Relatório da atividade K-fold

INF01017
Lucas Marques Dorneles - 00291329
Henrique da Silva Barboza - 00272730

Introdução ao problema

Pegamos como objetivo para este trabalho a tarefa de verificar se cogumelos são comestíveis ou venenosos via aprendizado de máquina em cima das características da aparência dos cogumelos. Utilizamos o *dataset* “mushroom” disponível em https://github.com/EpistasisLab/pmlb/tree/master/datasets/mushroom, e desenvolvemos o código usando Python 3.0 com Jupyter Notebooks, utilizando das bibliotecas *sklearn* e *pandas* para desenvolver o código, e *seaborn* para gerar os gráficos.

O *dataset* possui 8.124 instâncias, e 23 *features* ao todo, incluindo a *target*. A descrição de todas as *features* coletadas dos cogumelos pode ser vista no arquivo *metadata.yaml* dentro do repositório acima, mas em geral todas as *features* são categóricas. Existem neste *dataset features* categóricas com mais de dois valores possíveis, e neste *dataset* elas já vêm encodificadas de maneira ordinal. Elas variam desde o formato da cabeça do cogumelo (*cap-shape*), que pode ser em formato de sino (valor 0), cônico (1), convexo (2), achatado (3), arqueado (4), ou afundado (5), como o formato do caule, a cor do caule acima e abaixo dos anéis, se o cogumelo tem “machucados”, o habitat do cogumelo, e a quantidade de cogumelos encontrados em média juntos um lugar. Das instâncias do *dataset*, 3.916 representam cogumelos venenosos e 4.212 representam cogumelos comestíveis, assim havendo um balanço perto de 50% entre a representação das classes positiva e negativa.

Existem *features* binárias e com múltiplos valores possíveis, mas todas são originalmente categóricas, e não há dados faltando para nenhuma das instâncias. Todas as *features* foram encodificadas em numéricas via encodificador ordinal. A partir destas features, o objetivo é determinar se o cogumelo é comestível (valor *target* 0, considerado positivo), ou se é venenoso (valor *target* 1, considerado negativo).

Metodologia do problema

Para resolver este problema, decidimos escolher como os três algoritmos as técnicas de K-vizinhos mais próximos (KNN), florestas randomizadas (RFC), e regressão linear (LR). Escolhemos estes algoritmos por representarem um *spread* interessante de técnicas com bases teóricas diferentes; KNN é um algoritmo simples, rápido e eficiente, mas tende a ter poder preditivo menor para problemas complexos; RFC, por ser baseado em árvores de decisão, carrega as forças e fraquezas deste método além de ter um poder maior de distinguir relações entre *features* e resultados devido a escolha aleatória das *features* usadas; e regressão linear trabalha com uma fundamentação matemática de combinação linear que garante um bom poder preditivo e é fundamentalmente diferente das duas outras técnicas.

Para pré-processamento dos dados, deixamos eles exatamente como estavam. Percebemos isto apenas após rodar os testes e analisar os resultados, mas como o *dataset* já vinha convertido de maneira ordinal, não tínhamos como aplicar o algoritmo de *one hot encoding* para se adaptar ao LR, e achamos que seria interessante ver o impacto real de usar este tipo de conversão com um algoritmo que é sensitivo aos valores específicos usados nas *features* em contraste com KNN e RFC que não sofrem deste problema. Como as classes estão balanceadas de início, também não utilizamos nenhuma estratégia de *downsampling*, *upsampling* ou geração sintética de instâncias.

Para a configuração do hiperparâmetro K do KNN, rodamos a tarefa de classificação com diferentes configurações de K (3, 5, 9, 13, 27, 51) e verificamos que a versão com melhor desempenho em relação à precisão tendia a ser com K=9, e então escolhemos esta configuração para representar o KNN. Escolhemos precisão para essa análise pois o peso de um falso positivo é potencialmente letal, já que falso positivo representa um cogumelo sendo classificado como comestível quando na realidade é venenoso.

Para regressão linear, como está sendo usada para uma tarefa de classificação, precisamos definir uma função de ativação para ela. Utilizamos como função de ativação um simples cheque: como o *target* está entre 0 e 1, se o valor final for maior ou igual a 0.5, a instância será classificada como venenosa (função de ativação dispara), e caso abaixo será classificada como comestível (função de ativação não dispara). Usamos esta função de ativação pela simplicidade da implementação e porque acreditamos que seria desnecessário um algoritmo de ativação mais complexo para esta tarefa de classificação.

