Universidade Tecnológica Federal do Paraná

Relatório Atividade KNN e DWNNN

Aluno: Gustavo Rodrigues Bassaco
Professor: Rafael Mantovani

Junho
2023

Universidade Tecnológica Federal do Paraná

Relatório

Quarto relatório de Projeto de Pesquisa do Curso de Engenharia de Computação da Universidade Tecnológica Federal do Paraná .

Aluno: Gustavo Rodrigues Bassaco

Professor: Rafael Mantovani

Junho
2023

Conteúdo

1 Introdução

Nos últimos anos, temos testemunhado avanços significativos no campo da Inteligência Artificial (IA), impulsionados pelo desenvolvimento de técnicas e algoritmos inovadores. Entre essas técnicas, o Aprendizado Baseado em Instâncias (ABI) emergiu como uma abordagem promissora, capaz de lidar com problemas complexos de forma eficiente e eficaz.

O ABI é um subcampo da IA que se concentra no uso de instâncias individuais para realizar tarefas de aprendizado e inferência. Em vez de generalizar a partir de um conjunto de exemplos, o ABI utiliza casos de treinamento específicos para solucionar novos problemas. Essa abordagem se baseia na premissa de que exemplos semelhantes tendem a ter respostas semelhantes, e, portanto, é possível usar experiências passadas para orientar a tomada de decisões futuras.

Russell e Norvig (2010), em sua obra "Artificial Intelligence: A Modern Approach", exploram o conceito de ABI como uma técnica fundamental para o desenvolvimento de sistemas inteligentes. Eles destacam a importância de considerar exemplos individuais como unidades de conhecimento valiosas, que podem ser usadas para inferir soluções para problemas complexos. Essa abordagem, também conhecida como aprendizado baseado em casos, tem sido aplicada em diversos domínios, incluindo diagnóstico médico, processamento de linguagem natural e recomendação de produtos.

Dentro desse contexto, o presente projeto tem como objetivo aplicar as técnicas mais conhecidas do ABI, como KNN (K-Nearest Neighbors) e DWNN (Dynamic Weighted Nearest Neighbors), a dois conjuntos de dados diferentes: um para classificação e outro para regressão. Serão exploradas variações de medidas de distâncias e serão realizadas comparações com implementações presentes na literatura. Essa abordagem experimental permitirá avaliar a eficácia e a adequação dessas técnicas em relação aos conjuntos de dados utilizados, contribuindo para um maior entendimento do ABI e suas aplicações práticas.

2 Desenvolvimento

2.1 Obtenção de Dados para Classificação

Para este trabalho, utilizamos conjuntos de dados gratuitos disponíveis no site OpenML (https://www.openml.org). Para o problema de classificação, escolhemos um conjunto de dados de exames de diabetes de id=37, que consiste em oito informações e a classificação de se uma pessoa possui ou não

diabetes. Abaixo, são apresentadas as informações desse conjunto de dados:

Tabela 1: Informações do conjunto de dados de diabetes

Número	**Informação**
1	Número de vezes grávida
2	Concentração de glicose plasmática após 2 horas em um teste oral de tolerância à glicose
3	Pressão arterial diastólica (mm Hg)
4	Espessura da prega cutânea do tríceps (mm)
5	Insulina sérica de 2 horas (mu U/ml)
6	Índice de massa corporal (peso em kg/(altura em m)^2)
7	Função de pedigree de diabetes
8	Idade em anos
9	Variável de classe (0 ou 1)

Normalização do Dataset

A normalização dos dados é um processo fundamental no pré-processamento de dados em diversas tarefas de aprendizado de máquina. Consiste em ajustar as escalas dos atributos de um conjunto de dados, de modo que todas as variáveis estejam na mesma faixa ou intervalo. Isso é importante porque muitos algoritmos de aprendizado de máquina são sensíveis à escala dos atributos e podem ser influenciados negativamente por diferenças significativas de magnitude entre eles.

