Documentação
Algortimos II

Francisco Neves Tinoco Junior

January 10, 2022

Contents

1 Dataset

1.1 Definição

Uma classe criada para automatizar e padrozinar a formatação do dataset.

1.2 Funções

__init__ :

Resumo : Abre o arquivo .dat, coleta os pontos e aciona a função _create_dataset.

Entrada:

path : string -> Caminho para o arquivo .dat

Pseudo Codigo

. Abra o arquivo e leia o seu valor

. Separe as linhas pelo simbolo @, pois ele delimita as carecteristicas do dataset

. Percora as linhas até achar uma que se inicie com data e save os valores dos pontos

. Execute a função para criar o dataset (_create_dataset)

_create_dataset:

Resumo : Formata os valor dos pontos, transformar as categorias em numeros inteiros e salva todos em uma numpy.array.

Pseudo Codigo :

. Percorar toda a listta de pontos salvos anteriormente. Para cada um dos pontos, passe os seus valores para o tipo float.

. Crie um pandas.Dataframe com os pontos e as suas labels

. Utilize a função pd.factorize para transformar os valores das labels em numericos

. Save o valor dos pontos em uma numpy.array

1.3 Variaveis importantes

. dataset : np.array -> Conjunto de todos os pontos junto com o as suas classes. O ultimo valor de cada ponto é a sua classe

. indexador : dict -> Indexa os numeros com as classes originais

2 KD

2.1 Definição

Classe que implementa um arvore kd

2.2 Funções

__init__ :

Resumo : Utilize a função make_kd para fazer uma kdtree e salva na variavel *kd_tree*

Entrada:

data : np.array -> conjunto de pontos criado pela classe Dataset

Pseudo Codigo

. Utilize a função make_kd utilizando data com entrada para fazer uma kdtree e salva na variavel *kd_tree*

create_node:

Resumo : Cria o nó de uma arvore.

Pseudo Codigo :

. Retorne um nó

Propriedades do nó:

. CORTE : int -> O valor que divide a arvore. Caso None, indica que o nó é uma folha da arvore

. DIM : int -> Indica a diminsão que o valor de corte se refere. É igual a Deep mod Dim, aonde Deep é a profundidade da arvore e dim é a quantidade de dimensões dos pontos

. POINT : list -> Indica o valor do ponto. Sse len(POINT) == 0, então o nó não é uma folha

. MENOR: dict -> Arvore da esquerda com os valores menores que CORTE

. MAIOR: dict -> Arvore da direita com os valores maiores ou iguais que CORTE

Notas:

. A propriedade isinstance(tree["CORTE"], type(None)), i.e., caso tree["CORTE"] == None, vai ser usado para conferir se a arvore é uma folha

. Essa função foi criada para criar um padrão de representação entre os nós da arvore

insert_kd:

Resumo : Insere um nó com o valor de *point* na arvore.

Entrada:

tree :dict -> Arvore em que *point* tem que ser inserida

point : np.array -> Ponto a ser inserido

n_dim : int -> O numero de dimensões de *point*

Pseudo Codigo :

. Teste se a arvore é uma folha ou não.

. Caso seja.

  . Retire o ponto presente em tree["POINT"] e save ele em $p0$. Altere o valor de tree["POINT"] para None.

  . Crie dois novos nós, l_tree e r_tree, e atualize os seus valores de DIM para ((dim + 1) mod n_dim), sendo dim a dimensão de *tree*

. Confere o valor de $p0$ e *point* em relação a dim. Caso $p0 < point$, coloque $p0$ em l_tree e *point* em r_tree. Se não, faça o contrario.

. Use a mediana do valor de $p0$ e *point* em dim para definir o valor de Corte de tree

. Adicione l_tree na arvore MENOR de tree e r_tree na arvore MAIOR de tree

. Se não for:

. Veja se o valor da dimensão do ponto é menor ou não ao corte. Se for, repita insert_kd só que com tree = tree["MENOR"]. Se não, repita insert_kd só que com tree = tree["MAIOR"].

