Previsão do Diagnóstico de Tumores Cerebrais

Aprendizagem Automática - Licenciatura em Engenharia Informática

Trabalho realizado por:

- André Gonçalves nº58392
- André Zhan nº58762
- Diogo Pina nº58049

Departamento de Informática, Universidade de Évora, 22 de Dezembro de 2025

Introdução

Este notebook documenta o trabalho desenvolvido para o desafio Kaggle "Diagnóstico de Tumores Cerebrais". O conjunto de dados combina atributos demográficos dos pacientes com medidas de textura extraídas de imagens ADC (por fatia), sendo a unidade de predição o paciente (agregando múltiplas fatias por id).

O objetivo deste trabalho é construir modelos, usando as técnicas aprendidas, que classifiquem tumores como maligno (1) ou benigno (0), maximizar a métrica F1 (medida oficial do desafio) e produzir submissões robustas para a leaderboard do Kaggle.

Caracterização e análise do conjunto de dados

O conjunto de dados contém atributos demográficos do paciente (idade e sexo) e 18 medidas de textura extraídas de imagens ADC (por fatia). Os dados incluem múltiplas fatias por paciente e cada linha do ficheiro do conjunto de dados corresponde a uma fatia identificada por id_fatia e ao id do paciente.

A unidade de predição é o paciente. Para treinar e validar modelos agregamos as fatias por id (por ex.: média). Esta agregação reduz variabilidade intra-paciente e transforma o problema numa classificação binária por paciente (diagnóstico: 0=benigno, 1=maligno).

O dataset contém colunas numéricas (features de textura) e categóricas/demográficas. Colunas com valores nulos devem ser listadas e tratadas por imputação (mediana para numéricas; moda para categóricas).

Imports usados

Em baixo serão indicados os imports usados no código desenvolvido para a construção e submissão dos diferentes modelos testados:

```
import argparse  # CLI args (e.g., --stratified)
import os        # filesystem ops (mkdirs)
import json      # read/write JSON
```

```
import numpy as np            # numeric ops, dtypes
import pandas as pd           # DataFrame I/O & manipulation
from sklearn.model_selection import KFold, StratifiedKFold,
GridSearchCV  # CV & grid search
from sklearn.pipeline import Pipeline  # preprocess + model
from sklearn.compose import ColumnTransformer  # column-wise
transforms
from sklearn.preprocessing import OneHotEncoder, StandardScaler  #
encoding, scaling
from sklearn.impute import SimpleImputer  # missing value imputation
from sklearn.metrics import f1_score, make_scorer  # metric and scorer
import joblib  # save/load models
```

Além disso, também faziamos import do(s) algoritmo(s) a usar, como por exemplo:

```
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

# Estes dois algoritmos serão usados para a demonstração do código
mais à frente
```

Agregação das Fatias por Id e Atributos Utilizados

A agregação das fatias numa única linha por id é feita através da função aggregate_patient_features, cujas etapas são:

- Remoção de identificadores e do 'diagnostico' (não são features);
- Separar colunas numéricas das categóricas;
- Para cada coluna numérica calcula-se mean, std, min, max, median, skew e kurt (kurtose), para resumir a variação e distribuição dos valores de intensidade do tumor;
- Para cada coluna categórica pega-se a moda (valor mais frequente). Se não houver moda, pega o primeiro elemento (esta etapa garante que o atributo é único e consistente);
- Adiciona 'n_slices' como contagem de fatias por paciente;
- Agrupa por id todas as agregações;
- Retorna um DataFrame com uma linha por paciente (id) contendo todas as features agregadas.

Esta agregação é importante para converter a representação volumétrica fragmentada das fatias num vetor de características denso e estatisticamente representativo de cada paciente, consolidando assim toda a informação dispersa numa única instância de treino robusta e independente.