Ao normalizar os dados, é possível eliminar viés, acelerar a convergência do modelo, reduzir o impacto de valores discrepantes e melhorar a interpretação dos resultados. Para realizar essa normalização, utilizamos a biblioteca scikit-learn, que pode ser instalada através do comando pip install scikit-learn. A seguir, apresentamos o código para realizar a normalização dos dados:

```
from sklearn.preprocessing import MinMaxScaler

def data_normalizer(dataset):
    # Crie uma instância do MinMaxScaler
    scaler = MinMaxScaler(feature_range=(0, 1))

    # Aplique a normalização nas colunas do dataset
    dataset = scaler.fit_transform(dataset)
    return dataset
```

Imputação de dados

Uma etapa crucial no pré-processamento dos dados é a imputação, especialmente no caso do nosso banco de dados de diabetes. O conjunto de dados consiste em um total de 768 amostras, das quais 500 pertencem a pessoas sem diabetes e 268 pertencem a pessoas com diabetes. Essa disparidade no número de amostras pode resultar em dificuldades para o algoritmo classificar corretamente a classe das amostras.

Para lidar com esse desequilíbrio, realizamos a imputação de dados adicionais de 262 pessoas com diabetes, a fim de equilibrar o conjunto de dados. Nesse processo, utilizamos a média de cada coluna dos dados de pessoas com diabetes para preencher os valores das novas amostras geradas. Dessa forma, garantimos que as classes estejam mais equilibradas e reduzimos o viés potencial que poderia surgir devido ao desequilíbrio inicial. Abaixo o código usado para imputar dados:

```
import numpy as np

def data_imputer(data):
    # Filtrar os elementos com valor 1 na última coluna
    data_ones = data[data[:, -1] == 1]

    # Calcular a média de cada coluna (exceto a última)
    column_means = np.mean(data_ones[:, :-1], axis=0)

    # Criar novos dados com base nas médias das colunas
    new_data = np.tile(column_means, (232, 1))

    # Definir a última coluna dos novos dados como 1 (representando a classe)
    new_data = np.hstack((new_data, np.ones((232, 1))))

    # Concatenar os novos dados com os dados existentes
    data = np.concatenate((data, new_data), axis=0)
    return data
```

2.2 KNN discreto Implementado

A implementação do KNN discreto é uma etapa essencial para a classificação de dados em problemas de aprendizado de máquina. O algoritmo KNN (K-vizinhos mais próximos) é uma técnica de classificação baseada em instâncias, onde o objetivo é encontrar os K vizinhos mais próximos de um padrão de consulta em um conjunto de dados e atribuir a classe mais

frequente entre esses vizinhos ao padrão de consulta. A seguir está a implementação do KNN discreto em Python:

```
import numpy as np

def knn_discrete(dataset, query, res,  k):
    # calcular a distância de query para cada padrão do dataset
    distance = np.zeros(dataset.shape[0])
    for i in range(dataset.shape[0]):
        # Distancia Euclidiana
        distance[i] = np.sqrt(np.sum((dataset[i] - query)**2))
        # Distancia Manhattan
        #distance[i] = np.sum(np.abs(dataset[i] - query))

    # retornar os k mais próximos
    idx = np.argsort(distance)[:k]
    candidates = res[idx]

    occurrences = np.unique(candidates, return_counts=True)
    class_counts = occurrences[1]
    most_frequent_class = occurrences[0][np.argmax(class_counts)]
    return most_frequent_class
```

O algoritmo calcula a distância entre o padrão de consulta e cada instância do conjunto de dados de treinamento usando a distância Euclidiana ou a distância Manhattan (comentada). Em seguida, os índices dos k vizinhos mais próximos são obtidos e suas classes são armazenadas em `candidates`.

Para determinar a classe mais frequente entre os vizinhos, as ocorrências de cada classe são contadas usando a função `np.unique`, e a classe com maior contagem é retornada como a classe atribuída ao padrão de consulta.

Para realizar a simulação de todos os algoritmos, executamos cada um deles 10 vezes, variando o valor de k entre 1, 3, 5, 7, 9 e 11. Além disso, utilizamos conjuntos de teste que correspondem a 10%, 20%, 30%, 40%, 50% e 60% do conjunto total de dados.

