Campus Monterrey

Inteligencia artificial avanzada para la ciencia de datos I (Gpo 101)

Momento de Retroalimentación: Módulo 2 Análisis y Reporte sobre el desempeño del modelo. (Portafolio Análisis)

Juan Pablo Castañeda Serrano A01752030

Dataset

El dataset utilizado fue uno parecido al reto, llamado “Spaceship Titanic”(https://www.kaggle.com/competitions/spaceship-titanic) este fue escogido debido a mi familiarización con los modelos de clasificación y la estructura de sus datos.
Este dataset incluye diferentes variables, sin embargo para el entrenamiento del modelo se realizó solo con las siguientes:
[CryoSleep,Age,VIP,RoomService,FoodCourt,ShoppingMall,Spa,VRDeck]
Esto ya que son las variables con mayor porcentaje de correlación y tienen valores numéricos con pocos o nulos datos NA. Todos los demás fueron eliminados, la excepción fueron las variables de las ids de los tripulantes y su resultado final, las cuales no fueron consideradas en el entrenamiento pero fueron usadas para concatenar los valores finales con su respectivo id de tripulante

Modelo

El problema que tenemos con este dataset es similar al del reto, el poder predecir si un pasajero abordará la nave o si no. En primera instancia, podemos observar que la predicción está puesta como “False” y “True” en lugar de binarios, decidí convertir el false y el true a binario y luego volver a transformarlo a string en el momento de la concatenación. Como este problema es un problema de clasificación, sentí que el Desition Tree Clasifier. Un clasificador de árbol de decisión es un modelo de aprendizaje automático supervisado que se utiliza tanto para tareas de clasificación como de regresión. Funciona dividiendo datos de forma recursiva en subconjuntos hasta que alcanza algún criterio de terminación. El conjunto de datos sobre la nave espacial tiene una combinación de características categóricas y numéricas, Decision Trees puede manejar ambas sin mucho problema y esa fue otra razón para utilizar estas.

Entrenamiento y predicción

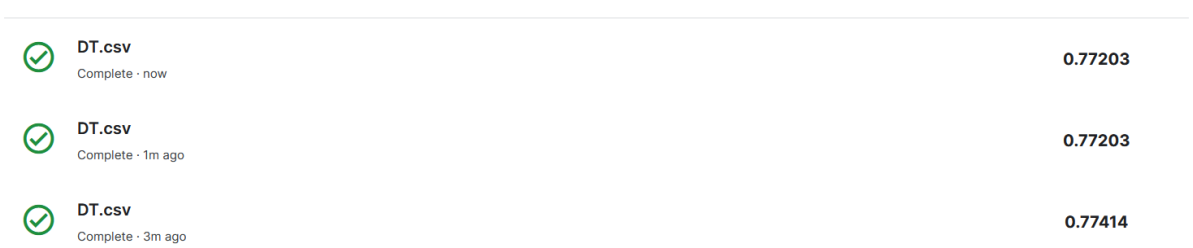
Las variables utilizadas en el modelo para crear diferentes predicciones fueron el cambio de los parámetros para la ecuación de predicción, las que se usaron fueron: “gini”, “entropy” y “log_loss”. De esta manera se hicieron 3 predicciones diferentes, sin embargo los resultados fueron equivalentes, siendo gini el que mejores resultados proveyó. Estos son los resultados:

Gini con 77.4% de accuracy:

Log-Loss con 77.2%

Entropy con 77.2%

Es interesante remarcar que Entropy y Log Loss produjeron virtualmente los mismos resultados



Análisis de modelo

Tras haber realizado un análisis de los datos y entrenado un modelo de Árbol de Decisión usando Scikit-learn, se observaron ciertas áreas de mejora y se llevó a cabo una optimización.
Modelo Inicial

El primer modelo de Árbol de Decisión (DTree) fue entrenado con los siguientes parámetros por defecto:

random_state=10
criterion="entropy"

