Inteligência Artificial

812839 - Vinícius Miranda de Araújo

Lista 04

Questão 1

Utilizando o **BayesSearchCV** e ajustando os hiperparâmetros do **Random Forest** e **Árvore de decisão** para o problema do TITANIC.

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.preprocessing import LabelEncoder
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
from yellowbrick.classifier import ConfusionMatrix
from sklearn import tree
from skopt import BayesSearchCV

# Ler o arquivo de treino
training_data = pd.read_csv( 'titanic/train.csv' )
# Ler o arquivo de teste
test_data = pd.read_csv( 'titanic/test.csv' )
# Ler o arquivo com a resposta correta do o conjunto de teste
truth_table = pd.read_csv( 'titanic/gender_submission.csv' )

# Adicionar coluna 'Survived' ao test_data
test_data = test_data.merge(truth_table, on='PassengerId', how='left')

# -------------------------------------
# --- Pre-processamentos de Dados
# -------------------------------------

# Remover colunas irrelevantes ou com muitos valores ausentes
columns_to_drop = ['PassengerId', 'Name', 'Ticket', 'Cabin', 'Embarked']

# Transformação de dados categóricos
encoder = LabelEncoder()
training_data['Sex'] = encoder.fit_transform( training_data['Sex'] )
test_data['Sex'] = encoder.transform( test_data['Sex'] )

# Preenchendo valores ausentes
training_data['Age'] = training_data['Age'].fillna( training_data['Age'].median( ) )
test_data['Age'] = test_data['Age'].fillna( test_data['Age'].median( ) )

training_data['Fare'] = training_data['Fare'].fillna( training_data['Fare'].median( ) )
test_data['Fare'] = test_data['Fare'].fillna( test_data['Fare'].median( ) )

# Separar variáveis independentes e dependentes
X_treino = training_data.drop( columns = columns_to_drop + ['Survived'], axis = 1 ) # X_treino = colunas de treino
y_treino = training_data['Survived']                                                 # y_treino = coluna de resposta

X_teste = test_data.drop( columns = columns_to_drop + ['Survived'], axis = 1 )  # X_teste = colunas de teste
y_teste = test_data['Survived']                                                 # y_teste = coluna de resposta

# -------------------------------------
# --- Descobrir melhores hiperparâmetros
# -------------------------------------

# Hiperparâmetros Decision Tree
dt_params = {
    'criterion': ['gini', 'entropy'],
    'max_depth': [None, 2, 4, 6, 8, 10],
    'max_features': [None, 'sqrt', 'log2', 0.2, 0.4, 0.6, 0.8],
    'min_samples_split': [20, 30, 40, 50]
}

# Hiperparâmetros Random Forest
rf_params = {
    'n_estimators': (50, 200),
    'max_depth': (2, 10),
    'max_features': ['sqrt', 'log2'],
    'min_samples_split': (2, 10)
```

```
}

# Encontrar melhores hiperparâmetros
dt_modelo = BayesSearchCV(
    DecisionTreeClassifier( ),
    search_spaces = dt_params,
    cv = 5,
    n_jobs  = -1,
    verbose = 1
)
dt_modelo.fit( X_treino, y_treino )
print( "Melhores hiperparâmetros   - Decision Tree:", dt_modelo.best_params_ )

# Otimização Random Forest
rf_modelo = BayesSearchCV(
    RandomForestClassifier( ),
    search_spaces = rf_params,
    cv = 5,
    n_jobs = -1,
    verbose = 1
)
rf_modelo.fit( X_treino, y_treino )
print( "Melhores hiperparâmetros   - Random Forest:", rf_modelo.best_params_ )

# -------------------------------------
# --- Treinar o Modelo
# -------------------------------------

# Treinar modelo final com os melhores hiperparâmetros
dt_modelo_final = DecisionTreeClassifier( **dt_modelo.best_params_ )
dt_modelo_final.fit( X_treino, y_treino )

rf_modelo_final = RandomForestClassifier( **rf_modelo.best_params_ )
rf_modelo_final.fit( X_treino, y_treino )

