CP1 報告

第11組-好問題
組員:E14106375 邱聖佐、F74106068 古嘉雋、F74106296 黃品叡

● 數據分析

○ 資料預處理

■ 早期版本-統一標準化

在數據集的初步瀏覽中，我們發現各個數據集的特徵數量範圍從 **4** 到 **97** 不等，並且包含了數值型和類別型的資料。根據這些觀察，我們最初設定將資料分為數值型和類別型兩類進行處理。

起初，我們對於數值型的資料做 StandardScaler 標準化，且未對類別型資料進行處理，但經過測試分析後發現，類別型資料的處理對最終結果的影響並不顯著。因此，後來我們選擇不對類別型資料進行特殊處理。

```python
def preprocess_data(X_train, X_test):
    """
    Data Preprocessing & Feature Engineering
      a) Automatically identify numerical and categorical features
      b) Feature Scaling
      c) Categorical features
    """
    # a
    numeric_features = X_train.select_dtypes(include=['float64']).columns
    categorical_features = X_train.select_dtypes(include=['int64']).columns

    # b
    scaler = StandardScaler()

    if len(numeric_features) > 0:  # if numerical features
        X_train[numeric_features] = scaler.fit_transform(X_train[numeric_features])
        X_test[numeric_features] = scaler.transform(X_test[numeric_features])

    # c

    return X_train, X_test
```

此外，我們發現某些特定的資料集在處理後的表現總是較為不佳。經過多次嘗試，我們最終決定使用現行的數據處理方法，並將其應用於所有資料集，以求得到更好的結果。

■ 現行版本-偏態程度

```python
def preprocessing(X, y):
    # clear missing value
    missing_value = X.isnull().any(axis=1) | y.isnull().any(axis=1)
    X, y = X[~missing_value], y[~missing_value]

    # clear duplicate value
    duplicate_value = pd.concat([X, y], axis=1).duplicated()
    X, y = X[~duplicate_value], y[~duplicate_value]

    # decide use minmax or zscore or yeojohnson
    # it will pass the discrete feature
    skewness = skew(X, axis=0)
    transformers = []
    for i, col in enumerate(X.columns):
        if X[col].dtype == 'int64':
            continue
        elif np.abs(skewness[i]) > 1:
            transformers.append((f'yeojohnson_{col}', PowerTransformer(method='yeo-johnson', standardize=False), [col]))
        elif np.abs(skewness[i]) < 0.5:
            transformers.append((f'standard_{col}', StandardScaler(), [col]))
        else:
            # clear outliers before minmax
            z_score = np.abs(zscore(X[col]))
            outliers = z_score > 3
            X, y = X[~outliers], y[~outliers]
            transformers.append((f'minmax_{col}', MinMaxScaler(), [col]))

    preprocessor = ColumnTransformer(transformers, remainder='passthrough')
    pipeline = Pipeline(steps=[('preprocessor', preprocessor)])

    X = pipeline.fit_transform(X)
    y = y.to_numpy().ravel()
    return X, y, pipeline
```

在現行的方法中，我們並不對所有資料進行相同的處理，而是根據每個資料集特徵數據的偏態程度做特徵縮放，針對，採用以下處理策略以改善分佈特性，提升數據分析與建模效果。

- 缺失值處理：我們猜測數據集中可能有缺失值，確保在訓練過程中不會因為缺失數據而影響模型的準確性。
- 刪除重複數據：我們猜測資料集中可能有重複的數據行並過濾掉，保證模型訓練所使用的數據是唯一且有代表性的。
- 根據偏態特徵縮放：相比於以往對數據進行統一處理的方式，目前的版本根據數據的偏態（Skewness）程度，選擇最適合的特徵縮放，針對性地改善數據分佈特性，希望提高分析與建模的準確性。
    - **Skewness > 1**：對於偏態較大的特徵，則使用 **Yeo-Johnson** 方法進行偏態矯正，對數據進行非線性轉換，適用於分佈 Nagative Skewness 與 Positive Skewness，改善模型對於不均勻分佈數據的學習效果。
    - **Skewness < 0.5**：對於偏態較小、接近對稱特徵的數值型特徵，我們使用標準化方法（**StandardScaler**）
    - **0.5 ≤ Skewness ≤ 1**：中度偏態特徵，首先計算 Z 分數 $\left(Z = \frac{x-\mu}{\sigma}\right)$，識別並去除 Z 分數超過 3 的極端值，然後使用

