

Trabalho 1 da disciplina de Aprendizado de máquina - Análise de um conjunto de dados sobre câncer de próstata

Giovani D. Silva¹

¹Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)

{giovani.silva}@inf.ufrgs.br

Resumo. O objetivo deste trabalho é usar algoritmos de aprendizado de máquina para classificar casos de câncer de próstata como benignos ou malignos com base em um conjunto de dados disponível no site Kaggle (<https://www.kaggle.com/datasets/sajidsaifi/prostate-cancer>). O conjunto de dados consiste em informações de 100 pacientes, composto por 9 variáveis, sendo 8 numéricas e uma categórica.

As variáveis numéricas incluem: Raio, Textura, Perímetro, Área, Suavidade, Compactação, Simetria e Dimensão Fractal. A variável categórica representa o "resultado diagnóstico" e indica se o caso é benigno ou maligno. Esta é a variável / classe a ser classificada.

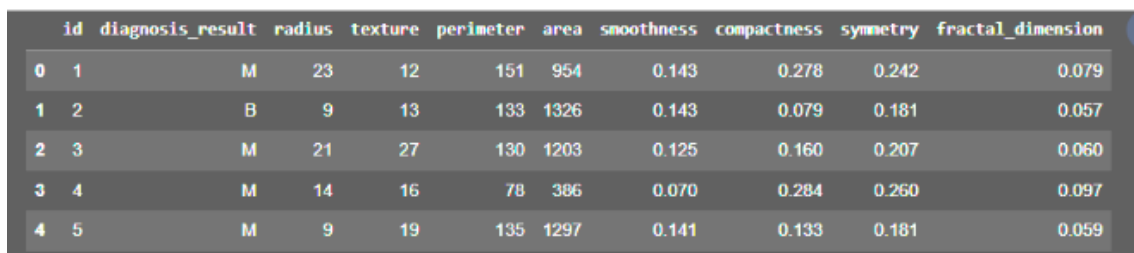
Esta informação é usada para treinar o algoritmo AM para rotular casos benignos e interpretar os resultados. Diferentes algoritmos AM foram aplicados, como Naive Bayes, Random Forest, Support Vector Machine e KNN, e o desempenho dos modelos foi avaliado por meio de diversas métricas, que serão descritas posteriormente.

1. Informações sobre a implementação

O trabalho foi implementado em Python3.7, mais precisamente em um Jupyter Notebook via google colab, utilizando-se de bibliotecas como NumPy, sci-kit learn e Pandas.

2. Pré-processamento

O pré-processamento foi realizado com a biblioteca Pandas, primeiramente, foi utilizado o comando `df.head(5)` para obter informações sobre os atributos e a classe



	id	diagnosis_result	radius	texture	perimeter	area	smoothness	compactness	symmetry	fractal_dimension
0	1	M	23	12	151	954	0.143	0.278	0.242	0.079
1	2	B	9	13	133	1326	0.143	0.079	0.181	0.057
2	3	M	21	27	130	1203	0.125	0.160	0.207	0.060
3	4	M	14	16	78	386	0.070	0.284	0.260	0.097
4	5	M	9	19	135	1297	0.141	0.133	0.181	0.059

Figura 1. Execução do código `df.head(5)`

Seguido disto, foi executado o comando `df.info()` para obter-se informação sobre o tipo dos dados e se existiam valores faltantes no dataset.

#	Column	Non-Null Count	Dtype
0	id	100 non-null	int64
1	diagnosis_result	100 non-null	object
2	radius	100 non-null	int64
3	texture	100 non-null	int64
4	perimeter	100 non-null	int64
5	area	100 non-null	int64
6	smoothness	100 non-null	float64
7	compactness	100 non-null	float64
8	symmetry	100 non-null	float64
9	fractal_dimension	100 non-null	float64
dtypes: float64(4), int64(5), object(1)			

Figura 2. Execução do código df.head(5)

Com a execução de ambos comandos percebe-se que não existem valores faltantes e todos os atributos são numéricos, dos tipos int e float.

Após isso, foi executado o comando `df = df.drop(['id'],axis=1)` de modo que a coluna id fosse excluída e não influenciasse em treinos futuros.

	diagnosis_result	radius	texture	perimeter	area	smoothness	compactness	symmetry	fractal_dimension
0	0	23	12	151	954	0.143	0.278	0.242	0.079
1	1	9	13	133	1326	0.143	0.079	0.181	0.057
2	0	21	27	130	1203	0.125	0.160	0.207	0.060