Identificación de Problemas

Tras evaluar la precisión del modelo en el conjunto de entrenamiento y validación, observamos un posible escenario de overfitting o underfitting. Se implementó una lógica para diagnosticar el rendimiento del modelo:

Si el modelo tiene un accuracy mayor al 90% en el conjunto de entrenamiento pero menos del 70% en el conjunto de validación, entonces tiene un alto sesgo (underfitting).
Si el accuracy está por encima del 90% en el conjunto de entrenamiento y entre el 70% y 85% en el conjunto de validación, el modelo tiene un sesgo medio y varianza media.
Si ambos accuracies superan el 90%, el modelo está bien ajustado.
En otros casos, el modelo sufre de alta varianza (overfitting).

Optimización con Grid Search

Para abordar estos posibles problemas y mejorar el rendimiento del modelo, implementamos un proceso de optimización de hiperparámetros usando GridSearchCV.

Los hiperparámetros evaluados fueron:

max_depth: [None, 10, 20, 30, 40]
min_samples_split: [2, 5, 10, 20]
min_samples_leaf: [1, 2, 5, 10]

max_features: [None, 'sqrt', 'log2']
max_leaf_nodes: [None, 10, 20, 30]

A través de este proceso, el modelo fue entrenado con diferentes combinaciones de hiperparámetros para identificar la mejor configuración que mejore el rendimiento.
Modelo Optimizado

Basándonos en los resultados de GridSearchCV, entrenamos un nuevo modelo (DTreeBetter) con la siguiente configuración:

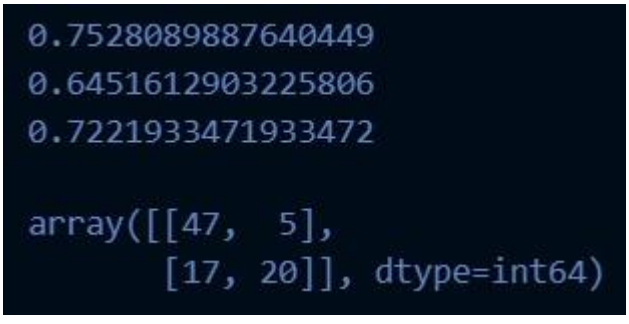
max_depth=10
max_features=None
max_leaf_nodes=30
min_samples_leaf=1
min_samples_split=2
criterion="entropy"

Evaluación del modelo

En la siguiente parte evaluaremos concretamente el modelo utilizado
Sesgo:Medio
Debido a que nuestra variable C (regularización inversa) está en un valor bajo, esto impactará en el sesgo, donde nuestro modelo está haciendo unas asunciones sobre los valores que especialmente no se encuentran suficientes datos, sin embargo esto se compensa con la varianza, ya que esta será menor a la hora de seguir entrenando al modelo
Varianza: Baja
De igual manera el mayor factor para asumir que esta varianza es baja es debido a la regularización inversa, esta al ser menor nos va a dar resultados más consistentes, todas las iteraciones de Optuna (alrededor de 1000). Nos dieron resultados similares en cuanto a la accuracy, lo que nos dice que esta es nuestra estadística más optimista
Nivel de ajuste: Adecuado
Debido a la limpieza de datos que logramos en nuestro primer entregable, este nivel se queda adecuado. Al tener los datos limpios podemos observar que gracias al tener pocas variables en nuestros data frames estos modelos funcionan mucho mejor y nos dan resultados de más del 78%.

Otras mediciones:
De arriba a abajo tenemos las mediciones:

- Accuracy
- ROC AUC
- F1 score
- Confusion matrix



Partes relevantes del código

Limpieza de los datos:

```python
import pandas as pd

def clean(name):
    df_train = pd.read_csv(name, encoding='unicode_escape', engine='python')

    df_train.drop(['HomePlanet','Cabin', 'Destination', 'Name'], axis=1, inplace=True)

    #df_train.dropna(inplace=True)

    mappingbin = {'False': 0, 'True': 1}
    df_train.replace({'CryoSleep': mappingbin, 'VIP': mappingbin}, inplace=True)

    return df_train

if __name__ == '__main__':
    clean("train.csv")
```