De seguida está a implementação desta parte:

```
def aggregate_patient_features(df):
    df = df.copy()
    # excluir identificadores e target das features
    exclude = {"id", "id_fatia", "diagnostico"}
```

```
    feat_cols = [c for c in df.columns if c not in exclude]
    # separar numéricas e categóricas
    numeric_cols =
df[feat_cols].select_dtypes(include=[np.number]).columns.tolist()
    categorical_cols = [c for c in feat_cols if c not in numeric_cols]

    # preparar agregações para colunas numéricas
    num_aggs = {}
    for c in numeric_cols:
        # kurt é função definida inline para usar kurtosis na
agregação
        def kurt(series):
            return series.kurt()
        num_aggs[c] = ['mean', 'std', 'min', 'max', 'median', 'skew',
kurt]

    # agregação categórica: escolher o modo (mais frequente)
    cat_aggs = {c: (lambda x: x.mode().iloc[0] if not x.mode().empty
else x.iloc[0]) for c in categorical_cols}

    # contar fatias por paciente
    df['n_slices'] = 1

    # agrupar por 'id' e aplicar agregações numéricas + soma de
n_slices
    grouped_num = df.groupby('id').agg({**num_aggs, **{'n_slices':
'sum'}})

    # renomear colunas agregadas (agg produz tuplas para (col, agg))
    new_cols = []
    for col in grouped_num.columns:
        if isinstance(col, tuple):
            agg_name = col[1].__name__ if callable(col[1]) else col[1]
            new_cols.append(f"{col[0]}_{agg_name}")
        else:
            new_cols.append(col)
    grouped_num.columns = new_cols
    grouped_num = grouped_num.reset_index()

    # agregar categóricas separadamente (usando modo) e juntar
    if len(cat_aggs) > 0:
        grouped_cat = df.groupby('id').agg(cat_aggs).reset_index()
        grouped = grouped_num.merge(grouped_cat, on='id')
    else:
        grouped = grouped_num

    # se existir coluna 'diagnostico', anexar o rótulo do paciente
(usa max para consolidar)
    if 'diagnostico' in df.columns:
        diag = df.groupby('id')['diagnostico'].max().reset_index()
```

```
        grouped = grouped.merge(diag, on='id')

    return grouped
```

Preprocessamento do DataFrame

Identifica colunas numéricas e categóricas (trata sexo como categórica); para as numéricas aplica imputação pela mediana e StandardScaler (média 0, desvio 1), para as categóricas usa OneHotEncoder com drop='first' (evita dummy trap) e handle_unknown='ignore' (não falha com categorias novas); um ColumnTransformer aplica cada transformação às colunas corretas e o preprocess resultante é usado no Pipeline juntamente com o modelo.

Em baixo está apresentado o código para o preprocessamento:

```
def build_preprocessor(X):
    # identificar colunas numéricas
    numeric_features =
X.select_dtypes(include=[np.number]).columns.tolist()
    # 'sexo' é tratada como categórica -> remover de numéricas se
presente
    if 'sexo' in numeric_features:
        numeric_features.remove('sexo')
    # colunas categóricas: dtype object ou a coluna 'sexo'
    cat_features = [c for c in X.columns if X[c].dtype == object or c
== 'sexo']
    # pipeline numérica: imputar pela mediana e padronizar
    from sklearn.pipeline import Pipeline as SKPipeline
    num_pipeline = SKPipeline([
        ('imputer', SimpleImputer(strategy='median')),
        ('scaler', StandardScaler())
    ])
    # ColumnTransformer: aplica OneHot às categóricas e a num_pipeline
às numéricas
    # - drop='first' evita dummy trap (colinearidade entre colunas
binárias)
    # - handle_unknown='ignore' permite categorias novas em dados de
teste
    # - remainder='drop' descarta colunas não especificadas
    preprocess = ColumnTransformer(
        transformers=[
            ('cat', OneHotEncoder(drop='first',
handle_unknown='ignore'), cat_features),
            ('num', num_pipeline, numeric_features),
        ],
        remainder='drop'
    )
    return preprocess
```

Construção do Modelo

Na função main é onde fazemos a construção do modelo propriamente dita, recorrendo às funções apresentadas anteriormente. Esta função aceita dois argumentos relacionados com a validação cruzada, nomeadamente o nº de folds e se a validação cruzada é ou não estratificada.

