Tuning de hiperparâmetros

A técnica de tuning de hiperparâmetros em machine learning refere-se ao processo de encontrar a melhor combinação de hiperparâmetros para um modelo, a fim de otimizar seu desempenho. Hiperparâmetros são os parâmetros que não são aprendidos diretamente pelo modelo durante o treinamento, mas precisam ser definidos antes, como a taxa de aprendizado, a profundidade de uma árvore de decisão ou o número de estimadores em um Random Forest. O objetivo do tuning é ajustar esses hiperparâmetros para melhorar a precisão, reduzir o overfitting ou underfitting e, assim, obter o melhor modelo possível para os dados.

Existem diversas abordagens para o tuning de hiperparâmetros, sendo as mais comuns:

- Busca em Grade (Grid Search): Testa todas as combinações possíveis de hiperparâmetros em uma grade predefinida.
- Busca Aleatória (Random Search): Testa combinações de hiperparâmetros aleatórias dentro de intervalos definidos.
- Optimizadores Bayesianos: Usam métodos probabilísticos para encontrar a combinação ideal de hiperparâmetros de forma mais eficiente.

O tuning é uma etapa importante para melhorar o desempenho dos modelos e é geralmente feito após a seleção de características e o treinamento inicial.

```
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, f1_score
from sklearn.feature_selection import SelectKBest, f_classif
```

No começo do código, estamos importando bibliotecas essenciais para a nossa análise. A função `train_test_split` da biblioteca `sklearn.model_selection` vai ser usada para dividir o nosso dataset em dois conjuntos: um para treinamento do modelo e outro para testar o desempenho do modelo após o treinamento. Já o `GridSearchCV`, também de `sklearn.model_selection`, é uma ferramenta importante que vamos usar para realizar a busca em grade (grid search), que é a técnica de tuning de hiperparâmetros. Ela vai nos ajudar a encontrar a melhor combinação de parâmetros para otimizar o desempenho do modelo.

Também estamos importando o `DecisionTreeClassifier` de `sklearn.tree`. Esse modelo de árvore de decisão é um dos algoritmos que vamos usar para fazer as classificações, no caso, para prever fraudes ou não fraudes. Em seguida, temos a importação das funções de avaliação de desempenho: `accuracy_score`, `confusion_matrix` e `f1_score`, todas da biblioteca `sklearn.metrics`. O `accuracy_score` vai nos dar a taxa de acerto do modelo, o `confusion_matrix` vai mostrar em detalhes como o modelo se saiu em termos de classificações corretas e incorretas, e o `f1_score` é uma métrica que nos ajuda a avaliar a precisão do modelo, especialmente em datasets desbalanceados, como o nosso caso de fraudes.

Por fim, importamos `SelectKBest` e `f_classif` de `sklearn.feature_selection`. O `SelectKBest` é uma técnica que vamos usar para selecionar os melhores atributos do nosso dataset, e o `f_classif` é o método estatístico que avalia a importância de cada atributo, utilizando o teste F. Isso vai nos ajudar a reduzir a dimensionalidade dos dados, escolhendo apenas os atributos mais relevantes para o modelo, o que também pode melhorar sua performance.

```python
import pandas as pd

# carregando o dataset
df = pd.read_csv('/content/creditcard.csv')

# informações básicas sobre o dataset
print(df.info())
print(df.describe())
```

No nosso código, começamos importando a biblioteca `pandas`, que é essencial para a manipulação e análise de dados. A usamos com o alias `pd`, que é uma convenção comum na comunidade de dados. A primeira ação que fazemos é carregar o dataset, utilizando o comando `pd.read_csv('/content/creditcard.csv')`. Esse comando lê o arquivo CSV contendo as transações de cartão de crédito e o armazena em um DataFrame chamado `df`. O DataFrame é uma estrutura de dados do pandas que nos permite trabalhar de forma eficiente com as informações carregadas.

Depois de carregar os dados, é importante realizar uma análise inicial para entender o formato e as características do dataset. Para isso, utilizamos dois métodos. O `df.info()` nos fornece informações gerais sobre o DataFrame, como o número de linhas, o tipo de dados em cada coluna e quantos valores não nulos existem em cada uma delas. Isso nos ajuda a identificar se há valores faltantes ou colunas que precisam de atenção. Já o `df.describe()` nos dá uma visão das estatísticas descritivas das colunas numéricas, como média, desvio padrão, mínimo, máximo e quartis. Esses dados estatísticos nos ajudam a ter uma ideia mais clara da distribuição dos valores e podem indicar a presença de outliers ou de comportamentos inesperados nos dados.

```python
# definindo as variáveis independentes (X) e a variável dependente (y)
X = df.drop(columns='Class')  # supondo que 'Class' seja a coluna de transação fraudulenta (1 para fraude, 0 para normal)
y = df['Class']

# dividindo o dataset em conjuntos de treino e teste
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=1)

# treinando e avaliando o modelo com todos os atributos
clf = DecisionTreeClassifier(random_state=1)
clf.fit(X_train, y_train)
y_pred_all = clf.predict(X_test)

print("Desempenho com todos os atributos:")
print("Acurácia:", accuracy_score(y_test, y_pred_all))
print("Matriz de Confusão:")
print(confusion_matrix(y_test, y_pred_all))
print("F1 Score:", f1_score(y_test, y_pred_all))
```

Neste trecho de código, estamos realizando o processo de treinamento e avaliação de um modelo de árvore de decisão, utilizando todos os atributos do dataset para prever se uma transação é fraudulenta ou não.