# -------------------------------------
# --- Testar e Avaliar o Modelo
# -------------------------------------

# Fazer previsões
dt_pred = dt_modelo_final.predict( X_teste )
rf_pred = rf_modelo_final.predict( X_teste )

# Avaliar o modelo
print( "Acurácia do modelo         - DecisionTree:"  , accuracy_score( y_teste, dt_pred ) )
print( "Matriz de Confusão         - DecisionTree:\n", confusion_matrix( y_teste, dt_pred ) )
print( "Relatório de Classificação - DecisionTree:\n", classification_report( y_teste, dt_pred ) )

print( "Acurácia do modelo         - RandomForest:"  , accuracy_score( y_teste, rf_pred ) )
print( "Matriz de Confusão         - RandomForest:\n", confusion_matrix( y_teste, rf_pred ) )
print( "Relatório de Classificação - RandomForest:\n", classification_report( y_teste, rf_pred ) )

# Importância das features
def plot_feature_importance( modelo, X_treino ):
    importancias = modelo.feature_importances_
    features     = pd.DataFrame( {'Feature': X_treino.columns, 'Importância': importancias} )
    features     = features.sort_values( by='Importância', ascending=False )
    print( features )
# plot_feature_importance( )

print( "Importância dos Atributos - Decision Tree" )
plot_feature_importance( dt_modelo_final, X_treino )

