps:

1. **Experimente com diferentes técnicas de feature engineering**:
2. Transformações não-lineares adicionais
3. Técnicas de redução de dimensionalidade (PCA, t-SNE)

4. Métodos de seleção de features baseados em modelos

5. **Explore recursos avançados do Apache Airflow**:

6. Sensores e gatilhos
7. Branching e condicionais

8. Paralelismo e pools

9. **Aprofunde-se em MLOps**:

10. Implementação completa com Feast
11. Monitoramento de modelos e features
12. Integração com plataformas de ML

A engenharia de features é uma habilidade fundamental para qualquer profissional de dados e MLOps. Ao dominar as técnicas apresentadas neste capítulo e automatizá-las com ferramentas como Apache Airflow, você estará bem equipado para construir pipelines de dados robustos e escaláveis para suas aplicações de machine learning.

Exercícios

1. **Básico**: Modifique o pipeline Airflow para incluir mais features temporais, como hora do dia e feriados.

2. **Intermediário**: Implemente uma técnica adicional de seleção de features, como SelectKBest ou RFE, e compare os resultados com o método baseado em variância.

3. **Avançado**: Estenda o pipeline para incluir detecção de drift de features, alertando quando as distribuições das features mudam significativamente.

4. **Desafio**: Implemente a integração completa com Feast conforme descrito na seção opcional e use-a para servir features para um modelo de previsão de temperatura.

Referências

- Zheng, A., & Casari, A. (2018). *Feature Engineering for Machine Learning: Principles and Techniques for Data Scientists*. O'Reilly Media.
- Kleppmann, M. (2017). *Designing Data-Intensive Applications*. O'Reilly Media.
- Apache Airflow Documentation: https://airflow.apache.org/docs/
- Feast Documentation: https://docs.feast.dev/
- scikit-learn Documentation: https://scikit-learn.org/stable/modules/feature_selection.html