As etapas da função main são:

- Carregar os dados do conjunto de treino;
- Fazer a agregação das fatias;
- Preparar o pré-processador, o algoritmo escolhido e o Pipeline;
- Escolher o esquema de validação cruzada;
- Configurar o GridSearchCV e executá-lo, para encontrar os melhores hiperparâmetros;
- Por fim, salvamos o modelo e avaliamos o seu desempenho no conjunto de treino completo.

Em baixo está apresentado o código da função main descrita:

```
# -- Para exemplificar usaremos o código em que foi utilizado o
Algoritmo LogisticRegression para construir o modelo --

def main(n_splits=5, stratified=False):
    # garantir diretórios de saída
    os.makedirs('models', exist_ok=True)
    os.makedirs('outputs', exist_ok=True)

    # carregar dados brutos e agregar por paciente
    print('Loading treino.csv...')
    df = pd.read_csv('data/treino.csv')
    df_agg = aggregate_patient_features(df)
    # X = features agregadas; y = rótulo por paciente
    X = df_agg.drop(columns=['id', 'diagnostico'])
    y = df_agg['diagnostico']
    print(f'Dataset aggregated: {X.shape[0]} patients, {X.shape[1]}
features')

    # construir pré-processador e pipeline (preprocessamento + modelo)
    preprocess = build_preprocessor(X)
    lr = LogisticRegression(random_state=42, max_iter=1000)
    pipeline = Pipeline([('preprocess', preprocess), ('model', lr)])

    # escolher esquema de cross-validation (estratificado opcional)
    if stratified:
        cv = StratifiedKFold(n_splits=n_splits, shuffle=True,
random_state=42)
        print('Using StratifiedKFold CV')
    else:
        cv = KFold(n_splits=n_splits, shuffle=True, random_state=42)
        print(f'Using KFold CV with {n_splits} splits')
```

```
    # usar F1-weighted como métrica otimizada
    scorer = make_scorer(f1_score, average='weighted')

    # grade de hiperparâmetros para GridSearchCV
    param_grid = {
        'model__C': [0.001, 0.01, 0.1, 1, 10, 100],
        'model__solver': ['lbfgs', 'liblinear', 'saga'],
        'model__penalty': ['l2', 'l1'],
        'model__class_weight': [None, 'balanced']
    }

    # executar GridSearchCV para encontrar melhores hiperparâmetros
(refit no melhor)
    print('Running GridSearchCV with KFold cross-validation optimized
for F1-weighted...')
    gs = GridSearchCV(pipeline, param_grid=param_grid, scoring=scorer,
cv=cv, n_jobs=-1, verbose=2, refit=True)
    gs.fit(X, y)
    print('\n=== GridSearchCV Results ===')
    print('Best params:', gs.best_params_)
    print(f'Best F1-weighted score (CV): {gs.best_score_:.4f}')

    # salvar parâmetros encontrados
    with open('models/best_params_lr.json', 'w') as f:
        json.dump(gs.best_params_, f, indent=2)

    # salvar o estimador final (pipeline com preprocess + modelo)
    final_est = gs.best_estimator_
    joblib.dump(final_est, 'models/model_lr.pkl')
    print('Saved final model to models/model_lr.pkl')

    # avaliar desempenho no conjunto de treino completo (apenas
informativo)
    try:
        train_preds = final_est.predict(X)
        train_f1_weighted = f1_score(y, train_preds,
average='weighted')
        train_f1_macro = f1_score(y, train_preds, average='macro')
        train_f1_micro = f1_score(y, train_preds, average='micro')
        print('\n=== Training Set Performance ===')
        print(f'Train F1-weighted (on full training set):
{train_f1_weighted:.4f}')
        print(f'Train F1-macro (on full training set):
{train_f1_macro:.4f}')
        print(f'Train F1-micro (on full training set):
{train_f1_micro:.4f}')
        # salvar métricas de treino em JSON
        with open('models/train_eval_lr.json', 'w') as f:
            json.dump({
                'train_f1_weighted': float(train_f1_weighted),
```

```
                'train_f1_macro': float(train_f1_macro),
                'train_f1_micro': float(train_f1_micro),
                'cv_best_f1_weighted': float(gs.best_score_),
                'cv_n_splits': n_splits
            }, f, indent=2)
    except Exception as e:
        # caso algo falhe na avaliação, apenas logar o erro
        print('Could not compute train F1:', e)
```