print( "\nImportância dos Atributos - Random Forest" )
plot_feature_importance( rf_modelo_final, X_treino )
```

Avaliação de Desempenho

Decision Tree:

- Acurácia:

> 0.8588 = 85.88%

- Classificação:

> | | precision | recall | f1-score |
> |---|---|---|---|
> | **0** | 0.88 | 0.90 | 0.89 |

	precision	recall	f1-score
1	0.82	0.78	0.80

Random Forest:

- Acurácia:

> 0.8851 = 88.51%

- Classificação:

	precision	recall	f1-score
0	0.89	0.93	0.91
1	0.87	0.80	0.84

Conclusão

O **Random Forest** obteve melhor desempenho, com **acurácia de 88.5%** contra **85.9% da Decision Tree**.

O **Random Forest** teve melhor recall para a classe 0 (não sobreviveu) e maior equilíbrio entre precisão e recall para a classe 1 (sobreviveu).

Avaliação de Atributos

Os **quatro atributos principais são os mesmos** em ambos os modelos, mas com **importâncias ligeiramente diferentes**:

Feature	Decision Tree	Random Forest
Sex	39.0%	40.0%
Fare	26.9%	21.8%
Age	18.2%	17.5%
Pclass	12.8%	12.1%
SibSp	1.9%	4.8%
Parch	0.9%	3.6%

Conclusão

- **Sexo** é a feature mais relevante nos dois modelos.
- **Fare (Tarifa paga pelo passageiro)** tem mais peso na Decision Tree do que no Random Forest.
- **SibSp e Parch** têm pouca importância, mas no Random Forest sua relevância é maior.

Questão 2

Utilizando o **SMOTE** para balanceamento dos dados:

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.preprocessing import LabelEncoder
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
from yellowbrick.classifier import ConfusionMatrix
from imblearn.over_sampling import SMOTE
from skopt import BayesSearchCV
from sklearn import tree

# Ler o arquivo de treino
training_data = pd.read_csv( 'titanic/train.csv' )
# Ler o arquivo de teste
test_data = pd.read_csv( 'titanic/test.csv' )
# Ler o arquivo com a resposta correta do o conjunto de teste
truth_table = pd.read_csv( 'titanic/gender_submission.csv' )

# Adicionar coluna 'Survived' ao test_data
test_data = test_data.merge(truth_table, on='PassengerId', how='left')

# -------------------------------------
# --- Pre-processamentos de Dados
# -------------------------------------

# Remover colunas irrelevantes ou com muitos valores ausentes
columns_to_drop = ['PassengerId', 'Name', 'Ticket', 'Cabin', 'Embarked']

# Transformação de dados categóricos
encoder = LabelEncoder()
```

```
training_data['Sex'] = encoder.fit_transform( training_data['Sex'] )
test_data['Sex'] = encoder.transform( test_data['Sex'] )

# Preenchendo valores ausentes
training_data['Age'] = training_data['Age'].fillna( training_data['Age'].median( ) )
test_data['Age'] = test_data['Age'].fillna( test_data['Age'].median( ) )

training_data['Fare'] = training_data['Fare'].fillna( training_data['Fare'].median( ) )
test_data['Fare'] = test_data['Fare'].fillna( test_data['Fare'].median( ) )

# Separar variáveis independentes e dependentes
X_treino = training_data.drop( columns = columns_to_drop + ['Survived'], axis = 1 ) # X_treino = colunas de treino
y_treino = training_data['Survived']                                                 # y_treino = coluna de resposta

X_teste = test_data.drop( columns = columns_to_drop + ['Survived'], axis = 1 ) # X_teste = colunas de teste
y_teste = test_data['Survived']                                                # y_teste = coluna de resposta

# Balenceamento dos dados com SMOTE
smote = SMOTE( random_state = 42 )
X_resampled, y_resampled = smote.fit_resample( X_treino, y_treino )

print( "Antes do balanceamento  - X:\n", X_treino.value_counts( ) )
print( "Depois do balanceamento - X:\n", X_resampled.value_counts( ) )

print( "Antes do balanceamento  - y:\n", y_treino.value_counts( ) )
print( "Depois do balanceamento - y:\n", y_resampled.value_counts( ) )

# -------------------------------------
# --- Descobrir melhores hiperparâmetros
# -------------------------------------

# Hiperparâmetros Random Forest
rf_params = {
    'n_estimators': (50, 200),
    'max_depth': (2, 10),
    'max_features': ['sqrt', 'log2'],
    'min_samples_split': (2, 10)
}

# Otimização Random Forest
rf_modelo = BayesSearchCV(
    RandomForestClassifier( ),
    search_spaces = rf_params,
    cv = 5,
    n_jobs = -1,
    verbose = 1,
    random_state = 42
)
rf_modelo.fit( X_resampled, y_resampled )
print( "Melhores hiperparâmetros   - Random Forest:", rf_modelo.best_params_ )

# -------------------------------------
# --- Treinar o Modelo
# -------------------------------------

# Treinar modelo final com os melhores hiperparâmetros
rf_modelo_final = RandomForestClassifier( **rf_modelo.best_params_, random_state = 42 )
rf_modelo_final.fit( X_resampled, y_resampled )

# -------------------------------------
# --- Testar e Avaliar o Modelo
# -------------------------------------

# Fazer previsões
rf_pred = rf_modelo_final.predict( X_teste )

# Avaliar o modelo
print( "Acurácia do modelo         - RandomForest:"  , accuracy_score( y_teste, rf_pred ) )
print( "Matriz de Confusão         - RandomForest:\n", confusion_matrix( y_teste, rf_pred ) )
print( "Relatório de Classificação - RandomForest:\n", classification_report( y_teste, rf_pred ) )

# Importância das features
def plot_feature_importance( modelo, X_treino ):
    importancias = modelo.feature_importances_
    features     = pd.DataFrame( {'Feature': X_treino.columns, 'Importância': importancias} )
    features     = features.sort_values( by='Importância', ascending=False )
    print( features )
# plot_feature_importance( )

print( "\nImportância dos Atributos - Random Forest" )
```

```
plot_feature_importance( rf_modelo_final, X_resampled )
```

Avaliação

ANTES:

- Acurácia:

> 0.8851 = 88.51%

- Classificação:

> | | precision | recall | f1-score |
> |---|---|---|---|
> | **0** | 0.89 | 0.93 | 0.91 |
> | **1** | 0.87 | 0.80 | 0.84 |

DEPOIS:

- Acurácia:

> 0.8660 = 86.6%

- Classificação:

> | | precision | recall | f1-score |
> |---|---|---|---|
> | **0** | 0.89 | 0.91 | 0.90 |
> | **1** | 0.83 | 0.80 | 0.81 |

Conclusão

A principal mudança após o balanceamento com SMOTE foi:

- Leve queda na acurácia (de 0.89 para 0.87).
- F1-score da classe 1 (sobreviventes) caiu um pouco (0.84 → 0.81).
- Os pesos das variáveis mudaram levemente, mas Sex, Fare e Age continuam sendo os atributos mais importantes.