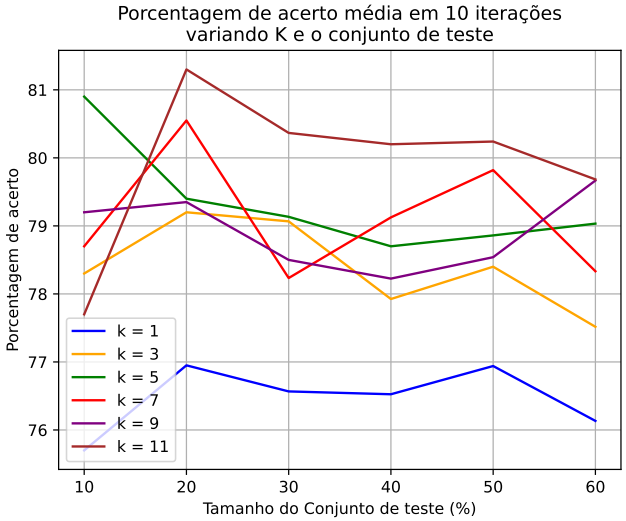
A escolha de executar o algoritmo em 10 iterações é feita com o objetivo de utilizar, provavelmente, todos os dados pelo menos uma vez no conjunto de teste. Dessa forma, podemos obter uma melhor compreensão do desempenho do algoritmo apresentado ao avaliá-lo em diferentes cenários de teste.

Ao realizar múltiplas execuções e variar tanto o valor de k quanto o tamanho do conjunto de teste, obtemos uma avaliação mais abrangente do

algoritmo KNN. Essa abordagem nos permite observar como o desempenho do algoritmo varia de acordo com diferentes configurações de hiperparâmetros e tamanhos de conjunto de teste.

Abaixo na figura 1 podemos ver os resultados obtidos na execução do KNN discreto usando a distância euclidiana.

Figura 1: KNN discreto distância Euclidiana



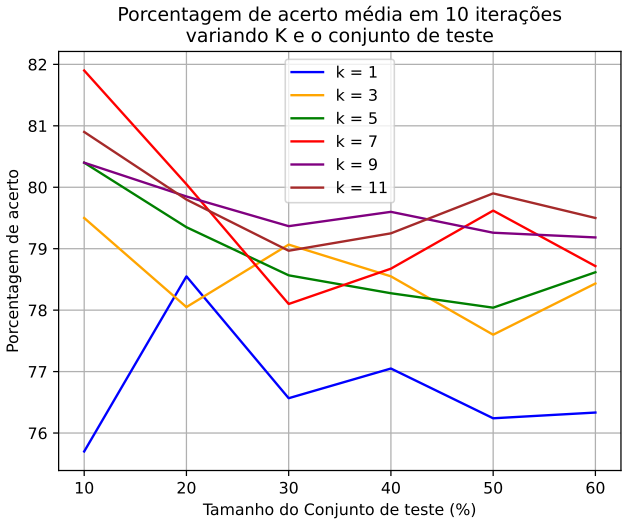
Fonte: Autoria própria

Pelo gráfico da figura 1, podemos notar que para essa implementação usando esse banco de dados, melhor opção é usar k = 11 e 20% da população para teste. Essa combinação foi a que teve um resultado mais convincente.

Na figura 2 podemos ver os resultados obtidos na execução do KNN discreto usando a distância Manhattan.

Com o gráfico gráfico da figura 2, pode-se notar que para essa implementação usando esse banco de dados, em quase todos os casos, o conjunto de teste de tamanho 10% é o mais indicado. Além disso a melhor opção é usar k = 11, pois essa combinação foi a que teve o melhor desempenho em classificar corretamente as entradas.