Min-Max Scaling，將數據縮放到 [0, 1] 的範圍，確保數據在統一尺度上分佈更均勻。

- 而外優化

一開始我們先將所有資料讀入接著做處理，後來覺得可以將讀取及處理一並進行。

```
for folder_name in os.listdir("./Competition_data"):
  if folder_name == "Dataset_15":
    dataset_names.append(folder_name)
    X_trains.append(pd.read_csv(f"./Competition_data/{folder_name}/X_train.csv",header=0))
    y_trains.append(pd.read_csv(f"./Competition_data/{folder_name}/y_train.csv",header=0))
    X_tests.append(pd.read_csv(f"./Competition_data/{folder_name}/X_test.csv",header=0))
```

```
for dataset_name in dataset_names:
    X_train = pd.read_csv(f'./Competition_data/{dataset_name}/X_train.csv')
    y_train = pd.read_csv(f'./Competition_data/{dataset_name}/y_train.csv')

    model, pipeline = processing(X_train, y_train)

    X_test = pd.read_csv(f'./Competition_data/{dataset_name}/X_test.csv')
    X_test = pipeline.transform(X_test)

    y_pred = model.predict_proba(X_test)[:, 1]
    y_pred = pd.DataFrame(y_pred)

    y_pred.columns = ["y_predict_proba"]

    y_pred.to_csv(f'./Competition_data/{dataset_name}/y_predict.csv', index=False)

    print(f"{dataset_name} done")
```

節省原本需要跑多個迴圈的時間，現行只需要跑一次迴圈。

- ## 發現的見解
  - ### 類別型特徵前處理的必要性較低:

    在初步分析中，我們觀察到，儘管尚未對類別型特徵進行專門的處理，模型已經能夠達到相當不錯的性能表現。這表明，在目前的數據特性和模型架構下，類別型特徵的處理（例如 one-hot encoding 或 label encoding）對結果的提升可能有限。

    基於這一想法，我們選擇將更多的精力集中於對模型性能影響更顯著的其他處理步驟上，而對類別型特徵的處理保持適度簡化。這樣的策略在提升模型性能的同時，也幫助我們更有效地完成數據處理流程。

    然而，類別型特徵的處理在特定場景下可能具有重要的潛在價值，特別是在類別特徵數量多的情況下。因此，在開發計畫中，我們原本預計探索更靈活且針對性的處理方法，以進一步增強模型的泛化能力和穩定性。然而，由於時間限制，我們尚未能完成這些潛在的改進，這將是未來發展的重要方向。

  - ### 偏態特徵需要特殊處理:

    偏態較大的數據特徵對模型訓練有顯著影響，特別是當偏態值超過 1 時。這些特徵如果不進行處理，可能會導致模型的訓練不穩定或 overfitting。因此，我們採用了 Yeo-Johnson 方法來矯正偏態，這在改善模型表現方面起到了關鍵作用。

  - ### 異常值對模型的影響不可忽視:

    異常值對數據的影響往往被低估，尤其是當異常值的數量較多時，它們會極大地影響模型的預測準確性。在處理異常值時，我們發現使用 z-score 超過 3 的方法可以有效去除極端數據，從而提升模型的穩定性和精確度。

- **選擇適合的特徵縮放方法對模型結果至關重要：**

一開始我們都是對所有資料集做相同的處理 StandardScaler，後來我們發現對數據特徵選擇合適的縮放方法能顯著提高模型的預測能力。

  - 對於偏態較大的數據，我們選擇 Yeo-Johnson 變換
  - 而對於偏態較小的數據，我們選擇 StandardScaler
  - 如果特徵分布對稱，則選擇 Min-Max 縮放並刪除明顯的極端值。

這些選擇使得特徵能夠更好地與模型匹配，並且改善了訓練效果。

- **模型堆疊法的有效性：**

在實驗過程中，我們嘗試了多種單一模型，並通過 `select_best_model` 函數選擇每個數據集最適合的模型。然而，我們進一步探討是否存在優化空間，於是嘗試引入堆疊模型 Stacking。