Para construir um modelo com um algoritmo diferente apenas trocávamos o algoritmo usado e os hiperparâmetros a serem testados, o resto do código e a lógica permaneciam iguais.

Para casos em que utilizámos um comité de peritos, o código seria quase igual, com adição das seguintes etapas:

- Criação da função utilitária para aplicar bootstrap;
- Criação de amostras bootstrap com reposição;
- Treinar os modelos com as amostras bootstrap;
- Por fim, ensemble por soft voting, com a média das probabilidades das várias estimativas.

De seguida está apresentada a função main com as alterações descritas:

```
# -- Para exemplificar usaremos o código em que foi utilizado o
Algoritmo LogisticRegression e SVM para criar o comité com soft voting
--

# Função utilitária: gera um bootstrap sample (com reposição)
def bootstrap_sample(X, y, n_samples=None, random_state=None):
    n = n_samples or X.shape[0]
    rng = np.random.RandomState(random_state)
    idx = rng.choice(X.index, size=n, replace=True)
    return X.loc[idx].reset_index(drop=True),
y.loc[idx].reset_index(drop=True)

# Alterações na função main...
def main(n_splits=5, stratified=False):
    # garantir diretórios de saída
    os.makedirs('models', exist_ok=True)
    os.makedirs('outputs', exist_ok=True)

    # carregar dados brutos e agregar por paciente
    print('Loading treino.csv...')
    df = pd.read_csv('data/treino.csv')
    df_agg = aggregate_patient_features(df)
    # X = features agregadas; y = rótulo por paciente
    X = df_agg.drop(columns=['id', 'diagnostico'])
    y = df_agg['diagnostico']
    print(f'Dataset aggregated: {X.shape[0]} patients, {X.shape[1]}
```

```
features')

    # construir pré-processador
    preprocess = build_preprocessor(X)

    # escolher esquema de cross-validation (estratificado opcional)
    if stratified:
        cv = StratifiedKFold(n_splits=n_splits, shuffle=True,
random_state=42)
        print('Using StratifiedKFold CV')
    else:
        cv = KFold(n_splits=n_splits, shuffle=True, random_state=42)
        print(f'Using KFold CV with {n_splits} splits')
    # usar F1-weighted como métrica otimizada
    scorer = make_scorer(f1_score, average='weighted')

    # -- antes de treinar os modelos --
    # Cria bootstrap samples (com reposição) para LR e SVM
    X_lr, y_lr = bootstrap_sample(X, y, random_state=42)
    X_svm, y_svm = bootstrap_sample(X, y, random_state=43)

    # ============== LOGISTIC REGRESSION (COM BOOTSTRAP)
==============
    print('\n--- Training Logistic Regression (bootstrap sample) ---')

    # Escolha do algoritmo e criação do pipeline
    lr_model = LogisticRegression(random_state=42, max_iter=1000,
n_jobs=-1)
    lr_pipeline = Pipeline([('preprocess', preprocess), ('model',
lr_model)])

    # Grade de hiperparâmetros para GridSearchCV
    lr_param_grid = {
        'model__C': [0.001, 0.01, 0.1, 1.0, 10.0, 100.0],
        'model__solver': ['lbfgs', 'liblinear'],
        'model__penalty': ['l2']
    }

    # Executar GridSearchCV para encontrar melhores hiperparâmetros
(refit no melhor)
    print('Running GridSearchCV for Logistic Regression (bootstrap