Primeiro, definimos as variáveis independentes (X) e dependente (y). A variável `X` contém todas as colunas do dataset, exceto a coluna 'Class', que é a variável dependente, pois representa se uma transação foi fraudulenta ou não. A variável `y` é a coluna 'Class', onde '1' indica fraude e '0' indica uma transação normal.

Depois, dividimos o dataset em conjuntos de treino e teste usando a função `train_test_split`. Com isso, 80% dos dados serão usados para treinar o modelo e 20% para testar sua performance. O parâmetro `random_state=1` garante que a divisão seja reprodutível, ou seja, os mesmos dados serão divididos da mesma forma cada vez que o código for executado.

Em seguida, treinamos o modelo com todos os atributos. Usamos o `DecisionTreeClassifier` para criar o modelo de árvore de decisão e, com o método `.fit()`, treinamos o modelo nos dados de treino (`X_train` e `y_train`). Após o treinamento, usamos o método `.predict()` para fazer previsões sobre o conjunto de teste (`X_test`), armazenando os resultados em `y_pred_all`.

Por fim, avaliamos o desempenho do modelo. Calculamos a acurácia com `accuracy_score`, que nos dá a porcentagem de previsões corretas. Também exibimos a matriz de confusão com `confusion_matrix`, que nos ajuda a entender como o modelo está classificando as transações (verdadeiros positivos, falsos positivos, etc.). Além disso, calculamos o F1 Score com `f1_score`, que é uma métrica importante para avaliar a performance do modelo, especialmente em casos de desbalanceamento das classes, como é comum em problemas de fraude.

```
# seleção de atributos com SelectKBest
selector = SelectKBest(f_classif, k=10)  # selecionando os 10 atributos mais relevantes
X_train_selected = selector.fit_transform(X_train, y_train)
X_test_selected = selector.transform(X_test)

# treinando e avaliando o modelo com os atributos selecionados antes do tuning
clf = DecisionTreeClassifier(random_state=1)
clf.fit(X_train_selected, y_train)
y_pred_selected = clf.predict(X_test_selected)

print("\nDesempenho com atributos selecionados (antes do tuning):")
print("Acurácia:", accuracy_score(y_test, y_pred_selected))
print("Matriz de Confusão:")
print(confusion_matrix(y_test, y_pred_selected))
print("F1 Score:", f1_score(y_test, y_pred_selected))
```

Neste trecho de código, estamos aplicando a seleção de atributos para melhorar o modelo de árvore de decisão, focando apenas nos atributos mais relevantes antes de realizar qualquer tuning de hiperparâmetros.

Primeiro, realizamos a seleção de atributos com a técnica SelectKBest. O SelectKBest é configurado com a função de avaliação f_classif, que usa o teste F para medir a relação entre cada atributo e a variável dependente (fraude ou não). O parâmetro k=10 indica que queremos selecionar os 10 atributos mais importantes com base nesse teste. Em seguida, usamos o método .fit_transform() nos dados de treino (X_train) para aplicar a seleção de atributos, e .transform() nos dados de teste (X_test) para garantir que a seleção de atributos seja aplicada de forma consistente entre treino e teste.

Após selecionar os 10 atributos mais relevantes, treinamos o modelo novamente. Usamos o mesmo modelo de árvore de decisão (DecisionTreeClassifier) e treinamos com os dados de treino filtrados (X_train_selected) e suas correspondentes saídas (y_train). Depois, fazemos previsões sobre o conjunto de teste com predict(), armazenando os resultados em y_pred_selected.

Por fim, avaliamos o desempenho do modelo com os atributos selecionados. Calculamos novamente a acurácia com accuracy_score, a matriz de confusão com confusion_matrix e o F1 Score com f1_score. Essas métricas nos ajudam a entender como o modelo se comportou após a seleção dos atributos mais importantes, comparado ao modelo que usou todos os atributos inicialmente. A ideia aqui é verificar se a redução da dimensionalidade do dataset com a seleção de atributos melhora ou não o desempenho do modelo antes de fazer o tuning de hiperparâmetros.

```python
# tuning de hiperparâmetros com GridSearchCV
# definindo o espaço de hiperparâmetros para a árvore de decisão
param_grid = {
    'max_depth': [None, 10, 20, 30, 40, 50],
    'min_samples_split': [2, 5, 10, 15],
    'min_samples_leaf': [1, 2, 5, 10]
}

# usando GridSearchCV para encontrar os melhores hiperparâmetros
grid_search = GridSearchCV(
    estimator=DecisionTreeClassifier(random_state=1),
    param_grid=param_grid,
    scoring='accuracy',
    cv=5,
    n_jobs=-1
)

# realizando a busca com os dados de treino (usando atributos selecionados)
grid_search.fit(X_train_selected, y_train)

# exibindo os melhores hiperparâmetros encontrados
print("\nMelhores hiperparâmetros:", grid_search.best_params_)

# treinando o modelo com os melhores hiperparâmetros
best_clf = grid_search.best_estimator_
best_clf.fit(X_train_selected, y_train)
y_pred_tuned = best_clf.predict(X_test_selected)

# avaliando o modelo ajustado
print("\nDesempenho com atributos selecionados e hiperparâmetros otimizados:")
print("Acurácia:", accuracy_score(y_test, y_pred_tuned))
print("Matriz de Confusão:")
print(confusion_matrix(y_test, y_pred_tuned))
print("F1 Score:", f1_score(y_test, y_pred_selected))
```