Figura 2: KNN discreto distância Manhattan



Fonte: Autoria própria

2.3 KNN discreto Literatura

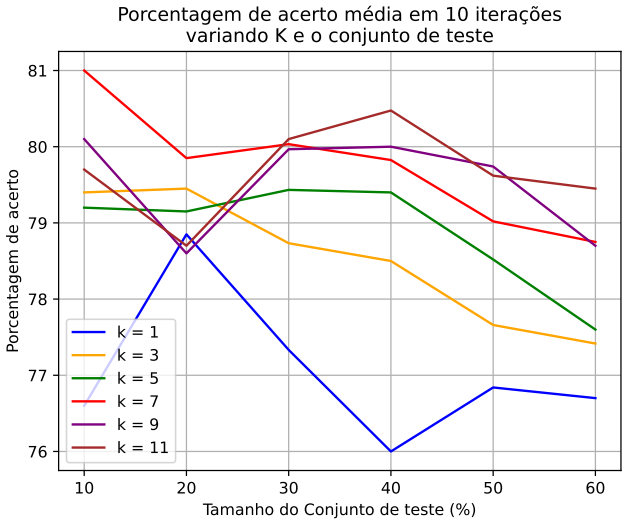
Como comparação, utilizamos e testamos o algoritmo KNN implementado na biblioteca scikit-learn, mais especificamente a função KNeighborsClassifier. Essa função é configurada para usar a distância euclidiana por padrão como medida de proximidade para classificação dos dados.

O código abaixo exemplifica a implementação do algoritmo KNN utilizando a função KNeighborsClassifier:

```
from sklearn.neighbors import KNeighborsClassifier

def knn_discrete_literatura(X_train, query, y_train, k):
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)
    prediction = knn.predict([query])
    return prediction[0]
```

É possível perceber que o gráfico gerado da figura 3, assim como na figura 2, tem o sua melhor resposta média quando k = 7 e a tamanho do teste é 10%. Apesar de o algoritmo usar a distância euclidiana, seus resultados forem diferente do KNN discreto euclidiano implementado. Isso deve-se ao fato que a cada iteração o conjunto de teste e treino é gerado novamente, e em alguns caso o teste pode ser mais fácil para o algoritmo.

Figura 3: KNN discreto Literatura



Fonte: Autoria própria

2.4 DWNN discreto Implementado

O algoritmo DWNN (Distance Weighted Nearest Neighbors) é uma variação do KNN que utiliza pesos para ponderar a influência das instâncias vizinhas na classificação. Esses pesos são calculados com base nas distâncias entre as instâncias vizinhas e o padrão de consulta.

No código, primeiro calculamos as distâncias entre o padrão de consulta e cada instância do conjunto de dados utilizando a distância Euclidiana ou Manhattan. Em seguida, utilizamos uma função de base radial para calcular os pesos correspondentes a cada instância. Nesse caso, a função de base radial utilizada é a distribuição gaussiana, onde sigma controla o decaimento dos pesos.

$$w(i) = \exp\left(-\frac{\text{distance}(i)^2}{2 \cdot \sigma^2}\right)$$

Em seguida, selecionamos os k vizinhos mais próximos com base nas distâncias calculadas e os respectivos pesos. A partir desses vizinhos, somamos os pesos de cada instância para cada classe presente no conjunto de dados. A classe com a maior soma de pesos é então selecionada como a classe prevista para o padrão de consulta.

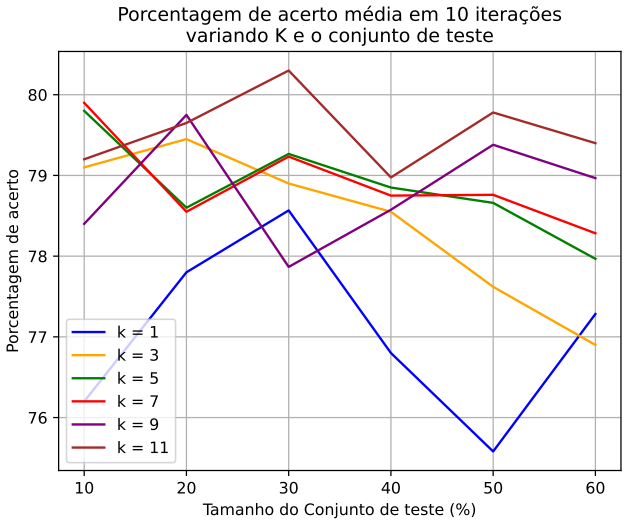
Abaixo podemos ver o código implementado:

```python
import numpy as np
def dwnn_discrete(dataset, query, res, k=3, sigma=1):
    # calcular a distância de query para cada padrão do dataset
    distance = np.zeros(dataset.shape[0])
    for i in range(dataset.shape[0]):
        # Distancia Euclidiana
        #distance[i] = np.sqrt(np.sum((dataset[i] - query)**2))
        # Distancia Manhattan
        distance[i] = np.sum(np.abs(dataset[i] - query))

    # calcula os pesos para cada instância usando uma função de base radial
    weights = np.exp(-(distance ** 2) / (2 * sigma ** 2))

    # retornar os k mais próximos
    idx = np.argsort(distance)[:k]
    candidates = res[idx]
    w = weights[idx]

    # calcular as distâncias para cada classe
    unique_classes = np.unique(res)
    total = np.zeros(len(unique_classes))

    # somar os pesos de cada instância para cada classe
    for i in range(len(candidates)):
        index = int(candidates[i])
        total[index] += w[i]

    # retornar o índice da classe com maior soma
    id_class = np.argmax(total)
    return unique_classes[id_class]
```