Notas:

. insert_kd foi implementada de maneira recursiva.

make_kd:

Resumo : Cria um arvore kd com todos os dados presentes em *data*

Entrada:

data : np.array -> conjunto de pontos criado pela classe Dataset

Pseudo Codigo :

. Cria um nó *kd_tree* que contem o primeiro valor de data e DIM = 0.

. Usando a função *insert_kd*, insira todos os pontos restantes de data em *kd_tree*

. Retorne *kd_tree*

2.3 Variaveis importantes

. kd_tree : dict -> Arvore kd criada usando os pontos presentes em data

3 X_nn

3.1 Definição

Classe que representa um modelo de classificação kneighbors

3.2 Funções

fit :

Resumo : Função que separa os conjuntos de treino e teste gerados por *Dataset* e cria a arvore kd a partir da classe KD

Entrada:

path : string -> Diretorio aonde está o dataset

test_size : int, default = 0.3 -> Proporção do conjunto de teste em relação à todos od dados

Pseudo Codigo:

. Utilizando *path* como entrada em *Dataset*, formate os dados e save em data

. Utilizando o valor de *test_size*, separe os dados em

. X_train : numpy.array -> Conjunto de treinamento contedo os pontos

. y_train : numpy.array -> Conjunto de treinamento contendo as labels

. X_train : numpy.array -> Conjunto de teste contedo os pontos

. y_train : numpy.array -> Conjunto de teste contendo as labels

. Utilizando X_train como entrada, monte uma arvore kd com a classe KD

knn:

Resumo : Função que encontra os k_size pontos mais proximos de todos os ponto em X_test.

Entrada:

k_size : int -> Numero de pontos a serem preditos

cpu : int, default = -1 -> Numero de cpus para serem usadas, Caso -1, usa todas

Pseudo Codigo:

. Para cada ponto presente X_test, utilize a função $multi\_knn$ para calcular os k_size pontos mais proximos. Guarde todas as pevisões em *predict*.

. Retorne predicts

multi_knn:

Resumo : Função intermediaria para ser possivel utilizar o multiprocessing.

Entrada:

point : np.array -> Ponto para ser encontra os k_size pontos mais proximos

Pseudo Codigo :

. Utilizando a função knn_aux, calcule os k_size pontos mais proximos de *point*. Salve o resultado em *kneighbor*.

. Retorne kneighbor

knn_aux:

Resumo : Função que localiza os k_size mais proximos de point

Entrada:

tree : dic -> Arvore em que sera realizada a busca

point : np.array -> Ponto para ser encontra os k_size pontos mais proximos

k_size : int -> Numero de neighbors a serem localidados

kneighbor : list -> Pontos mais proximos

maior_distancia : list-> Maior distancia do ponto mais longe de point em kneighbor e a sua posição

check : list -> Numero de pontos testados pelo algoritmo

dists : list -> Lista com as distancias entre o kneighbor e point

Pseudo Codigo:

. Teste se *tree* é uma folha

. Se for:

. Teste se o numero de pontos em *kneighbor* já atingiu o maximo (k_size).

. Se atingiu:

. Teste se a distancia do novo ponto a *point* dist é menor que a maior distancia no momento.

. Se for:

Troque o ponto mais longe com esse novo ponto em *kneighbor*, assim como a suas distancias em *dists*.

Calcule o novo ponto mais longe assim como a sua distacia e salve esses dados em *maior_distancia*[1].

. Se não atingiu:

. Adicione o ponto a *kneighbor*.

. Se a distacia de *point* ao novo ponto *dist* for a maior, defina *maior_distancia*[0] = dist e *maior_distancia*[1] = len(kneighbor) - 1, i.e., salve a nova maior distancia e a posição do ponto na lista

. Adicione dist a *dists*.

