Relatório TP2 - Introdução à Inteligência Artificial

Introdução

O trabalho consiste na implementação de dois algoritmos: K-nearest neighbors (Knn), de aprendizado supervisionado e K-means, de aprendizado não supervisionado. Esses algoritmos foram implementados "*from scratch"* e serão analisados em uma próxima seção. Além disso, em visão dos pontos extras oferecidos, foram também utilizados os algoritmos Knn e K-means da biblioteca *sklearn*, e vamos também realizar uma análise nos dois e compará-los com os que foram implementados.

Desenvolvimento

- **Divisão**
  O trabalho foi implementado em 4 arquivos diferentes:
  - **Knn.py** - contém as análises e execução do algoritmo Knn
  - **KMeans.py** - igual o acima, só que para o KMeans
  - **Algorithms.py -** Onde a implementação dos métodos desses algoritmos foi implementada
  - **Metrics.py -** Implementa as métricas usadas na análise

Além disso, temos os arquivos **Extra_Knn.py** e **Extra_KMeans,** que importam as funções da biblioteca ***sklearn*** para realizar as análises referentes aos pontos extras.

- **Implementação dos Algoritmos**

  Primeiro foi implementada a função ***model(),*** que com relação a esse trabalho apenas passa pelos .csv oferecidos e salva os dados em uma matriz. No caso do Knn, ele recebe apenas seus dados de "treino"; No caso do K-Means, todos os dados são passados como entrada.

  A lógica do Knn está na função ***Knn_predict(),*** que recebe como entrada o modelo, o valor de "k" os dados de teste, ou seja, os dados que serão classificados. Ele compara cada ponto de teste, através de uma função ***DistanciaEuclidiana(),*** e rotula o ponto de teste com a mesma label do ponto do modelo de menor distância dele. O algoritmo retorna um vetor de labels, que é então comparado com as labels reais dos pontos de teste com as funções de métrica: ***Precision(),Accuracy(),Recall(),F1_score().*** No arquivo **Knn.py** são feitas essas análises e depois a matriz de confusão é impressa na tela.

No caso do KMeans é semelhante. A função **KMeans()** recebe o número máximo de iterações, o valor de “k”, e o modelo de dados, que nesse caso é apenas uma matriz contendo todos os dados de entrada. Inicialmente ele escolhe randomicamente k-centróides. Após isso, ele agrupa os demais pontos em relação a esses centróides da seguinte forma: O ponto P pertence ao cluster “k” caso ele tenha menor distância euclidiana para o k-centróides do que para os outros centróides. Para finalizar uma iteração, o algoritmo tira a média dos valores de todos os pontos em um mesmo *cluster* e escolhe, assim, novos centróides. O algoritmo acaba quando o número de iterações máximo é atingido ou quando os centróides não se alteram de uma iteração para outra. A função retorna os centróides e os clusters. Os centróides são, então, impressos na tela, além de uma análise de cada cluster.

Análise

Vamos fazer agora uma análise dos resultados obtidos, começando pelo algoritmo Knn.

- **Knn com k = 2**
  - **Acurácia: 0.6268656716417911**
  - **Precisão: 0.6574074074074074**
  - **Revocação: 0.8452380952380952**
  - **F1-score: 0.7395833333333334**

- **Knn com k = 10**
  - **Acurácia: 0.667910447761194**
  - **Precisão: 0.711229946524064**
  - **Revocação: 0.7916666666666666**
  - **F1-score: 0.7492957746478872**

- **Knn com k = 25**
  - **Acurácia: 0.6604477611940298**
  - **Precisão: 0.7225433526011561**
  - **Revocação: 0.744047619047619**
  - **F1-score: 0.7331378299120235**

- **Knn com k = 50**
  - **Acurácia: 0.6865671641791045**
  - **Precisão: 0.7359550561797753**
  - **Revocação: 0.7797619047619048**
  - **F1-score: 0.7572254335260116**

Como podemos ver, a acurácia se mantém basicamente constante, aumentando muito pouco. Porém, vemos variações maiores tanto em revocação quanto em precisão. A revocação diminui com o aumento de k, e a precisão tende a aumentar. E ainda mais, se olharmos para a F1-Score, podemos ver que ela atinge seu maior valor com k = 50. Então, embora com valores maiores de k nem a precisão nem a revocação estejam em seus valores mais altos, a média harmônica entre os dois é de fato maior, o que mostra que as duas estão num balanço de valores melhor para esse valor de k.