Metodologia e código

Não desenvolvemos o código como uma função; como estamos trabalhando com Jupyter Notebooks, ao invés disso fizemos um template no qual substituímos o algoritmo sendo utilizado em cada célula de cada algoritmo.

```python
#Create K folds
K = 5
number_of_entries = data.shape[0]
number_of_entries_edible = data[data['target'] == 0].shape[0]
number_of_entries_poisonous = data[data['target'] == 1].shape[0]

estrat = number_of_entries_poisonous / number_of_entries

instancias_por_fold = number_of_entries / K
instancias_por_fold_poison = math.floor(number_of_entries_poisonous / K)
instancias_por_fold_edible = math.floor(number_of_entries_edible / K)

fold_dataframes = []

for i in range(1,K):
    #Sample a number of poisonous and edible instances that matches the stratification of the dataset
    fold_df_poison = data[data['target'] == 1].sample(instancias_por_fold_poison)
    fold_df_edible = data[data['target'] == 0].sample(instancias_por_fold_edible)
    fold_df = pd.concat([fold_df_poison, fold_df_edible])
    #Drop these instances from the source table so there may be no duplicates in each fold
    data = data.drop(fold_df.index)
    fold_dataframes.append(fold_df)

#The last fold will inherit the rest of the instances in the dataset after the other K-1 folds have been created.
#This is so that we may easily handle instances where the dataset used is not perfectly divisible by the number of folds.
#The last fold will have more instances than the others in the instance this is true.
fold_dataframes.append(data)
```

Figura 1: Criação de partições. Cada *fold* é um dataset diferente, pegando parte dos dados do *dataset* original.

```python
KNN = KNeighborsClassifier(n_neighbors=51)
KNN_metrics = []
KKN_metrics_calculated = []

for i in range(0, K):
    train_data = {}

    #Cria dataset com todos os folds menos o i-ésimo, que será usado para teste
    train_data = pd.concat(fold_dataframes)
    train_data.drop(fold_dataframes[i].index)

    #Pega dados de teste, remove a coluna target
    features = train_data.columns[1:-1]
    X = train_data.loc[:, features]
    y = train_data.target

    #Faz o fitting do algoritmo
    KNN.fit(X, y)

    #Testa com dados de teste
    test_data = fold_dataframes[i].loc[:, features]
    predictions = KNN.predict(test_data)
    truth_val = fold_dataframes[i].loc[:, 'target'].tolist()

```

Figura 2: Gera-se os dados de treinamento usando todas as partições e depois removendo as instâncias da partição a ser usada para testes nesta iteração. Após isso, faz-se o fit do modelo e prepara os dados de teste para a avaliação do modelo nesta iteração.

```
    vp = 0
    fp = 0
    vn = 0
    fn = 0
    #Contabiliza verdadeiros positivos, verdadeiros negativos, falsos positivos e falsos negativos
    for j in range(0, test_data.shape[0]):
        if predictions[j] == 0 and predictions[j] == truth_val[j]:
            vp += 1
        elif predictions[j] == 0 and predictions[j] != truth_val[j]:
            fp += 1
        elif predictions[j] == 1 and predictions[j] == truth_val[j]:
            vn += 1
        else:
            fn += 1

    results = [vp, fp, vn, fn]
    KNN_metric = {}
    KNN_metric["acc"] = (vp + vn) / (vp + vn + fp + fn)
    KNN_metric["sens"] = (vp) / (vp + fn)
    KNN_metric["prec"] = (vp) / (vp+fp)

    β = 0.5
    KNN_metric["f1"] = (1+β*β)*KNN_metric["prec"]*KNN_metric["sens"]/(β*β * KNN_metric["prec"] + KNN_metric["sens"])

    KKN_metrics_calculated.append(KNN_metric)
    display(results)
    KNN_metrics.append(results)
```