Neste trecho de código, estamos aplicando o tuning de hiperparâmetros para otimizar o desempenho do modelo de árvore de decisão. Vamos detalhar cada parte.

Primeiro, definimos o espaço de hiperparâmetros que será explorado durante a busca. A variável param_grid é um dicionário onde as chaves são os parâmetros do modelo de árvore de decisão que queremos ajustar, e os valores são as possíveis opções para esses parâmetros. Os parâmetros que estamos ajustando são:

- max_depth: controla a profundidade máxima da árvore (quanto maior, mais complexa é a árvore). Vamos testar diferentes valores, incluindo None, que significa que a árvore cresce até o final.
- min_samples_split: determina o número mínimo de amostras necessárias para dividir um nó. Testamos valores de 2 a 15.
- min_samples_leaf: define o número mínimo de amostras em uma folha. Também testamos valores entre 1 e 10.

Com o espaço de hiperparâmetros definido, usamos o GridSearchCV para buscar os melhores parâmetros. O GridSearchCV é uma técnica de validação cruzada que testa todas as combinações possíveis dos parâmetros definidos em param_grid. O parâmetro scoring='accuracy' indica que queremos otimizar a acurácia do modelo. O cv=5 significa que usaremos validação cruzada com 5 dobras, o que ajuda a evitar overfitting e fornece uma avaliação mais robusta do desempenho. O n_jobs=-1 permite que o GridSearch utilize todos os núcleos do processador para acelerar o processo de busca.

Após realizar a busca com os dados de treino (X_train_selected e y_train), exibimos os melhores hiperparâmetros encontrados com grid_search.best_params_. Esses são os parâmetros que o modelo considera mais adequados para maximizar a acurácia com base nos dados de treino.

Com os melhores hiperparâmetros identificados, treinamos o modelo novamente. Usamos o modelo otimizado (grid_search.best_estimator_) para treinar nos dados de treino e, em seguida, fazemos previsões sobre o conjunto de teste com predict(), armazenando os resultados em y_pred_tuned.

Finalmente, avaliamos o desempenho do modelo ajustado. Calculamos a acurácia, a matriz de confusão e o F1 Score, assim como fizemos nas etapas anteriores, para verificar como o modelo melhorado (com atributos selecionados e hiperparâmetros otimizados) está se comportando.

O objetivo aqui é comparar o desempenho do modelo após o tuning de hiperparâmetros com o modelo anterior, onde os hiperparâmetros não foram ajustados. Isso nos permite verificar se o ajuste dos parâmetros realmente trouxe melhorias no modelo.

Conclusão

O código realizado focou em melhorar o desempenho do modelo de árvore de decisão por meio de seleção de atributos e tuning de hiperparâmetros.

Primeiro, utilizamos o SelectKBest para selecionar os 10 atributos mais relevantes, eliminando variáveis desnecessárias e aumentando a eficiência do modelo. Após, aplicamos o GridSearchCV para otimizar os hiperparâmetros da árvore de decisão, como max_depth, min_samples_split e min_samples_leaf, buscando a combinação que resultasse no melhor desempenho.

A comparação das métricas (acurácia, matriz de confusão e F1 Score) entre o modelo com atributos selecionados e o modelo com hiperparâmetros otimizados ajudou a verificar se a otimização trouxe melhorias, visando uma maior precisão na classificação de fraudes.

A análise final deve mostrar que o tuning de hiperparâmetros aprimorou a performance do modelo.

Antes:

```
Desempenho com atributos selecionados (antes do tuning):
Acurácia: 0.9318181818181818
Matriz de Confusão:
[[306   8]
 [ 22 104]]
F1 Score: 0.8739495798319328
```

Depois:

```
Desempenho com atributos selecionados e hiperparâmetros otimizados:
Acurácia: 0.9386363636363636
Matriz de Confusão:
[[308   6]
 [ 21 105]]
F1 Score: 0.8739495798319328
```

https://colab.research.google.com/drive/1b1aVyP9AVpSQNF3Am9NdtGeaaDq1_Pkp#scrollTo=cGEcBCjxdObK