. Se não for:

. Seja dif a diferença entre o *point* e corte.

. Caso ainda não tenha atingido o limite de pontos ou dif seja menor que a maior distancia, continue procurando nas arvores $MENOR$ e $MAIOR$

. Se não, confira em qual arvore o ponto entraria pelo $CORTE$ e continue a busca somente nela.

Notas:

. As variaveis *maior_distancia* e *dists* são usadas para evitar ter que ficar recalculado as distancias entre *point* e os pontos em *kneighbor*

define_class:

Resumo : Função que classifica a classe considerando os kneighbors

Entrada:

kneighbors : list -> Lista contendo k_size neighbors para um ponto

Pseudo Codigo :

. Para cada ponto n em kneighbors, procure em *y_train* qual a sua classe.

. Retorne a classe com mais pontos. Em caso de empate, retorne aleatoriamente uma das classes com mais pontos.

Notas A escrever:

. É necessario compara n aos pontos em *X_train* para saber qual é a sua posição e, sucecivamente, a sua classe.

predict:

Resumo : Função que prediz a classe de todos os valores em *X_test*

Entrada:

k_size : int -> Numero de pontos a serem preditos.

cpu : int, default = -1 -> Numero de cpus para serem usadas, Caso -1, usa todas

Pseudo Codigo A escrever :

. Utilizando a função *knn*, predica os *k_size* vizinhos de cada ponto contido em *X_test* e guarde em *pred*

. Utilizado define_class, predica a classe de cada ponto i com os vizinhos *pred*[*i*]. Armazene as classes previstas em *classes_pred*

. Retorne *classes_pred*

evaluate:

Resumo : Função que calcula as metricas de acerto.

Entrada:

k_size : int -> Numero de pontos a serem preditos.

cpu : int, default = -1 -> Numero de cpus para serem usadas, Caso -1, usa todas

Pseudo Codigo A escrever :

. Utilizando predict, pege cada classe prevista para cada ponto em *X_test*.

. Calcule qual é classe c com mais pontos no conjuto de teste.

. Usando *evaluate_aux*, os valores de true_positives(tp), true_negatives(tn), false_positives(fp) e false_negatives(fn)

. Retorne as metricas de acuracia, precição e revocação.

evaluate_aux:

Resumo : Função que calcula as metricas de acerto.

Entrada:

c : int -> Classe tomada como padrão

pred : list -> Classes previstas para os pontos

Pseudo Codigo :

. Considerando *c* como a classe referencial, calcule os true_positives(tp), true_negatives(tn), false_positives(fp) e false_negatives(fp) de *pred* em relação a *y_test*