堆疊模型的核心在於結合 base_models 的預測結果，再利用 meta_model 進行整合。在我們的實做中，base_models 包括

- KNN
- 隨機森林（Random Forest）
- 支持向量機（SVM）
- XGBoost
- LightGBM

meta_model 則採用邏輯回歸（Logistic Regression）。這種方法能充分利用各基模型的優勢，提升整體性能。

```
base_models = [
    ('knn', KNeighborsClassifier(**knn_params)),
    ('rf', RandomForestClassifier(**rf_params)),
    ('svc', SVC(**svc_params)),
    ('xgb', xgb.XGBClassifier(**xgb_params)),
    ('lgb', lgb.LGBMClassifier(**lgb_params)),
]

meta_model = LogisticRegression(**meta_model_params)

stacking_model = StackingClassifier(
    estimators=base_models,
    final_estimator=meta_model,
    cv=StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
)

stacking_model.fit(X, y)
```

結果表明，堆疊模型有效平衡了不同模型的特長，顯著提升了預測準確性和穩定性，比原先 `select_best_model` 提供更強大的性能優化。

- 嘗試過的方法
  - 最初，我對所有數據集進行了相同的預處理，並套用了相同的模型。經過初步實驗後，我發現 **`RandomForestClassifier(n_estimators=100, random_state=42)`** 在最簡化的處理流程中表現出色，達到了 **0.67557** 的準確率。
  - 隨後，我針對這個模型進行了交叉驗證（Cross-Validation, CV），想透過調整超參數來進一步優化其性能。經過交叉驗證，我確定了最佳的超參數組合：

```
param_grid = {
  'n_estimators': [100, 200],
  'max_depth': [10, 20],
  'min_samples_split': [2, 5, 10],
  'min_samples_leaf': [1, 2, 4],
  'max_features': ['sqrt', 'log2']
}
```

    這一調整使得模型的表現提高至 **0.759853**。
  - 為了進一步提升模型的通用性，我開發了 **`select_best_model`** 函數，該函數會對每個數據集套用多種模型，並自動選擇最適合的模型。這樣的改進成功將模型表現提升至 **0.819854**，顯著提高了整體的預測效果。

```
def select_best_model(X_train, y_train):
  """
  Multiple models and choose the best one
  based on AUC or cross-validation scores
  """
  models = {
    'SVM': SVC(probability=True, random_state=42),
    'KNN': KNeighborsClassifier(),
    'Logistic Regression': LogisticRegression(random_state=42),
    'XGB': XGBClassifier(eval_metric='logloss', random_state=42),
    # 'LGB': lgb.LGBMClassifier(n_estimators=200, max_depth=5, learning_rate=0.1, random_state=42)
  }

  best_model = None
  best_auc = 0
  best_model_name = ""

  X_train_split, X_test_split, y_train_split, y_test_split = train_test_split(
      X_train, y_train, test_size=0.2, random_state=42, stratify=y_train
  )

  for model_name, model in models.items():
    model.fit(X_train_split, y_train_split)
    y_pred_prob = model.predict_proba(X_test_split)[:, 1]
    auc = roc_auc_score(y_test_split, y_pred_prob)

    if auc > best_auc:
      best_auc = auc
      best_model = model
      best_model_name = model_name

  best_model.fit(X_train, y_train)
  y_pred_prob = best_model.predict_proba(X_test_split)[:, 1]
  auc = roc_auc_score(y_test_split, y_pred_prob)

  return best_model, best_model_name, auc
```

  - 最後我們引入且新增了：

- 堆疊模型
- 特徵偏態進行優化與改進

在模型選擇上，我們延續了之前 `select_best_model` 函數設計的理念，進一步引入 stacking_model。堆疊模型通過結合多個基模型的預測結果，再利用元模型進行綜合判斷，有效平衡了基模型各自的優勢。這種設計不僅提升了整體模型的穩定性，也顯著提高了預測準確性。

除了模型選擇外，我們還針對資料的前處理進行了針對性的改進，特別是對數據特徵偏態的處理。我們根據每個特徵的偏態值，自動選擇適合的轉換方法，StandardScaler、MinMaxScaler、Yeo-Johnson 變換，來減少偏態對模型學習的影響。此外，對偏態較大的特徵進行預處理後，我們還清理了極端值，避免其對模型性能的負面影響。這些優化措施不僅提升了特徵的分佈均勻性，也為模型提供了更穩定的輸入數據。