Agora, veremos o caso do ***KMeans*****.** Vale mencionar que o número de iterações máximas = 1000. Números mais altos poderiam trazer resultados possivelmente melhores, mas o algoritmo já demora cerca de 1 minuto para rodar com esse parâmetro, então foi escolhido deixar dessa forma. Além disso, como os centróides são iniciados de forma randômica, um pouco de diferença é esperado entre cada execução.

- **KMeans com k = 2**
  - **Cluster 0: Label 0 = 370 || Label 1 = 299**
  - **Cluster 1: Label 0 = 139 || Label 1 = 532**

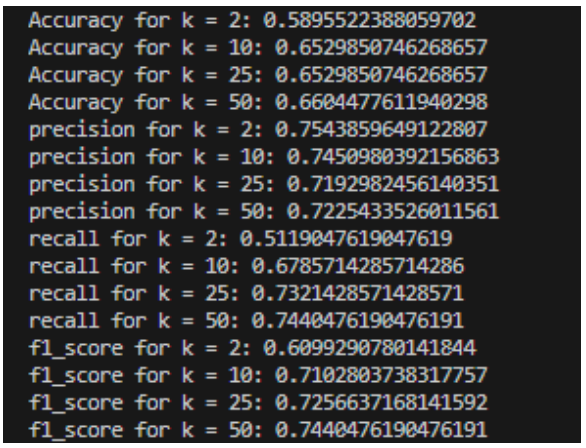
- **KMeans com k = 3**
  - **Cluster 0: Label 0 = 274 || Label 1 = 192**
  - **Cluster 1: Label 0 = 86 || Label 1 = 352**
  - **Cluster 2: Label 0 = 149 || Label 1 = 287**

Como podemos ver, para k = 2, os números se separaram de uma forma mais pronunciada, mesmo que muitas das labels ainda acabaram em clusters errados. Isso se deve no entanto, no que aparenta ser um desbalanceamento das labels 0 e 1, o que faz mais sentido, já que se espera que a maioria dos jogadores que cheguem a NBA, permaneçam na liga por mais de 5 anos, independentemente de sua performance.

Com k = 3, uma coisa similar acontece, porém aqueles jogadores que estão mais no limiar entre um e outro cluster acabam se agrupando no *Cluster 2*.

Podemos ver, no entanto, que o K Means não consegue realmente separar esses jogadores em grupos muito bem definidos baseados nos atributos presentes. Existe uma grande semelhança entre os grupos.

- **Algoritmos Extras**
  Para o KNN temos que:



Como podemos observar, para o KNN**,** os valores se mantêm bastante parecidos com os implementados, no entanto eles na verdade parecem atingir valores mais baixos do que os encontrados pelo o outro KNN. Isso talvez se deva ao fato de que os dados tenham sido transformados pela função *StandartScaler*(), que nesse caso pode ter na verdade piorado a separação entre esses dados e tornado mais difícil o trabalho de classificação realizado pelo KNN. De fato a fase de “treinamento” (ou o que se passa por treinamento no caso do KNN) deve ter influenciado nessa diferença

Para o KMeans, devido a forma como ele é implementado pelo *sklearn*, não consegui a recuperação direta das labels e portanto utilizei o método de *sillhouette_score(),* um método que retorna valores próximos de 1 quando os pontos estão bem classificados, 0 quando estão divididos, e valores negativos para classificações erradas

Para k = 2 temos o valor de : 0.33512238303792363
Para k = 3 temos: 0.22332777349927468

Por esse valor, podemos ver que a situação é parecida também. Em nenhum caso as labels realmente se agrupam de forma muito nítida, porém podemos ver que essa divisão ocorre de forma maior e melhor para o valor de k = 2, exatamente o que aconteceu no outro *KMeans*

Conclusão

Todos os algoritmos pedidos e podemos observar como esses algoritmos lidam com um problema como esse. Por serem algoritmos tão simples, já era esperado que os resultados não fossem maravilhosos, mas o Knn em particular pareceu ter uma performance significativa, especialmente considerando o fato de ser tão simples e não possuir um “modelo” de fato.

Fizemos comparações entre os algoritmos para diferentes valores de k e realizamos análises dos resultados obtidos e a possível explicação deles.

Vale ressaltar que um README irá ser enviado junto com esse TP, mostrando como executar os diferentes algoritmos.

Referências

Gerais
https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html
https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html

Para o algoritmo extra de KNN
https://www.datacamp.com/tutorial/k-nearest-neighbor-classification-scikit-learn

Aluno: Pedro Cimini Mattos de Freitas - 2019007031