Figura 3. Execução do código df.head(5) após df.drop

Após, buscou-se entender o intervalo de valor de cada atributo

```
#valores minimos de cada coluna
df.min()

diagnosis_result    0.000
radius              9.000
texture             11.000
perimeter           52.000
area                202.000
smoothness          0.070
compactness          0.038
symmetry            0.135
fractal_dimension   0.053
dtype: float64

#valores maximos de cada coluna
df.max()

diagnosis_result    1.000
radius             25.000
texture            27.000
perimeter          172.000
area              1878.000
smoothness          0.143
compactness         0.345
symmetry            0.304
fractal_dimension   0.097
dtype: float64
```

Figura 4. Execução do código df.min e após df.max

Observou-se que há uma discrepância grande de valores entre alguns atributos do dataset, o que pode influenciar negativamente em algoritmos como o SVM, apesar de não influenciar significativamente o desempenho do Naive Bayes e do Random Forest, a decisão foi normalizar os atributos para valores entre 0 e 1 utilizando o método Min-MaxScaler() da biblioteca sci-kit learn

```
11) normalizar entre 0 e 1 os atributos
scaler = MinMaxScaler()
df[['radius', 'texture', 'perimeter', 'area', 'smoothness', 'compactness', 'symmetry', 'fractal_dimension']] = scaler.fit_transform(df[['radius', 'texture', 'perimeter', 'area', 'smoothness', 'compactness', 'symmetry', 'fractal_dimension']])
df.tail()
```

	diagnosis_result	radius	texture	perimeter	area	smoothness	compactness	symmetry	fractal_dimension
95	0	0.0750	0.3125	0.666667	0.833333	0.707271	0.307937	0.445787	0.068183
96	1	0.8125	0.1875	0.716667	0.166666	0.479437	0.107437	0.375444	0.795433
97	1	0.6250	1.0000	0.083333	0.083409	0.480396	0.048960	0.000000	0.363636
98	1	0.7500	0.8125	0.183333	0.155895	0.273973	0.105021	0.159763	0.795433
99	0	0.4375	1.0000	0.300000	0.363126	0.363062	0.247057	0.313609	0.200000

Figura 5. Execução da normalização

Também foi usado uma matriz de correlação (figura 6) por meio da biblioteca Pandas para determinar a correlação entre os atributos de modo a evitar informações redundantes.

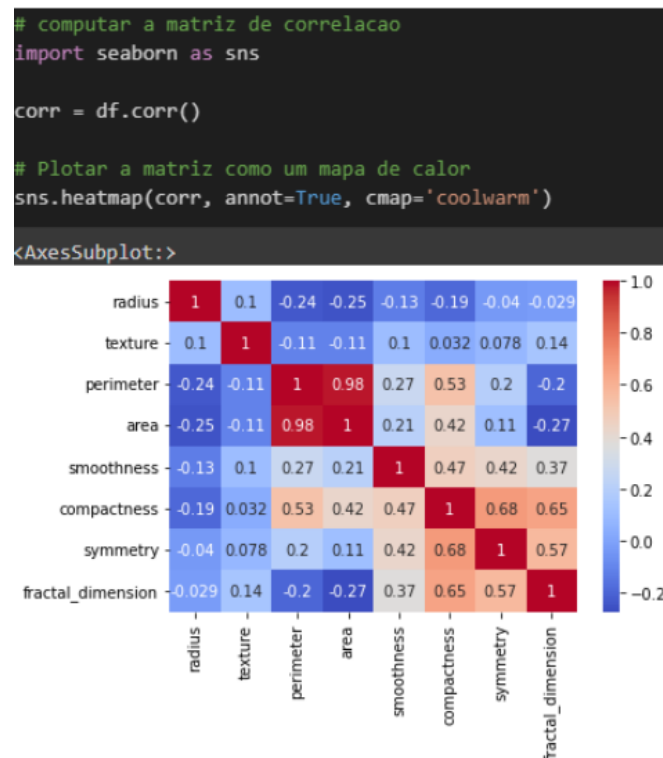


Figura 6. Execução do código da matriz de correlação

Por meio dessa matriz percebe-se que perímetro e área estão extremamente correlacionados, o que pode ser danoso para o dataset, em função disso, houve a retirada do atributo perímetro.

3. Algoritmos escolhidos

Os algoritmos escolhidos para a classificação do dataset tem como base de escolha, mas não só, o seguinte cheat sheet apresentao na disciplina de Aprendizado de Máquina.