Com isso é possível ver na figura 4, os resultados obtidos usando o DWNN usando a distância Euclidiana. É possível notar a melhor escolha de hiperparâmtros seria k = 11 e tamanho do teste de 30%.

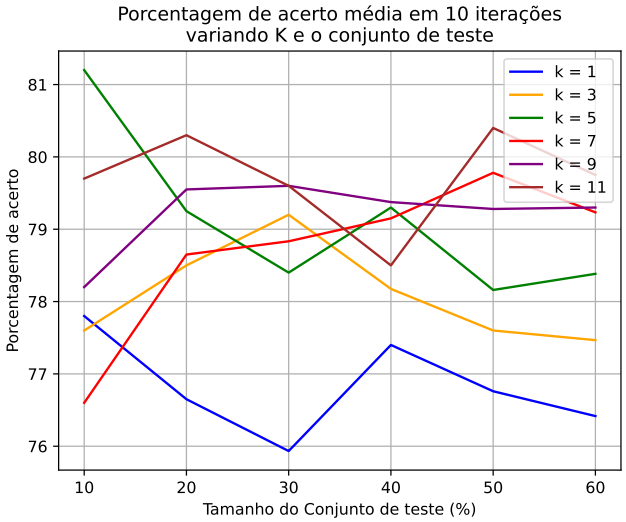
Já para o DWNN usando distância Manhattan, como podemos ver na figura 5, a melhor combinação seria k = 5 e tamanho de teste de 10

Figura 4: DWNN discreto distância Euclidiana



Fonte: Autoria própria

Figura 5: DWNN discreto distância Manhattan



Fonte: Autoria própria

2.5 DWNN discreto Literatura

Para fins de comparação, utilizamos e testamos o algoritmo NearestNeighbors implementado na biblioteca scikit-learn. A biblioteca não possui especi-

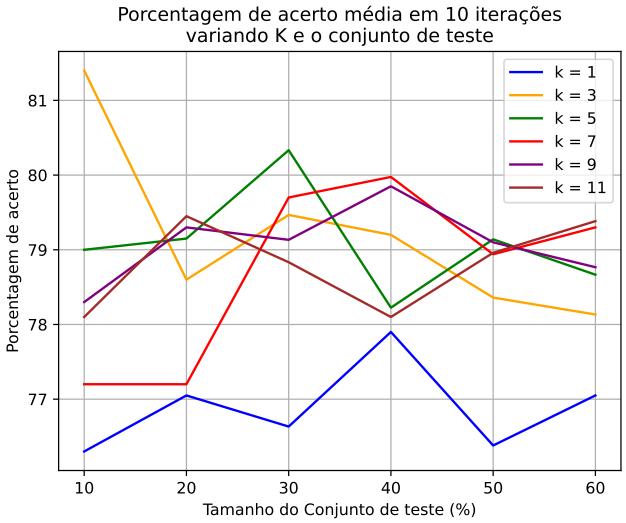
ficamente um DWNN classifier, porém é possível encontrar os vizinhos mais proximos com NearestNeighbors e calcular os pesos separadamente.

A função custom_weights é definida para calcular os pesos personalizados com base nas distâncias, utilizando a fórmula da distribuição gaussiana. Esses pesos são usados para ponderar a influência de cada vizinho durante a classificação.