Questão 3

Antes, eu estava imputando os valores ausentes utilizando a **mediana** para preencher os campos vazios. Esse método é simples e eficaz para evitar a perda de dados, mas não leva em consideração as relações entre diferentes variáveis.

Agora, vou utilizar o **KNNImputer**, que utiliza a média dos k vizinhos mais próximos para preencher os valores ausentes de forma mais inteligente.

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.preprocessing import LabelEncoder
from sklearn.ensemble import RandomForestClassifier
from sklearn.impute import KNNImputer
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
from yellowbrick.classifier import ConfusionMatrix
from imblearn.over_sampling import SMOTE
from skopt import BayesSearchCV
from sklearn import tree

# Ler o arquivo de treino
training_data = pd.read_csv( 'titanic/train.csv' )
# Ler o arquivo de teste
test_data = pd.read_csv( 'titanic/test.csv' )
# Ler o arquivo com a resposta correta do o conjunto de teste
truth_table = pd.read_csv( 'titanic/gender_submission.csv' )

# Adicionar coluna 'Survived' ao test_data
test_data = test_data.merge(truth_table, on='PassengerId', how='left')

# ------------------------------------
# --- Pre-processamentos de Dados
# ------------------------------------

# Remover colunas irrelevantes ou com muitos valores ausentes
columns_to_drop = ['PassengerId', 'Name', 'Ticket', 'Cabin', 'Embarked']

# Transformação de dados categóricos
```

```
encoder = LabelEncoder()
training_data['Sex'] = encoder.fit_transform( training_data['Sex'] )
test_data['Sex'] = encoder.transform( test_data['Sex'] )

# Preenchendo valores ausentes com KNNImputer
# training_data['Age'] = training_data['Age'].fillna(training_data['Age'].median())
# test_data['Age'] = test_data['Age'].fillna(test_data['Age'].median())

# training_data['Fare'] = training_data['Fare'].fillna(training_data['Fare'].median())
# test_data['Fare'] = test_data['Fare'].fillna(test_data['Fare'].median())

imputer = KNNImputer( n_neighbors=5 )
columns_to_input = ['Age', 'Fare']
training_data[columns_to_input] = imputer.fit_transform( training_data[columns_to_input] )
test_data[columns_to_input] = imputer.transform( test_data[columns_to_input] )

# Separar variáveis independentes e dependentes
X_treino = training_data.drop( columns = columns_to_drop + ['Survived'], axis = 1 ) # X_treino = colunas de treino
y_treino = training_data['Survived']                                                 # y_treino = coluna de resposta

X_teste = test_data.drop( columns = columns_to_drop + ['Survived'], axis = 1 ) # X_teste = colunas de teste
y_teste = test_data['Survived']                                                # y_teste = coluna de resposta

# Balenceamento dos dados com SMOTE
smote = SMOTE( random_state = 42 )
X_resampled, y_resampled = smote.fit_resample( X_treino, y_treino )

print( "Antes do balanceamento  - X:\n", X_treino.value_counts( ) )
print( "Depois do balanceamento - X:\n", X_resampled.value_counts( ) )

print( "Antes do balanceamento  - y:\n", y_treino.value_counts( ) )
print( "Depois do balanceamento - y:\n", y_resampled.value_counts( ) )

# -------------------------------------
# --- Descobrir melhores hiperparâmetros
# -------------------------------------

# Hiperparâmetros Random Forest
rf_params = {
    'n_estimators': (50, 200),
    'max_depth': (2, 10),
    'max_features': ['sqrt', 'log2'],
    'min_samples_split': (2, 10)
}

# Otimização Random Forest
rf_modelo = BayesSearchCV(
    RandomForestClassifier( ),
    search_spaces = rf_params,
    cv = 5,
    n_jobs = -1,
    verbose = 1,
    random_state = 42
)
rf_modelo.fit( X_resampled, y_resampled )
print( "Melhores hiperparâmetros   - Random Forest:", rf_modelo.best_params_ )