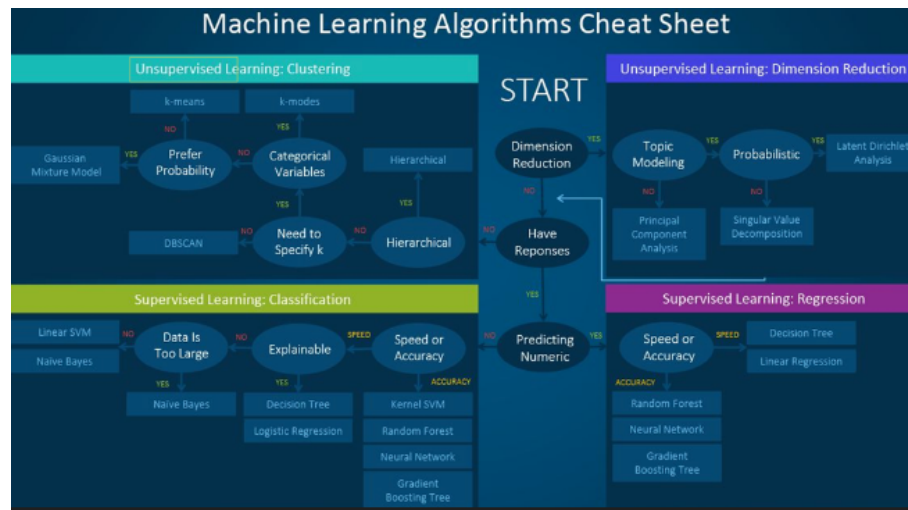


Figura 7. Cheat Sheet disponibilizado

7

Os algoritmos escolhidos foram Naive Bayes, Random Forest, Support Vector Machine e KNN, por suas características de lidarem bem com problemas de Aprendizado Supervisionado de classificação, os mesmos serão descritos a seguir e seus resultados no dataset serão avaliados num tópico futuro.

3.1. Naive Bayes

Naive Bayes é um modelo de aprendizado de máquina usado para classificar dados em problemas de análise preditiva. Baseia-se no teorema de Bayes, que estabelece a probabilidade condicional de uma classe dada a ocorrência de um conjunto de características ou atributos.

Uma das principais vantagens do Naive Bayes é sua simplicidade e eficiência ao lidar com grandes conjuntos de dados. O modelo assume que todos os atributos têm o mesmo efeito sobre a variável dependente, o que torna a implementação e interpretação do modelo relativamente simples.

Nesse trabalho, é utilizada uma distribuição Gaussiana Naive Bayesiana, que assume que cada atributo segue uma distribuição normal e utiliza essa informação para estimar a probabilidade de pertencer a uma classe. Isso é feito calculando a média e o desvio padrão de cada recurso para cada classe e usando esses valores para calcular a probabilidade usando a fórmula de distribuição normal.

O hiperparâmetro principal utilizado foi o var smoothing, O var smoothing é um parâmetro do modelo de classificação Naive Bayes Gaussian que controla a adição de uma pequena quantidade (chamada de suavização) de variância a todas as variáveis do conjunto de dados, incluindo aquelas que têm variância zero. Esse processo é conhecido

como suavização de Laplace. Essa suavização é útil para evitar problemas de divisão por zero e estabilizar a estimativa de probabilidade do modelo.

Para avaliar qual o melhor hiperparâmetro ou melhor combinação de hiperparâmetros para os algoritmos, foi utilizado o GridSearch, que em resumo cria um grid(grid) com as combinações de valores de hiperparâmetros especificadas e testa as melhores combinações possíveis, retornando o melhor desempenho para uma certa combinação.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.2)
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import GridSearchCV

classifier = GaussianNB()

#values to be tested as var smoothing
param_grid = {'var_smoothing': [0, 1e-100, 1e-25, 1e-20, 1e-15, 1e-13, 1e-11,

grid_search = GridSearchCV(classifier, param_grid, cv=5)
grid_search.fit(X_train, y_train)

y_pred = grid_search.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)

print("Best parameters: ", grid_search.best_params_)
print(f"Accuracy: {accuracy:.2f}")

>>>Best parameters:  {'var_smoothing': 0}
>>>Accuracy: 0.84
```

Foram testados diversos valores de var smoothing, porém como o dataset é pequeno e apresenta pouca variação nos valores, o melhor valor retornado foi zero.

Abaixo temos a execução de um código com esse valor.

```
from sklearn.naive_bayes import GaussianNB
classifier = GaussianNB(var_smoothing = 0)

# Train classifier on training data
classifier.fit(X_train, y_train)

# Evaluate classifier on testing data
y_pred = classifier.predict(X_test)
from sklearn.metrics import accuracy_score, precision_score, recall_score
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
```

```
f1 = f1_score(y_test, y_pred)
print(f"Naive Bayes >>>>> Accuracy: {accuracy:.2f}, Precision: {precision:.2f}, Recall: {recall:.2f}, F1: {f1:.2f}")

>>> Naive Bayes >>>>> Accuracy: 0.84, Precision: 0.84, Recall: 0.84, F1: 0.84
```

3.2. Random Forest

O algoritmo de floresta aleatória envolve a criação de várias árvores de decisão independentes, onde cada árvore de decisão é treinada usando uma parte aleatória do conjunto de dados. Após a criação dessas árvores, elas são combinadas para gerar a previsão final. A aleatoriedade é introduzida em duas etapas: selecionar amostras aleatoriamente para treinar cada árvore e selecionar aleatoriamente um subconjunto de recursos (variáveis) para cada divisão de nó.