sample)...')
    lr_gs = GridSearchCV(lr_pipeline, param_grid=lr_param_grid,
scoring=scorer, cv=cv, n_jobs=-1, verbose=1, refit=True)
    lr_gs.fit(X_lr, y_lr)
    print('Best LR params (bootstrap):', lr_gs.best_params_)
    print(f'Best LR F1-weighted score (CV on bootstrap):
{lr_gs.best_score_:.4f}')

    # Salvar os melhores hiperparâmetros encontrados e o modelo
```

```
treinado
    with open('models/best_params_lr.json', 'w') as f:
        json.dump({k.split('__')[1]: v for k, v in
lr_gs.best_params_.items()}, f, indent=2)
    lr_final = lr_gs.best_estimator_
    joblib.dump(lr_final, 'models/model_lr_bootstrap.pkl')
    print('Saved LR model (bootstrap) to
models/model_lr_bootstrap.pkl')

    # ============== SUPPORT VECTOR MACHINE (COM BOOTSTRAP)
==============
    print('\n--- Training Support Vector Machine (bootstrap sample)
---')

    # Escolha do algoritmo e criação do pipeline
    svm_model = SVC(kernel='rbf', random_state=42, probability=True)
    svm_pipeline = Pipeline([('preprocess', preprocess), ('model',
svm_model)])

    # Grade de hiperparâmetros para GridSearchCV
    svm_param_grid = {
        'model__C': [0.1, 1.0, 10.0, 100.0],
        'model__gamma': [0.001, 0.01]
    }

    # Executar GridSearchCV para encontrar melhores hiperparâmetros
(refit no melhor)
    print('Running GridSearchCV for SVM (bootstrap sample)...')
    svm_gs = GridSearchCV(svm_pipeline, param_grid=svm_param_grid,
scoring=scorer, cv=cv, n_jobs=-1, verbose=1, refit=True)
    svm_gs.fit(X_svm, y_svm)
    print('Best SVM params (bootstrap):', svm_gs.best_params_)
    print(f'Best SVM F1-weighted score (CV on bootstrap):
{svm_gs.best_score_:.4f}')

    # Salvar os melhores hiperparâmetros encontrados e o modelo
treinado
    with open('models/best_params_svm.json', 'w') as f:
        json.dump({k.split('__')[1]: v for k, v in
svm_gs.best_params_.items()}, f, indent=2)
    svm_final = svm_gs.best_estimator_
    joblib.dump(svm_final, 'models/model_svm_bootstrap.pkl')
    print('Saved SVM model (bootstrap) to
models/model_svm_bootstrap.pkl')

    # ============== ENSEMBLE (média de probabilidades dos modelos
treinados por bootstrap) ==============
    print('\n--- Ensemble (média de probabilidades dos modelos
treinados por bootstrap) ---')
    ensemble_proba = (lr_final.predict_proba(X)[:, 1] +
```

```
svm_final.predict_proba(X)[:, 1]) / 2.0
    ensemble_preds = (ensemble_proba >= 0.5).astype(int) # threshold
padrão
    ensemble_f1 = f1_score(y, ensemble_preds, average='weighted')

    # Métricas individuais sobre o conjunto completo (apenas para
comparação)
    lr_f1 = f1_score(y, lr_final.predict(X), average='weighted')
    svm_f1 = f1_score(y, svm_final.predict(X), average='weighted')

    # Resultados
    print(f'Ensemble F1-weighted: {ensemble_f1:.4f}')
    print(f'LR F1-weighted: {lr_f1:.4f}')
    print(f'SVM F1-weighted: {svm_f1:.4f}')

    # Salva modelos treinados
    joblib.dump({'lr': lr_final, 'svm': svm_final},
'models/ensemble_lr_svm_models.pkl')
    print('Saved bootstrap-trained models to
models/ensemble_lr_svm_models.pkl')
```