. Retorne tp,tn,fp,fn

3.3 Variaveis importantes

. X_train : np.array -> Contem os dados de treinamento

. y_train : np.array -> Contem a label dos dados de treinamento

. X_test : np.array -> Contem os dados de teste

. y_test : np.array -> Contem a label dos dados de teste

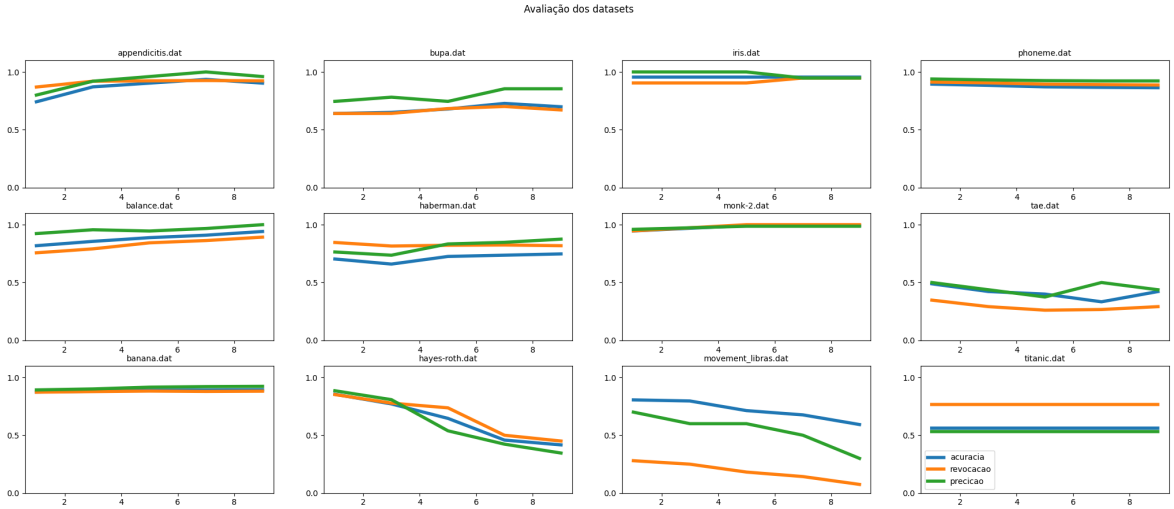
. kd_tree : dict -> Uma kd_tree criada a partir dos dados de treinamento

3.4 Notas

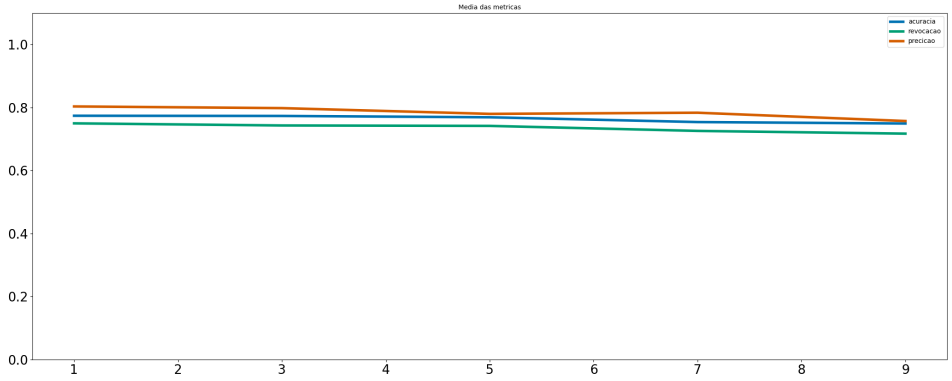
. A classe foi criada em cima de calcularas metricas e predições do dataset passado em *path*. Logo, utiliza-la para tentar prever pontos fora do arquivo não é possivel.

4 Resultado do modelo

A fim de testar como o modelo estava se comportando, foram escolhidos doze datasets que continham um conjunto de pontos e a suas labels. Foi-se aplicado o X_nn em cada um dos datasets variando a quantidade de vizinhos a serem considerados na classificação.



O grafico acima demonstra os resultados para cada um dos datasets. É possivel observar que o algoritmo teve um resultado possitivo na classificação da maioria dos datasets, sendo que o aumento do numero de vizinhos influenciava as taxas de acertos, via de regra positivamente. Nos datasets em que o resultado foi misto ou ruim, é possivel ver uma relação inversa, aonde o aumento do numero de vizinhos acabava prejudicando os resultados. Essa oscilação de resultados pode ser resultante de diversos fatores, desde falhas na implementação, até a separação de alguns datasets não se muito linear, necessitando de se aplicar algumas tecnicas, como aplicar um kernel, antes de se utilizar o algoritmo. Todas as metricas acabaram tendo uma flutuação bastante parecida.



Na figura acima, é possivel ver os resultados medios de todos os datasets. O resultando parece ser contraditorio com o apresentado individualmente, aonde o numero de vizinhos parece não afetar o resultado. Isso provalvemente está ocorrendo pelas oscilações negativas estarem cancelando as positivas.

O codigo que gerou essas imagens está disponivel no arquivo images.py