```
# hyperparameters for Random forest
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import make_scorer, accuracy_score
#Possible hyperparameters
param_grid = {
    'n_estimators': [10, 30, 50, 100, 200, 300],
    'max_depth': [1, 5, 10, 20, 30, 40],
    'min_samples_split': [2, 4, 6, 8, 10, 15],
    'min_samples_leaf': [1, 2, 3, 5, 7],
    'max_features': ['sqrt', 'log2', None]
}

classifier = RandomForestClassifier(random_state=0)

scorer = make_scorer(accuracy_score)

grid_search = GridSearchCV(classifier, param_grid=param_grid, scoring='accuracy')

grid_search.fit(X_train, y_train)

# Avalia o modelo com o dataset de teste
y_pred = grid_search.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Random Forest >>>>> Best params: {grid_search.best_params_}, Accuracy: {accuracy:.2f}")

>>> Random Forest >>>>> Best params: {'max_depth': 1, 'max_features': 'sqrt'}
```

Os hiperparâmetros da floresta aleatória afetam o desempenho do modelo e podem ser ajustados para melhorar a precisão e evitar o overfitting. Alguns hiperparâmetros principais incluem:

- n_estimators: Número de árvores na floresta. Quanto maior o número de árvores,

mais preciso é o modelo, mas também mais lento o tempo de treinamento. Então, para não ficar muito caro, os hiperparâmetros testados foram: 10, 30, 50, 100, 200, 300

max depth: A profundidade máxima de cada árvore. Árvores mais profundas podem capturar relacionamentos mais complexos nos dados, mas também são mais propensas a overfitting. Os valores testados são: 5,10,20,30,40

min samples split: O número mínimo de amostras necessárias para dividir um nó de árvore interna. Valores mais altos podem impedir que a árvore cresça muito rápido e reduzir o overfitting, mas isso pode levar a uma menor precisão do modelo. Os valores testados são: 1,2,4,6,8,10,15

min samples leaf: O número mínimo de amostras necessárias em uma folha. Valores mais altos podem reduzir o overfitting, mas também resultar em menor precisão. Os valores testados são: 1, 2, 3, 5, 7

max features: é uma forma de controlar a complexidade do modelo, o número máximo de recursos (variáveis) considerados para cada divisão de nó. Valores menores podem reduzir a correlação entre as árvores na floresta e aumentar a diversidade, mas também podem reduzir a precisão do modelo.

Abaixo, uma execução de random forest com esses valores otimizados.

```
from sklearn.ensemble import RandomForestClassifier
classifier = RandomForestClassifier(n_estimators = 30, criterion = 'entropy')
# Train classifier on training data
classifier.fit(X_train, y_train)

# Evaluate classifier on testing data
y_pred = classifier.predict(X_test)

from sklearn.metrics import accuracy_score, precision_score, recall_score
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, average='macro')
recall = recall_score(y_test, y_pred, average='macro')
f1 = f1_score(y_test, y_pred, average='macro')
print(f"Random Forest >>>>> Accuracy: {accuracy:.2f}, Precision: {precision:.2f}, Recall: {recall:.2f}")

>>> Random Forest >>>>> Accuracy: 0.80, Precision: 0.82, Recall: 0.77,
```

3.3. KNN

KNN (K-Nearest Neighbors) é um algoritmo de aprendizado de máquina supervisionado usado para classificação ou regressão. Baseia-se na ideia de que objetos semelhantes geralmente estão próximos uns dos outros no espaço de recursos. O "k" no nome refere-se ao número de vizinhos mais próximos usados na classificação ou regressão. Por exemplo, se K=3, o algoritmo usa os três vizinhos mais próximos de um objeto desconhecido para determinar sua classe ou valor. Na classificação, KNN usa o status (valor mais frequente) de K classes de vizinhos mais próximos para determinar a classe desconhecida.

KNN funciona melhor em conjuntos de dados com muitas amostras e poucos re-

curso, onde a relação entre as amostras é bem definida. Apesar das poucas amostras deste dataset, ele apresentou resultados ainda sim satisfatórios devido aos seus dados serem bem estruturados.

O hiperparâmetro `n_neighbors` do algoritmo KNN especifica o número de vizinhos mais próximos usados para classificação ou regressão de um objeto desconhecido. Escolher o valor ideal de `n` vizinhos pode afetar significativamente o desempenho do modelo. Se o valor de `n` vizinhos for muito pequeno, o algoritmo KNN se torna mais sensível a dados inconsistentes e pode levar a um modelo instável com alta variância. Isso ocorre porque o modelo pode "lembrar" o treinamento, não consegue generalizar para novos dados. Se o valor de `n` vizinhos for muito alto, o algoritmo KNN torna-se mais propenso a erros de classificação e pode resultar em um modelo com alto viés e baixa generalização. Isso porque o modelo ignora as nuances dos dados e trata todos os casos como se fossem iguais. Portanto, as previsões geradas pelo modelo podem não ser suficientemente precisas, comprometendo sua capacidade de generalização para novos dados e prejudicando o modelo.

Abaixo, vamos a determinação desse hiperparâmetro com `GridSearch`.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.2)

knn = KNeighborsClassifier()
from sklearn.model_selection import GridSearchCV

param_grid = {'n_neighbors': np.arange(1, 12)}
grid_search = GridSearchCV(knn, param_grid, cv=5)
grid_search.fit(X_train, y_train)

print("Best n_neighbors:", grid_search.best_params_['n_neighbors'])
print("Accuracy:", grid_search.best_score_)

>>>Best n_neighbors: 3
>>>Accuracy: 0.8533333333333333
```