O trecho de código a seguir é usado auxiliarmente para configurar os argumentos da função main:

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--n_splits', type=int, default=5,
help='Number of KFold splits (non-stratified)')
    parser.add_argument('--stratified', action='store_true', help='Use
StratifiedKFold (optional)')
    args = parser.parse_args()
    # A linha em baixo está em comentário para não executar a função
main e evitar erro na célula
    # main(n_splits=args.n_splits, stratified=args.stratified)
```

Criação das Previsões

Agora apresentamos o código para a criação dos ficheiros de submissão (implementado num ficheiro .py separado), que são as previsões ao conjunto de teste. Para tal são carregados os dados do conjunto de teste, é feita a agregação das fatias por paciente, como feito anteriormente para o conjunto de treino e, por fim, são feitas as previsões por paciente:

```
def main(model_path='models/model.pkl', test_csv='data/teste.csv',
output='outputs/submission.csv'):
    # Garante que a pasta de outputs existe (onde a submissão será
escrita)
    os.makedirs('outputs', exist_ok=True)
```

```
    # Verifica se o arquivo do modelo existe e carrega-o com joblib
    # O 'model_path' deve apontar para um pipeline ou modelo
serializado
    if not os.path.exists(model_path):
        raise FileNotFoundError(f'Model not found: {model_path}')
    model = joblib.load(model_path)

    # Lê o CSV de teste com as fatias (uma linha por fatia)
    df_test = pd.read_csv(test_csv)

    # Função local para agregar as fatias por paciente (igual ao usado
no treino)
    # Objetivo: transformar múltiplas fatias por 'id' em uma linha por
paciente com estatísticas
    def aggregate(df):
        df = df.copy()
        # Colunas que não são features por fatia
        exclude = {"id", "id_fatia", "diagnostico"}
        feat_cols = [c for c in df.columns if c not in exclude]

        # Identifica colunas numéricas e categóricas
        numeric_cols =
df[feat_cols].select_dtypes(include=[np.number]).columns.tolist()
        cat_features = [c for c in feat_cols if c not in numeric_cols]

        # Para cada coluna numérica calculamos várias agregações
(média, desvio, min, max, mediana, skew, kurtosis)
        num_aggs = {}
        for c in numeric_cols:
            # kurt é função definida inline para usar kurtosis na
agregação
            def kurt(series):
                return series.kurt()
            num_aggs[c] = ['mean', 'std', 'min', 'max', 'median',
'skew', kurt]

        # Para colunas categóricas usamos a moda (valor mais
frequente)
        cat_aggs = {c: (lambda x: x.mode().iloc[0] if not
x.mode().empty else x.iloc[0]) for c in cat_features}

        # Conta número de fatias por paciente
        df['n_slices'] = 1

        # Aplica agregações numéricas por 'id'
        grouped_num = df.groupby('id').agg({**num_aggs, **{'n_slices':
'sum'}})

        # renomear colunas agregadas (agg produz tuplas para (col,
agg))
```

```
        new_cols = []
        for col in grouped_num.columns:
            if isinstance(col, tuple):
                agg_name = col[1].__name__ if callable(col[1]) else
col[1]
                new_cols.append(f"{col[0]}_{agg_name}")
            else:
                new_cols.append(col)
        grouped_num.columns = new_cols
        grouped_num = grouped_num.reset_index()