最終表現 **0.876279**。這一結果表明，堆疊模型以及特徵偏態進行優化與改進，這些方法顯著提升了模型的泛化能力，進一步驗證了我們的做法和思路的正確性。

● 新開發的方法

我們並未完全開發全新的方法，但在探索過程中，我們嘗試設計了 `select_best_model` 函數，用於針對每個數據集自動篩選最適合的模型。該函數通過評估多種模型的表現，幫助我們高效地找到最佳模型。在此基礎上，我們進一步發現，僅依賴單一模型的表現可能存在一定的局限性。為了解決這一問題，我們嘗試引入堆疊模型（stacking model），將多個基模型的優勢結合起來。

堆疊模型不僅提升了我們處理不同數據集的靈活性，還能充分發揮基模型的多樣性，通過元模型進一步優化整體性能。這種方法有效彌補了單一模型的不足，成功實現了對每個數據集的性能最大化。事實證明，通過細緻的特徵工程與模型選擇，我們能夠針對每個數據集找到最佳的處理方式和模型搭配，進一步驗證了我們思路的正確性。

- 最佳超參數與效能比較
  - 在我們的實驗中，為了提升效能，我們選擇了捨棄原本的交叉驗證（CV）方法

```python
for i in range(len(dataset_names)):
  tmp_X_train, tmp_X_test, tmp_y_train, tmp_y_test = train_test_split(processed_X_trains[i], y_trains[i],
                                                                      test_size=0.2, random_state=42, stratify=y_trains[i])
  rf = RandomForestClassifier(random_state=42)
  param_grid = {
    'n_estimators': [100, 200],
    'max_depth': [10, 20],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': ['sqrt', 'log2']
  }

  grid_search = GridSearchCV(estimator=rf, param_grid=param_grid, cv=5, n_jobs=-1, verbose=1)
  grid_search.fit(tmp_X_train, tmp_y_train.squeeze())

  model = grid_search.best_estimator_
  models.append(model)
  models_name.append("Random Forest")


  tmp_y_prob = model.predict_proba(tmp_X_test)[:, 1]
  tmp_auc = roc_auc_score(tmp_y_test, tmp_y_prob)
  dataset_aucs.append(tmp_auc)

  # Now use select_best_model to see if there is a better model
  best_model, best_model_name, auc = select_best_model(processed_X_trains[i], y_trains[i])

  # Compare AUCs and update the model if needed
  if auc > tmp_auc:
    models[i] = best_model
    dataset_aucs[i] = auc
    models_name[i] = best_model_name
```

，改為使用預設的超參數，並套用於每個數據集。這樣的策略使我們能在相對短的時間內完成模型的訓練和測試。

```python
knn_params = {
    'n_neighbors': 3,          # avoid overlapping
    'weights': 'uniform',
    'algorithm': 'auto',
    'p': 2,
}
rf_params = {
    'n_estimators': 100,      # fewer trees
    'max_depth': 5,
    'min_samples_split': 2,
    'min_samples_leaf': 2,
    'max_features': 'sqrt',
    'random_state': 42,
}
svc_params = {
    'kernel': 'linear',
    'C': 0.1,                  # avoid overlapping
    'gamma': 'scale',          # avoid overlapping
    'probability': True,
    'random_state': 42,
}
```

```python
xgb_params = {
    'objective': 'binary:logistic',
    'eval_metric': 'logloss',
    'learning_rate': 0.01,
    'n_estimators': 1000,
    'max_depth': 3,
    'min_child_weight': 1,
    'subsample': 0.8,
    'colsample_bytree': 0.8,
    'reg_alpha': 0.1,                  # L1 regularization
    'reg_lambda': 0.1,                 # L2 regularization
    'scale_pos_weight': 1,             # Used for class imbalance problems
    'random_state': 42,
}
lgb_params = {
    'objective': 'binary',
    'metric': 'binary_error',
    'learning_rate': 0.01,
    'num_leaves': 31,
    'max_depth': 4,
    'min_data_in_leaf': 20,
    'lambda_l1': 0.1,                 # L1 regularization
    'lambda_l2': 0.1,                 # L2 regularization
    'max_bin': 255,
    'scale_pos_weight': 1,            # Used for class imbalance problems
    'boosting_type': 'gbdt',
    'verbose': -1,
    'random_state': 42,
}
meta_model_params = {
    'solver': 'liblinear',
    'C': 1.0,
    'random_state': 42,
}
```