Abaixo, um treinamento desse algoritmo com esses hiperparâmetros otimizados.

```
# Create KNN classifier
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=5)

# Fit the classifier to the training data
knn.fit(X_train, y_train)

# Make predictions on the test data
y_pred = knn.predict(X_test)

# Calculate accuracy
from sklearn.metrics import accuracy_score, precision_score, recall_score
```

```

accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
print(f"KNN >>>>> Accuracy: {accuracy:.2f}, Precision: {precision:.2f},
>>> KNN >>>>> Accuracy: 0.76, Precision: 0.64, Recall: 0.90, F1-score:

```

3.4. Support Vector Machine

O SVM cria uma curva que separa os dados em duas classes, com o objetivo de maximizar a distância entre essa curva e os pontos mais próximos de cada classe.

Esse processo é chamado de "margem máxima". O SVM é muito útil quando as classes não são facilmente separáveis por uma linha reta. Nesse caso, é possível mapear os dados para um espaço de maior dimensão usando o que é chamado de "função kernel". Essa função transforma os dados para um espaço onde a separação pode ser realizada por uma linha reta.

No caso do trabalho, como o conjunto de dados é pequeno, foi utilizado o método LinearSVC do scikit learn, que utiliza uma função kernel linear que busca traçar uma linha reta para separar as classes.

Os hiperparâmetros utilizados no LinearSVC foram os seguintes:

C: controla a penalidade para erros e a margem. Maior C = menos margem, mais penalidade para erros. Menor C = mais margem, menos penalidade para erros.

Kernel: Define a função kernel usada para mapear dados em um espaço dimensional superior. Exemplos comuns incluem linear, polinomial e RBF.

gamma: parâmetro para o kernel RBF, controla a complexidade da função de decisão. Maior gamma = função de decisão mais complexa.

grau: parâmetro polinomial do kernel, define o polinômio de grau usado para mapear os dados.

coef0: parâmetro para os kernels polinomial e sigmoid, controla a influência da parte constante na função de decisão

Vamos a determinação dos melhores HP's:

```

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(x, y, test_size

from sklearn.model_selection import GridSearchCV
from sklearn.svm import LinearSVC

param_grid = {'C': [0.1, 1, 10],
               'tol': [1e-4, 1e-3, 1e-2],
               'max_iter': [100, 300, 500, 1000],
               'class_weight': [None, 'balanced']}

```

```

model = LinearSVC()

grid_search = GridSearchCV(model, param_grid=param_grid, cv=5, n_jobs=-1)
grid_search.fit(X_train, y_train)

print('Melhores hiperparâmetros:', grid_search.best_params_)
print('Melhor score:', grid_search.best_score_)
>>>Melhores hiperparâmetros: {'C': 10, 'class_weight': 'balanced', 'max_iter': 1000}
>>>Melhor score: 0.8666666666666668

```

Abaixo, uma execução com esses hiperparâmetros

```

### SUPPORT VECTOR MACHINE
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.2)
from sklearn.svm import LinearSVC

classifier = LinearSVC(C= 1, class_weight = 'balanced', max_iter= 1000, tol= 1e-5)
classifier.fit(X_train, y_train)

# Evaluate classifier on testing data
y_pred = classifier.predict(X_test)
from sklearn.metrics import accuracy_score, precision_score, recall_score
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
print(f"Linear SVC >>>>> Accuracy: {accuracy:.2f}, Precision: {precision:.2f}, Recall: {recall:.2f}, F1-score: {f1:.2f}")

>>> Linear SVC >>>>> Accuracy: 0.80, Precision: 0.88, Recall: 0.64, F1-score: 0.75

```

4. Métricas de classificação

4.1. Stratified K fold cross validation

É uma técnica usada para avaliar o desempenho de um modelo em um conjunto de dados.