        # Agrega colunas categóricas (se existirem) e faz merge com
agregados numéricos
        if len(cat_aggs) > 0:
            grouped_cat = df.groupby('id').agg(cat_aggs).reset_index()
            grouped = grouped_num.merge(grouped_cat, on='id')
        else:
            grouped = grouped_num

        return grouped

    # Agrega o conjunto de teste por paciente
    df_test_agg = aggregate(df_test)

    # Remove coluna 'id' para obter matriz de features compatível com
o modelo
    X_test = df_test_agg.drop(columns=['id'])

    # Faz previsões por paciente
    preds = model.predict(X_test)

    # Cria DataFrame de submissão e persiste em CSV
    sub = pd.DataFrame({'id': df_test_agg['id'].values, 'diagnostico':
preds})
    sub.to_csv(output, index=False)
    print(f'Wrote submission to {output}')
```

Para configurar os argumentos da função main responsável pelas previsões, permitindo manipular o modelo a ser usado e o nome do ficheiro de submissão a ser criado, é usado auxiliarmente o seguinte trecho de código:

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--model', type=str,
default='models/model.pkl')
    parser.add_argument('--test', type=str, default='data/teste.csv')
    parser.add_argument('--output', type=str,
default='outputs/submission.csv')
    args = parser.parse_args()
```

```
    # A linha em baixo está em comentário para não executar a função
main e evitar erro na célula
    # main(model_path=args.model, test_csv=args.test,
output=args.output)
```

Estratégia para Pesquisa e Escolha dos Modelos

A nossa estratégia para a pesquisa dos modelos foi inicialmente testar todos os diferentes algoritmos aprendidos no decorrer das aulas, e juntar com ensemble os que tiveram melhor score na parte pública do conjunto de teste.

Para a construção dos modelos utilizámos sempre GridSearchCV para procurar os melhores hiperparâmetros e optámos sempre por usar validação cruzada estratificada com 5 folds pois decidimos que era a melhor configuração para a validação cruzada, após fazer alguns testes. Além disso utilizámos sempre os mesmos atributos e transformações mencionados anteriormente na agregação por fatias para eliminar a necessidade de tratar múltiplas linhas por paciente na fase de treino/avaliação.

A escolha dos 3 modelos finais resumiu-se principalmente à nossa percepção da capacidade de generalização dos modelos, de forma a evitar sobre-ajustamento ao conjunto de treino e assegurar a capacidade de generalização em dados novos, visando maximizar o score na parte privada do conjunto de teste.

Descrição do Subconjunto dos Modelos Criados

Todos os modelos criados e submetidos envolveram o uso de algoritmos como: LogisticRegression, K-Nearest Neighbors, Support Vector Machine, Naive Bayes, Decision Tree, Random Forest, Extra Trees, Gradient Boosting Trees e Multi-Layer Perceptron.

Em acréscimo, em todos os modelos submetidos foi aplicado o GridSearchCV para a procura dos melhores hiperparâmetros, assim como validação cruzada estratificada com 5 folds, com exceção de um modelo em que usamos 10 folds em vez de 5. Para a procura dos melhores hiperparâmetros, nós testamos para cada algoritmo os hiperparâmetros mais usuais e indicados nas aulas teóricas.

Nós fizemos questão de mencionar o algoritmo usado em cada modelo submetido no kaggle, incluindo o nome do modelo no nome do ficheiro de submissão, apesar de que, nas nossas primeiras submissões, não o fizemos, pois ainda não estávamos totalmente organizados.

Caracterização dos 3 Modelos Submetidos

submission_lr_0.8166.csv

Neste modelo foi usado o algoritmo LogisticRegression e validação cruzada estratificada com 5 folds. Para o GridSearchCV testámos os seguintes hiperparâmetros:

```
param_grid = {
        'model__C': [0.001, 0.01, 0.1, 1, 10, 100],
```

```
        'model__solver': ['lbfgs', 'liblinear', 'saga'],
        'model__penalty': ['l2', 'l1'],
        'model__class_weight': [None, 'balanced']
    }
```

Sendo que os melhores hiperparâmetros encontrados foram:

```
best_param = {
  "model__C": 0.01,
  "model__class_weight": "balanced",
  "model__penalty": "l2",
  "model__solver": "liblinear"
}
```

Além disso, como indicado no nome do ficheiro, o F1-Weighted score no conjunto de treino completo foi 0.8166.