- 儘管我們的調參策略相對簡單，主要通過固定超參數進行模型套用，但每個模型的調整和選擇都基於我們對數據特性的理解，這使得每個模型能夠針對特定的數據集達到最優性能。這些超參數設定和對應的模型最終使得我們的模型表現有了顯著的提升。
- 然而，我們也認為，如果時間允許，可以進一步嘗試重新套用交叉驗證（CV）來優化超參數。雖然這將大幅增加執行時間，但我們猜測這對模型的表現將有顯著的提升。未來的實驗中，可以考慮在精確度和時間成本之間進行權衡，以進一步優化結果。在未來的實驗中，我們可以在精確度和時間成本之間進行權衡，以進一步優化模型的效果。
- 為了實現這一點，我們可以設計了一個靈活的功能，讓用戶能夠根據需求選擇訓練過程中更注重「效率」還是「表現」。當選擇「效率」時，模型將使用預設的超參數快速訓練；而選擇「表現」時，模型將通過交叉驗證來尋找最佳的超參數，實現性能的最大化。

- # 為什麼使用 XGBoost 和 LightGBM

我們選擇使用 XGBoost 和 LightGBM 的主要目的是為堆疊模型（Stacking Model）中的基模型（Base Models）提供更多選擇，以增強元模型（Meta Model）整合多樣性資訊的能力。雖然 XGBoost 和 LightGBM 的強大性能在這裡可以說是「大材小用」，但這兩個模型的加入，不僅豐富了堆疊模型的基礎層結構，還進一步提升了我們處理多樣化數據的能力。雖然在本實驗中部分功能未被充分利用，但它們的潛力為我們的模型設計提供了更多可能性，也為未來更複雜的應用場景打下了堅實基礎。

此外，XGBoost 和 LightGBM 都具備透過充分的參數調整就能發揮的優勢，未來可能在更複雜的場景中發揮更大潛力：

- **XGBoost**：
  - 增量學習能力
  - 特徵重要性評估
  - 靈活的正則化控制
- **LightGBM**：
  - 高效處理類別型特徵
  - 適應不同數據規模
  - 處理不平衡數據

透過參數的靈活調整，XGBoost 和 LightGBM 能滿足對模型多樣性的需求，還能針對特定數據集的特徵進一步優化表現。然而，由於這次實驗中參數在一開始就已固定設定，且未使用交叉驗證（CV）來進一步調整優化，XGBoost 和 LightGBM 的部分優勢未能充分發揮，這在一定程度上限制了它們的潛力發揮。

```python
xgb_params = {
    'objective': 'binary:logistic',
    'eval_metric': 'logloss',
    'learning_rate': 0.01,
    'n_estimators': 1000,
    'max_depth': 3,
    'min_child_weight': 1,
    'subsample': 0.8,
    'colsample_bytree': 0.8,
    'reg_alpha': 0.1,                  # L1 regularization
    'reg_lambda': 0.1,                 # L2 regularization
    'scale_pos_weight': 1,             # Used for class imbalance problems
    'random_state': 42,
}
lgb_params = {
    'objective': 'binary',
    'metric': 'binary_error',
    'learning_rate': 0.01,
    'num_leaves': 31,
    'max_depth': 4,
    'min_data_in_leaf': 20,
    'lambda_l1': 0.1,                 # L1 regularization
    'lambda_l2': 0.1,                 # L2 regularization
    'max_bin': 255,
    'scale_pos_weight': 1,            # Used for class imbalance problems
    'boosting_type': 'gbdt',
    'verbose': -1,
    'random_state': 42,
}
```

即便如此，這兩個模型憑藉其穩定性和高度可調整性，仍然對堆疊模型的性能提升起到了關鍵作用。未來，在更複雜的實驗設計中，我們計劃引入動態參數調整和交叉驗證策略，以進一步挖掘這兩個模型的全部潛力，為模型的表現帶來更顯著的提升。