Na validação cruzada k-fold, é dividido os dados em "folds", e o modelo é treinado e avaliado k (número de folds) vezes, cada vez usando um fold diferente para validação e os folds K-1 restantes para treinamento.

Essa técnica permite uma avaliação satisfatória do dataset em um geral, pois permite de certa utiliza-lo ao seu todo e não só uma parte em cada treinamento e teste.

Na validação cruzada k-fold estratificada, os dados são divididos em k folds de forma que cada fold tenha o mesmo número de amostras de cada classe. Através disso o modelo consegue ser avaliado em uma amostra representativa dos dados e pode ajudar a evitar problemas como overfitting ou underfitting que podem surgir quando os dados

não estão balanceados, como por exemplo, pegar uma parte do dataset que a classe é quase sempre B e treinar em cima de outro que tenha só valores de classe M, gerando underfitting.

O desempenho desse método de avaliação é percebido quando se é retornado numa lista de acurácias, que são os resultados desta medição em cada fold.

Pseudo algoritmo utilizado na implementação:

1. Crie uma matriz de índices que correspondem às linhas de X e y

É criada uma lista de índices que corresponde a cada linha dos dados dos atributos X, isso é feito para que os dados sejam randomizados.

2. Embaralhar/randomizar os índices aleatoriamente.

Os índices das linhas serão embaralhados com o intuito de garantir a aleatoriedade dos folds

3. Randomiza-se os dados X e os classe y de acordo com os índices embaralhados.

Como dito no item anterior, existe a necessidade de se embaralhar os folds para que exista uma certa aleatoriedade em cada um, evitando que por exemplo, um fold "j" seja um fold representativo de certa parte do dataset, podendo gerar bias/viés sobre aquela parte dos dados.

4. Calcule o tamanho de cada fold como o número total de amostras dividido pelo número de folds k.

Isso é feito dividindo por k e garantido que o tamanho do fold seja inteiro para não ter problemas de índice do tipo float. Exemplo: número de instância = 100, k= 5 , tamanho dos folds = 20.

5. Iniciar uma lista de precisão que vai armazenar o valor da precisão de cada fold

6. Para cada fold i no intervalo k:

a. Calcule os índices inicial e final da fold de teste.

Calcula de onde até onde no dataset vai o fold de teste, sendo os limites de início o índice do fold atual e o fim calculando com base no tamanho do fold em questão

b. Selecione os atributos e classe para a fold de teste.

Com base nos limites do item anterior, seleciona os dados de teste.

c. Selecione os atributos e classe restantes como dados e classe de treinamento.

O que sobrou da seleção do item anterior é treinamento

d. Divida os índices das amostras de treinamento em matrizes separadas para cada valor da classe.

Isso é feito para embaralhar esses índices para garantir a randomicidade dos folds, por exemplo, se y = [0,0,1,0,1,0,1,1] vai dividir em 2 arrays: [0,1,3,5] e [2,4,6,7] o pri-

meiro indicando os índices da ocorrência 0 e o segundo da ocorrência 1

- e. **Embaralhe os índices de cada label de classe aleatoriamente.**
 - f. **Divida os índices embaralhados de cada classe em k folds de tamanho igual.**
 - g. **Combine as folds correspondentes de cada classe para criar os índices de treinamento para cada fold.** Dos passos anteriores, temos k folds de tamanhos iguais e randomizados para cada valor da classe
 - h. **Treinar o modelo nos dados e classe de treinamento usando os índices de treinamento selecionados.**
 - i. **Avaliar o modelo treinado nos dados e classe de teste e calcular a acurácia.**
 - j. **Acrescente a acurácia da fold atual à lista de acurácias.**
- Retorna a lista de acurácias para cada fold.**

A implementação do pseudo-código esta descrita a seguir:

```
from sklearn.metrics import accuracy_score

def stratified_k_fold_cross_validation(X, y, model, k=5):
    X = np.array(X)
    y = np.array(y)
    indices = np.arange(X.shape[0])
    np.random.shuffle(indices)
    X = X[indices]
    y = y[indices]
    fold_size = X.shape[0] // k
    #if number of samples = 20 and K = 5 => 20
    accuracy = []
    for i in range(k):
        start_fold = i * fold_size

        end_fold = (i + 1) * fold_size
        #split test in 20:40,49:60, etc
        X_test = X[start_fold:end_fold]
        y_test = y[start_fold:end_fold]
        #split test in what is remaining
        X_train = np.concatenate((X[:start_fold], X[end_fold:]))
        y_train = np.concatenate((y[:start_fold], y[end_fold:]))

        # index of each class
        class_indices = []
        #print("y train: ")
        #print(y_train)
        for val in np.unique(y_train):
            class_indices.append(np.where(y_train == val)[0])
        #print("indices: ")
        #print(class_indices)
```

```

# randomize the indexes
for indices in class_indices:
    np.random.shuffle(indices)

# split classes in k folds of equal size
class_folds = [np.array_split(indices, k) for indices in class_...
#print("class folds:")
#print(class_folds)

# Combine the folds to create the training indices for each fold
train_indices = []
for i in range(k):
    fold_indices = [indices[i] for indices in class_folds]
    train_indices.append(np.concatenate(fold_indices))
train_indices = np.concatenate(train_indices)
#print(train_indices)

# Train
model.fit(X_train[train_indices], y_train[train_indices])
#Evaluate
y_pred = model.predict(X_test)
accuracy.append(accuracy_score(y_test, y_pred))

return accuracy