# -------------------------------------
# --- Treinar o Modelo
# -------------------------------------

# Treinar modelo final com os melhores hiperparâmetros
rf_modelo_final = RandomForestClassifier( **rf_modelo.best_params_, random_state = 42 )
rf_modelo_final.fit( X_resampled, y_resampled )

# -------------------------------------
# --- Testar e Avaliar o Modelo
# -------------------------------------

# Fazer previsões
rf_pred = rf_modelo_final.predict( X_teste )

# Avaliar o modelo
print( "Acurácia do modelo         - RandomForest:"  , accuracy_score( y_teste, rf_pred ) )
print( "Matriz de Confusão         - RandomForest:\n", confusion_matrix( y_teste, rf_pred ) )
print( "Relatório de Classificação - RandomForest:\n", classification_report( y_teste, rf_pred ) )

# Matriz de Confusão
def plot_confusion_matrix( modelo, X_treino, y_treino, X_teste, y_teste ):
    cm = ConfusionMatrix( modelo )
    cm.fit( X_treino, y_treino )
```

```
    cm.fit( X_treino, y_treino )
    cm.score( X_teste, y_teste )
    cm.show( )
# plot_confusion_matrix( )

# Árvores de Decisão
def plot_decision_tree( modelo, title ):
    plt.figure( figsize=(15, 10) )
    tree.plot_tree(
        modelo,
        feature_names = X_treino.columns,
        class_names   = ['Não Sobreviveu', 'Sobreviveu'],
        filled        = True,
        rounded       = True
    )
    plt.title( title )
    plt.show( )
# plot_decision_tree( )

# Importância das features
def plot_feature_importance( modelo, X_treino ):
    importancias = modelo.feature_importances_
    features     = pd.DataFrame( {'Feature': X_treino.columns, 'Importância': importancias} )
    features     = features.sort_values( by='Importância', ascending=False )
    print( features )
# plot_feature_importance( )

print( "\nImportância dos Atributos - Random Forest" )
plot_feature_importance( rf_modelo_final, X_resampled )
```

Avaliação

ANTES

- Melhores hiperparâmetros:

```
OrderedDict({'max_depth': 10, 'max_features': 'sqrt', 'min_samples_split': 2, 'n_estimators': 53})
```

```
Acurácia do modelo: 0.8660287081339713
```

- Relatório de Classificação:

```
                precision    recall  f1-score   support

             0       0.89      0.91      0.90       266
             1       0.83      0.80      0.81       152

      accuracy                           0.87       418
     macro avg       0.86      0.85      0.85       418
  weighted avg       0.87      0.87      0.87       418
```

- Importância dos Atributos

```
      Feature  Importância
1     Sex     0.382119
5     Fare    0.226022
2     Age     0.205113
0  Pclass     0.109001
3   SibSp     0.044947
4   Parch     0.032797
```

DEPOIS

- Melhores hiperparâmetros:

```
OrderedDict({'max_depth': 10, 'max_features': 'sqrt', 'min_samples_split': 2, 'n_estimators': 200})
```

```
Acurácia do modelo: 0.8660287081339713
```

- Relatório de Classificação:

```
              precision    recall  f1-score   support

           0       0.88      0.91      0.90       266
           1       0.83      0.79      0.81       152

    accuracy                           0.87       418
   macro avg       0.86      0.85      0.85       418
weighted avg       0.87      0.87      0.87       418
```

- Importância dos Atributos:

```
   Feature  Importância
1      Sex     0.377391
5     Fare     0.221382
2      Age     0.213991
0   Pclass     0.108402
3    SibSp     0.044182
4    Parch     0.034652
```

Conclusão

A substituição da mediana pelo **KNNImputer não trouxe ganho significativo na acurácia**. No entanto:

- Houve uma pequena mudança na importância das variáveis.
- A idade passou a ter um pouco mais de impacto no modelo.
- A escolha dos hiperparâmetros foi diferente.