O desempenho deste modelo na parte pública do conjunto de teste foi de 0.769, enquanto que na parte privada foi 0.709.

submission_svm_0.7852.csv

Neste modelo foi usado o algoritmo Support Vector Machine e validação cruzada estratificada com 5 folds. Para o GridSearchCV testámos os seguintes hiperparâmetros:

```
param_grid = {
            'model__C': [0.01, 0.1, 1, 10, 100],
            'model__kernel': ['linear', 'rbf', 'poly'],
            'model__gamma': [0.001, 0.01],
            'model__degree': [2, 3],
            'model__class_weight': [None, 'balanced']
        }
```

Sendo que os melhores hiperparâmetros encontrados foram:

```
best_param = {
  "model__C": 1,
  "model__class_weight": None,
  "model__degree": 2,
  "model__gamma": 0.001,
  "model__kernel": "rbf"
}
```

Além disso, como indicado no nome do ficheiro, o F1-Weighted score no conjunto de treino completo foi 0.7852.

O desempenho deste modelo na parte pública do conjunto de teste foi de 0.740, enquanto que na parte privada foi 0.600.

submission_nb_0.7500.csv

Neste modelo foi usado o algoritmo Naive Bayes e validação cruzada estratificada com 5 folds. Para o GridSearchCV testámos os seguintes hiperparâmetros:

```
param_grid = {
            'model__var_smoothing': [1e-12, 1e-11, 1e-10, 1e-9, 1e-8,
1e-7]
        }
```

Sendo que os melhores hiperparâmetros encontrados foram:

```
best_param = {
  "model__var_smoothing": 1e-12
}
```

Além disso, como indicado no nome do ficheiro, o F1-Weighted score no conjunto de treino completo foi 0.7500.

O desempenho deste modelo na parte pública do conjunto de teste foi de 0.740, enquanto que na parte privada foi 0.705.

Observações

É ainda importante mencionar que nos 3 diferentes modelos escolhidos foi usada a mesma engenharia de atributos, recorrendo à função inicialmente apresentada responsável pela agregação das fatias, em que para cada coluna numérica calcula-se mean, std, min, max, median, skew e kurt (kurtose), adicionando ainda 'n_slices' como contagem de fatias por paciente.

Discussão e Conclusões

Ao usar os diferentes algoritmos mencionados e tendo como referência o desempenho dos modelos na parte pública, chegámos à conclusão que comités de árvores de decisão (Random Forest, Extra Trees e Gradient Boosting Machine) não generalizavam tão bem em dados nunca vistos como algoritmos mais simples, como por exemplo Naive Bayes e LogisticRegression. Nós achamos que a justificação para isto deve-se ao tamanho do conjunto de dados, que é pequeno/médio, e também à sensibilidade do modelo a pequenas variações nos dados de treino e, por isso, algoritmos como Random Forest vão sobre-ajustar-se facilmente ao conjunto de treino e, consequentemente, não terem um bom desempenho em dados novos.

Dito isto, na escolha final dos 3 modelos, nós escolhemos modelos construídos sem comités de árvores de decisão, visando evitar complexidade e maximizando a capacidade de generalização em dados novos.

Quanto aos modelos finais escolhidos, o desempenho dos mesmos teve pequenas diferenças da parte pública para a privada, à exceção do modelo em que foi usado o algoritmo SVM, que teve uma diferença de desempenho de 0.140, sendo o menor score o da parte privada, o que demonstra má generalização por parte do modelo.

Acrescentando, o modelo submetido no Kaggle pela nossa equipa com o desempenho mais alto na parte privada do conjunto de teste foi o submission_dt_0.8314.csv, o qual foi construído com o algoritmo Decision Tree. O seu desempenho na parte pública foi de 0.580, enquanto que na privada foi de 0.833. Achámos este resultado estranho inicialmente, mas o mesmo pode ser explicado pela eventual e possível existência de variância nas amostras da parte pública para a parte privada e, consequentemente, o modelo pode "ter azar" na parte pública, mas performar melhor na parte privada.