```

4.2. Matriz de confusão

Uma matriz de confusão é uma ferramenta usada em aprendizado de máquina para avaliar a precisão de um modelo de classificação. Basicamente, mostra quantas vezes o modelo acertou ou errou ao classificar os dados em diferentes categorias.

A matriz é dividida em quatro quadrantes: verdadeiros positivos (VP), falsos positivos (FP), falsos negativos (FN) e verdadeiros negativos (VN). VP são momentos em que o modelo classificou corretamente os dados como pertencentes a uma determinada classe. FP é quando o modelo classificou os dados em uma classe, mas na verdade eles pertenciam a outra classe. FN são os momentos em que o modelo não classificou os dados como não pertencentes a uma determinada classe. E VN é quando o modelo classificou corretamente os dados como não pertencentes a uma determinada classe.

A implementação da matriz está descrita a seguir, onde são comparados os valores "atuais" e preditos de cada classe.

```

def calculate_matrix(self) -> np.array:
    classes = np.unique(self.actual)

```

```

matrix = np.zeros((len(classes), len(classes)), dtype=int)

for i, j in enumerate(classes):
    matrix[i] = np.bincount(self.predicted[self.actual == j], m

return matrix

```

As métricas utilizadas na matriz de confusão serão desfritas a seguir:

Acurácia: proporção de previsões corretas do modelo em relação ao número total de previsões. Acurácia indica que o modelo tem um bom desempenho na classificação de dados gerais. Baixa acurácia pode indicar que o modelo tem dificuldade em classificar os dados ou que as classes são muito desbalanceadas.

```

def calculate_accuracy(self) -> float:
    """accuracy = (TP + TN) / (TP + TN + FP + FN)"""
    all_sum = np.sum(self.matrix)
    accuracy = (self.true_positive + self.true_negative) / all_sum
    return accuracy

```

Precisão: proporção de verdadeiros positivos em relação ao total de previsões positivas do modelo. Indica que o modelo é capaz de identificar verdadeiros positivos e reduzir falsos positivos. Baixa precisão pode significar que o modelo está produzindo muitos falsos positivos e precisa ser ajustado.

```

def calculate_precision(self) -> float:
    """ precision = tp / tp+fp """
    precision = self.true_positive / (self.true_positive + self.false_posi
    return precision

```

Recall: Um recall alto indica que o modelo é capaz de detectar verdadeiros positivos e minimizar falsos negativos. Retornos baixos podem significar que o modelo carece de muitos benefícios reais e precisa de ajustes. É proporção de verdadeiros positivos em relação ao total de verdadeiros positivos e falsos negativos.

```

def calculate_recall(self) -> float:
    """recall = TP / (TP + FN)"""
    recall = self.true_positive / (self.true_positive + self.false_posi
    return recall

```

F1-score: Média harmônica da precisão e do recall. Um F1-score alto indica que o modelo está tendo um bom desempenho tanto na precisão quanto no recall. Um F1-score baixo pode indicar que o modelo precisa ser ajustado para melhorar sua capacidade de identificar verdadeiros positivos e minimizar falsos positivos.

```

def calculate_f1_score(self) -> float:
    """F1 = 2 * (precision * recall) / (precision + recall)"""

```

```
f1_score = 2 * (self.precision_val * self.recall_val) / (self.p  
return f1_score
```

5. Resultados e conclusão

Os resultados após escolha dos hiperparâmetros e implementação dos algoritmos, junto aos métodos de avaliação apresentados, foram os seguintes:

```
- - - - knn - - - -  
matrix KNN:  
[[12  0]  
 [ 2  6]]  
acc KNN:  
0.9  
precision KNN:  
1.0  
recall KNN:  
0.75  
f1 KNN:  
0.8571428571428571  
stratified k fold cross validation =  
[0.85, 0.8, 0.8, 0.7, 0.8]  
  
- - - - Random forest - - - -  
matrix RF:  
[[12  0]  
 [ 2  6]]  
acc RF:  
0.9  
precision RF:  
1.0  
recall RF:  
0.75  
f1 RF:  
0.8571428571428571  
stratified k fold cross validation =  
[0.85, 0.9, 0.85, 0.85, 0.85]
```