Figura 3: Mensura-se o número de verdadeiros positivos (vp), verdadeiros negativos (vn), falsos positivos (fp) e falsos negativos (fn) e calculam-se as métricas. Nota-se que “verdadeiro” está sendo considerado uma instância que é comestível e tem valor *target* 0.

```
#Create K folds
K = 5
number_of_entries = data.shape[0]
number_of_entries_edible = data[data['target'] == 0].shape[0]
number_of_entries_poisonous = data[data['target'] == 1].shape[0]

estrat = number_of_entries_poisonous / number_of_entries

instancias_por_fold = number_of_entries / K
instancias_por_fold_poison = math.floor(number_of_entries_poisonous / K)
instancias_por_fold_edible = math.floor(number_of_entries_edible / K)

fold_dataframes = []

for i in range(1,K):
    #Sample a number of poisonous and edible instances that matches the
stratification of the dataset
    fold_df_poison = sample_poisonous(data, instancias_por_fold_poison)
    fold_df_edible = sample_edible(data, instancias_por_fold_edible)
    fold_df = join_dataframes(fold_df_poison, fold_df_edible )
    #Drop these instances from the source table so there may be no
duplicates in each fold
    fold_dataframes.append(fold_df)

#The last fold will inherit the rest of the instances in the dataset
after the other K-1 folds have been created.
#This is so that we may easily handle instances where the dataset used
is not perfectly divisible by the number of folds.
#The last fold will have more instances than the others in the instance
this is true.
fold_dataframes.append(data)
```

Figura 4: Pseudocódigo da criação dos *folds*

```
#Aqui fica o exemplo do KNN, mas a lógica é a mesma para todos os
algoritmos; muda apenas o nome do algoritmo sendo utilizado
KNN = KNeighborsClassifier(n_neighbors=9)

for i in range(0, K):
    train_data = remove_fold(i, fold_dataframes)

    #Pega dados de teste, remove a coluna target
    X = get_data_without_target(train_data)
    y = get_target_values(train_data)

    #Faz o fitting do algoritmo
    KNN.fit(X, y)

    #Testa com dados de teste
    test_data = extract_only_fold(i, fold_dataframes)
    predictions = KNN.predict(test_data)
    truth_val = get_target_values(i, fold_dataframes)

    vp = 0; fp = 0; vn = 0; fn = 0
    #Contabiliza verdadeiros positivos, verdadeiros negativos, falsos
positivos e falsos negativos
    for j in range(0, test_data.shape[0]):
        if prediction was edible and was correct:
            vp += 1
        elif prediction was edible and was incorrect:
            fp += 1
        elif prediction was poisonous and was correct:
            vn += 1
        elif prediction was poisonous and was incorrect:
            fn += 1

    results = [vp, fp, vn, fn]
    β = 0.5
    #Calcula acurácia, precisão, sensibilidade e F1-measure e retorna
eles dentro de um dicionário
    KNN_metric = calculate_metrics(results, β)

    KKN_metrics_calculated.append(KNN_metric)
```

Figura 5: Pseudocódigo da aplicação do algoritmo KNN. O desvio padrão e as médias são calculadas posteriormente com base em KNN_metrics_calculated

O algoritmo KNN implementado funciona em duas partes. Primeiro criamos os *folds*/partições, e depois usamos estes *folds* para treinar e avaliar os algoritmos. Quando avaliando o algoritmo, o i-ésimo *fold* designa a partição que será usada para testar o desempenho enquanto o resto dos *folds* serão usados para o treinamento. O treinamento é feito pelas próprias funções do *sklearn*, no exemplo acima representado por *KNN.fit* e *KNN.predict*, que treinam o modelo e predizem os resultados do conjunto de teste respectivamente. Após a tabulação dos falso positivos, falso negativos, verdadeiro positivos e verdadeiro negativos, se calcula as métricas de acurácia, precisão, sensibilidade/*recall* e *F1-measure* para esta escolha de dados de treinamento e teste. O cálculo das médias e do desvio padrão é feito posteriormente quando os gráficos forem gerados, mas como *KKN_metrics_calculated* é uma lista de dicionários, o cálculo das médias e desvio dados como resultado do *K-fold* são triviais.

Resultados e análise

Seguem o desempenho médio dos algoritmos segundo as métricas de acurácia, *recall*, precisão e F1, assim como o desvio padrão deles.