Modelo original y modelo mejorado

```python
def DTree(x_train, y_train, x_val):
    model_t = DecisionTreeClassifier(random_state=10, criterion="entropy")
    model_t.fit(x_train, y_train)
    y_hat_t = model_t.predict(x_val)
    return y_hat_t

def DTreeBetter(x_train, y_train, x_val):
    model_t = DecisionTreeClassifier(max_depth=10,max_features=None,max_leaf_nodes=30,min_samples_leaf=1,min_samples_split=2, criterion="entr
    model_t.fit(x_train, y_train)
    y_hat_t = model_t.predict(x_val)
    return y_hat_t
```

Concatenación

```python
def concatenar(ids, emission, name):
    final = pd.DataFrame()
    final['PassengerId'] = ids
    final['Transported'] = pd.Series(emission)
    final.to_csv(name, index=False)
    print(f"checa tu csv con nombre: {name}")
    return final
```

Evaluación (Leer comentarios del codigo)

```
if __name__ == '__main__':
    df = clean('train.csv')
    dfTest = clean('test.csv')

    x = df.drop(['PassengerId', 'Transported'], axis=1)
    x2 = dfTest.drop(['PassengerId'],axis=1)
    y = df['Transported']

    # Separar datos en entrenamiento, validación y prueba
    x_train, x_temp, y_train, y_temp = train_test_split(x, y, test_size=0.4, random_state=42)
    x_val, x_test, y_val, y_test = train_test_split(x_temp, y_temp, test_size=0.5, random_state=42)

    # Normalización
    scaler = MinMaxScaler()
    x_train_transformada_full = scaler.fit_transform(x)
    x_train_transformada = scaler.fit_transform(x_train)
    x_val_transformada = scaler.transform(x_val)
    x_test_transformada = scaler.transform(x2)

    # Entrenar y evaluar
    y_train_pred = DTree(x_train_transformada, y_train, x_train_transformada)
    y_val_pred = DTree(x_train_transformada, y_train, x_val_transformada)

    train_accuracy = accuracy_score(y_train, y_train_pred)
    val_accuracy = accuracy_score(y_val, y_val_pred)

    print(f"Accuracy en el conjunto de entrenamiento: {train_accuracy}")
    print(f"Accuracy en el conjunto de validación: {val_accuracy}")
```

```
    # Diagnóstico del modelo
    if train_accuracy > 0.9 and val_accuracy < 0.7:
        print("El modelo tiene un alto sesgo (underfitting).")
    elif train_accuracy > 0.9 and (val_accuracy >= 0.7 and val_accuracy <= 0.85):
        print("El modelo tiene un sesgo medio y varianza media.")
    elif train_accuracy > 0.9 and val_accuracy > 0.85:
        print("El modelo está bien ajustado (fit).")
    else:
        print("El modelo tiene alta varianza (overfitting).")

    # Otras funciones (graficar, generar CSV, etc.)
    # graficar(x_train_transformada, y_train)
    # concatenar(dfTest.PassengerId, y_val_pred, 'DT.csv')

    # Definir los hiperparámetros y sus posibles valores
    param_grid = {
    'max_depth': [None, 10, 20, 30, 40],
    'min_samples_split': [2, 5, 10, 20],
    'min_samples_leaf': [1, 2, 5, 10],
    'max_features': [None, 'sqrt', 'log2'],
    'max_leaf_nodes': [None, 10, 20, 30]
    }
```

Llamada final

```
DT = DTreeBetter(x_train_transformada_full, y, x_test_transformada)
concatenar(dfTest['PassengerId'], DT, 'DT.csv')
```

Conclusiones

Optimizar los hiper parámetros de un modelo es esencial para mejorar su rendimiento y evitar problemas como el overfitting y underfitting. En este caso, el uso de GridSearchCV permitió identificar una configuración más adecuada para el Árbol de Decisión, que se espera tenga un mejor rendimiento en datos no vistos.