Figura 8. Resultados


```

- - - - naive bayes - - - -
matrix naive bayes:
[[7 5]
 [1 7]]
acc naive bayes:
0.7
precision naive bayes:
0.5833333333333334
recall naive bayes:
0.875
f1 naive bayes:
0.7000000000000001
stratified k fold cross validation =
[0.8, 0.7, 0.75, 0.75, 0.65]

- - - - SVN - - - -
matrix SVN:
[[10 1]
 [ 2 7]]
acc SVN:
0.85
precision SVN:
0.875
recall SVN:
0.7777777777777778
f1 SVN:
0.823529411764706
stratified k fold cross validation =
[0.7, 0.95, 0.8, 0.85, 0.75]

```

Figura 9. Resultados

Sobre a matriz de confusão, analisando os resultados, pode-se perceber que o desempenho dos modelos é semelhante, exceto o Naive Bayes, que apresentou acurácia inferior aos demais modelos, e o SVN, que apresentou acurácia um pouco menor, porém melhor desempenho na medida de precisão. Na matriz de confusão, podemos ver que os resultados dos modelos KNN e Random Forest foram semelhantes, capturando todos os casos negativos e produzindo apenas 2 falsos negativos cada. Naive Bayes deu 5 falsos negativos e SVN deu 1 falso negativo e 1 falso positivo. Observando as métricas de precisão e recall, vemos que Naive Bayes teve uma alta taxa de recall, mas baixa precisão, indicando que o modelo detectou a maioria dos casos positivos, mas também classificou muitos casos negativos como positivos. Por outro lado, o SVN teve uma precisão relativamente alta, mas uma recall ligeiramente inferior, indicando que o modelo identificou corretamente muitos casos negativos, mas perdeu alguns casos positivos. Isso ocorre provavelmente por Naive Bayes ser um modelo probabilístico que assume que os atributos

são independentes uns dos outros. Isso pode não ser verdade em todos os casos, o que pode levar a baixa precisão porque o modelo pode considerar atributos irrelevantes como preditores positivos. Já o SVN é um modelo que tenta encontrar o hiperplano que melhor separa as classes no espaço de propriedades. Isso é particularmente útil se as classes forem linearmente separáveis, o que aparenta ser o caso desse dataset. Caso contrário, as classes podem se sobrepor, o que pode resultar em menor recall, pois alguns casos positivos podem ser erroneamente classificados como negativos.

A validação cruzada estratificada foi usada para avaliar o desempenho dos modelos em diferentes conjuntos de dados. Podemos ver que a acurácia média para cada modelo é relativamente consistente em todos folds, menos para o SVN, que apresentou uma variação maior nos resultados. Para SVN, pode ser que a distribuição de classes em alguns folds tenha sido mais difícil para o modelo aprender, levando a um desempenho reduzido. Por outro lado, em outros folds, o modelo poderia aprender melhor a distribuição das classes, levando a melhores resultados. Isso pode ter ocasionado maiores diferenças nos resultados das acurácias dos folds em relação aos demais modelos.

Quanto as medidas do Random Forest e KNN, elas são bastante semelhantes, ambos apresentando uma acurácia, precisão, recall e F1 score bons. Além disso, ambos modelos tiveram bom desempenho na matriz de confusão, capturando a maioria dos casos positivos e negativos e produzindo poucos falsos positivos e falsos negativos. Isso sugere que os modelos foram capazes de generalizar bem os dados e identificar corretamente a maioria dos casos positivos. Isso ocorre muito provavelmente pelos dados estarem bem estruturados no caso do KNN, conseguindo dividir bem as classes, enquanto o random forest conseguiu combinar bem as árvores da floresta nesse dataset.

6. Referências

<https://towardsdatascience.com/stratified-k-fold-cross-validation-on-grouped-datasets-b3bca8f0f53e>

<https://www.geeksforgeeks.org/stratified-k-fold-cross-validation/>

<https://www.researchgate.net/figure/Pseudo-code-for-nested-k-fold-cross-validation-The-inner-loop-performs-cross-validation-fig4-331292817>

<https://www.projectpro.io/recipes/explain-stratified-k-fold-cross-validation>

<https://en.wikipedia.org/wiki/Confusion-matrix>

<https://towardsdatascience.com/understanding-confusion-matrix-a9ad42dcfd62>

<http://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusionmatrix.html>

<https://www.inf.ufpr.br/dagoncalves/IA07.pdf>

<https://pt.wikipedia.org/wiki/M>