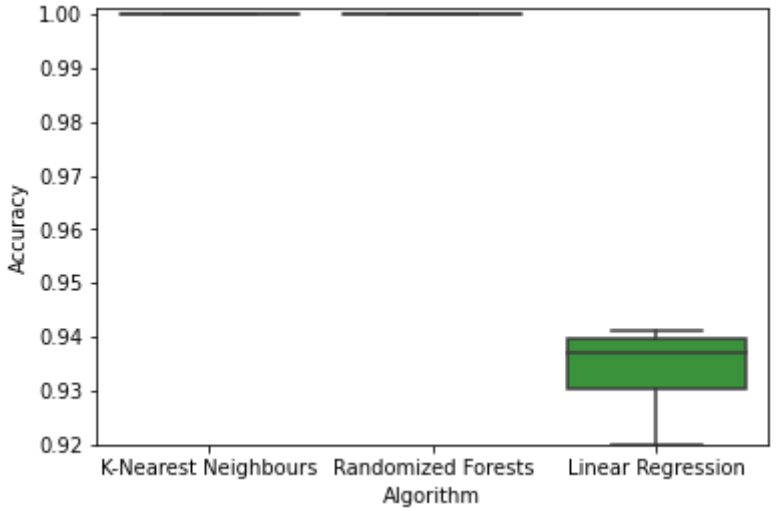

Figura 6: Desempenho médio dos algoritmos em relação à acurácia.

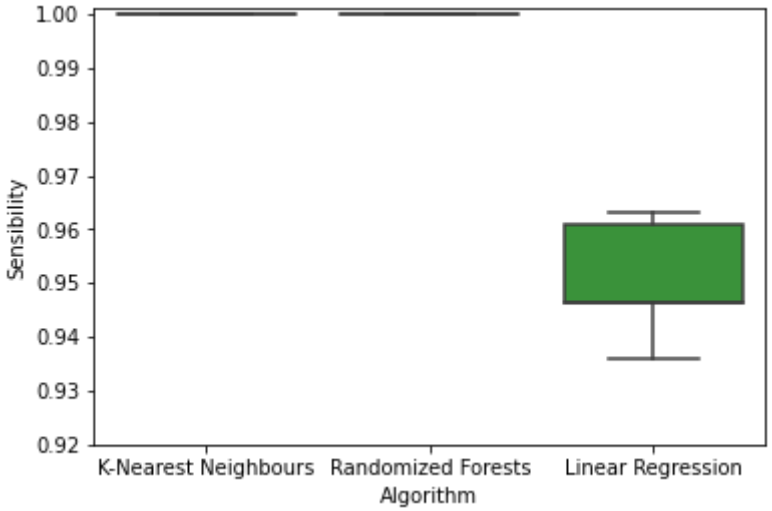

Figura 7: Desempenho médio dos algoritmos em relação à sensibilidade

.

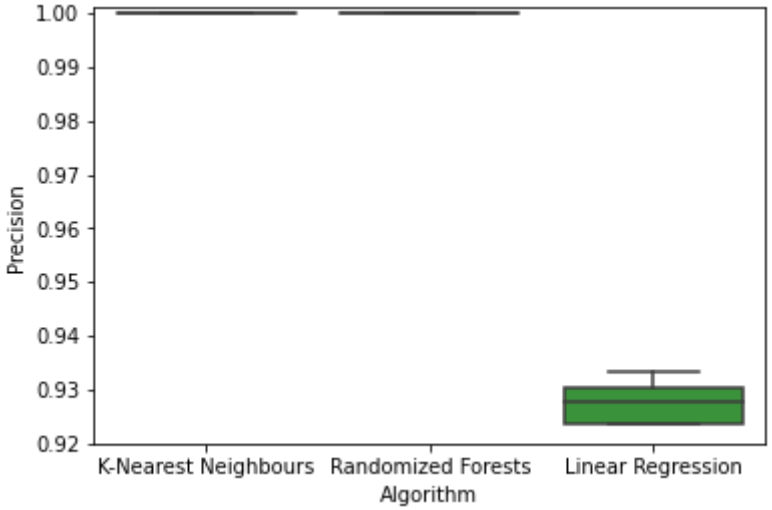

Figura 8: Desempenho médio dos algoritmos em relação à precisão.

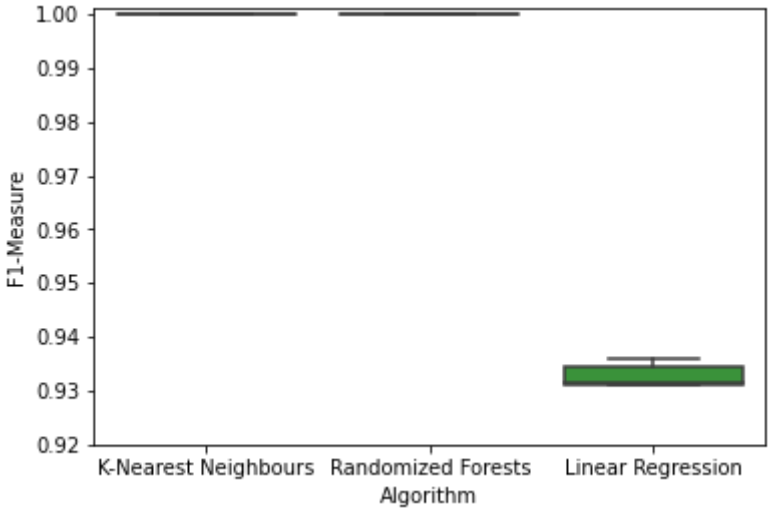

Figura 9: Desempenho médio dos algoritmos em relação à *F1-measure*.

Algorithm	Accuracy	Sensibility	Precision	F1-Measure
K-Nearest Neighbours	0.000000	0.000000	0.000000	0.000000
Randomized Forests	0.000000	0.000000	0.000000	0.000000
Linear Regression	0.007762	0.010163	0.010563	0.008733

Figura 10: Desvio padrão dos algoritmos em relação às métricas medidas.

Como podemos ver, KNN e RFC tiveram um desempenho perfeito, conseguindo prever com 100% de acurácia todas as instâncias de teste em todos os *folds*, enquanto LR teve um desempenho na casa dos 90% em todas as métricas. Vale relembrar que os positivos se referem a cogumelos comestíveis e os negativos aos venenosos. Usando a precisão para fazer uma análise, pois eliminar falsos positivos seria de extrema importância quando o falso positivo se referiria a um cogumelo venenoso sendo identificado como comestível, KNN e RFC tem desempenho perfeito enquanto LR beira aos 93% de precisão.

Para uma aplicação no mundo real, mesmo 93% de precisão sendo alto, ainda proveria uma taxa de risco alta quando a falha de identificação de um cogumelo venenoso teria consequências potencialmente letais. Assim, o LR não poderia ser usado para definitivamente dizer se um cogumelo é ou não venenoso, mas sua performance acima de 90% indica que poderia ser utilizado como uma ferramenta dentro de um suíte de ferramentas para indicar se é mais provável que seja venenoso do que comestível. Enquanto isso, os modelos aprendidos por KNN e RFC expressam mais confiança sobre seus resultados, e poderiam ser confiados para dar respostas sobre a comestibilidade de um cogumelo sem o auxílio de outros algoritmos.

Conclusão

Analisando a razão por trás das performances perfeitas de KNN e RFC, nós chegamos a conclusão que o próprio *dataset* é expressivo o suficiente com as *features* escolhidas que até algoritmos relativamente simples como o KNN conseguem aprender as relações entre as *features* e as classes alvo.

Inicialmente achamos curioso que o LR, mesmo sendo um algoritmo com poder preditivo mais robusto que o KNN, obteve desempenho pior que ele. Olhamos para o *dataset* original e as características do algoritmo de recursão linear, e percebemos que o *dataset* fez a conversão de *features* categóricas para numéricas de maneira ordinal, botando ordem entre o valor de *features* que não tem relacionamento de ordem na sua origem. Como o LR é sensitivo ao valor da *feature* em si para os seus cálculos internos, imaginamos que ele estava aprendendo relações entre os valores das *features* que não existem na realidade. Isso fazia com que ele obtivesse performance pior que KNN e RFC, já que esses algoritmos não são sensíveis a relação de ordem nos valores das *features*. Pensamos em tentar rodar o *one hot encoder* em cima do dataset original e ver como isso ajudava o LR, mas como os dados já vinham pré-encodificados ordinalmente, não conseguimos fazer o encodificador fazer a conversão.

Assim, RFC e KNN apresentam performances perfeitas ao problema descrito por esse *dataset*, e o LR apresenta uma performance boa, mas não boa o suficiente para atingir níveis aceitáveis de confiabilidade como algoritmo de detecção de cogumelos comestíveis e venenosos. Gostaríamos de ter conseguido aplicar o LR com *one hot encoding* para ver como que ele agiria com dados apropriados ao seu funcionamento, mas foi interessante de qualquer maneira ver como o pré-processamento pode afetar os resultados do algoritmo, e especialmente ver como a conversão ordinal de dados categóricos pode diminuir a performance do LR. No geral, achamos que foi uma experiência de aprendizado importante utilizar esses algoritmos por nós mesmos e analisar seus resultados.