Abaixo o código implementado:

```
from sklearn.neighbors import NearestNeighbors

def custom_weights(distances, sigma=1):
    weights = np.exp(-(distances ** 2) / (2 * sigma ** 2))
    return weights

def dwnn_discrete_literatura(X_train, query, y_train, k=3, sigma=1):
    # Calcular as distâncias
    nbrs = NearestNeighbors(n_neighbors=k, algorithm='kd_tree', metric='euclidea
    nbrs.fit(X_train)
    distances, indices = nbrs.kneighbors([query])

    # Calcular os pesos personalizados
    weights = custom_weights(distances, sigma)

    # Calcular a soma ponderada das classes vizinhas
    weighted_sum = np.sum(weights * y_train[indices])

    # Calcular a classe predita
    prediction = round(weighted_sum / np.sum(weights))
    return prediction
```

Figura 6: DWNN discreto Literatura



Fonte: Autoria própria

2.6 Obtenção de Dados para Regressão

Para o problema de classificação, escolhemos um conjunto de dados de casas_brasileiras de id=42688, que consiste em 12 informações sobre o terreno e o valor correspondente da casa. Abaixo, são apresentadas as informações desse conjunto de dados:

Tabela 2: Informações do conjunto de dados de casas_brasileiras

Número	**Informação**
1	total_(BRL) (alvo)
2	cidade
3	área
4	quartos
5	banheiro
6	vagas de estacionamento
7	chão
8	animal
9	mobília
10	hoa_(BRL)
11	aluguel_valor_(BRL)
12	propriedade_tax_(BRL)
13	fire_insurance_(BRL)

Deixando os valores numéricos

Como alguns valores do conjunto de dados de casa brasileiras era numéricos, foi necessário transforma-los em numéricos. Para isso foi implementada um função que utiliza o LabelEncoder presente no pacote sklearn. abaixo é mostrado o código usado:

```
from sklearn.preprocessing import LabelEncoder

def preprocess(data):
    # Extrair informações do cabeçalho
    attributes = data['attributes']
    data_instances = data['data']
    # Converter atributos categóricos em numéricos
    le = LabelEncoder()
    for attr_index, attr in enumerate(attributes):
        attr_type = attr[1]
        if isinstance(attr_type, list):
            # Atributo categórico
            attr_values = [instance[attr_index] for instance in data_instances]
            encoded_values = le.fit_transform(attr_values)
            for instance_index, instance in enumerate(data_instances):
                instance[attr_index] = encoded_values[instance_index]
    return data_instances
```

2.7 KNN contínuo Implementado

Tradicionalmente o KNN é utilizado para classificar objetos em classes distintas, porém o KNN Regressão é uma variante que busca prever valores contínuos em vez de atribuir uma classe discreta.

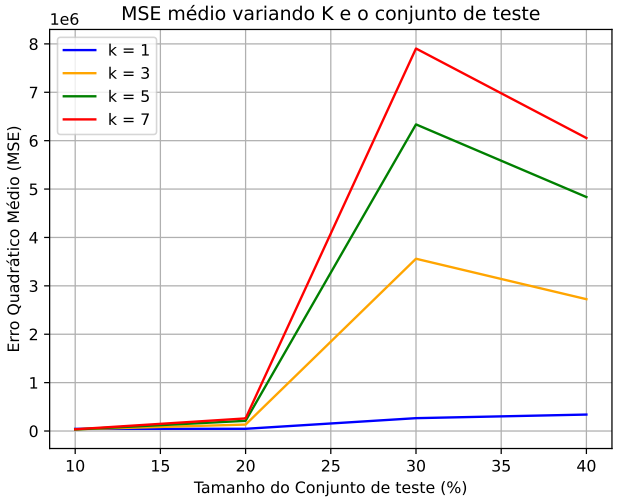
O objetivo do KNN Regressão é estimar um valor numérico para uma determinada variável alvo com base nas características dos pontos de dados mais próximos. O algoritmo utiliza uma abordagem de vizinhança, onde os pontos de dados são representados em um espaço multidimensional e a similaridade entre eles é medida com base em uma métrica de distância, como a distância Euclidiana ou a distância de Manhattan.

Com isso foi implementado o algoritmo KNN de regressão usando duas possíveis medidas de distância, Euclidiana e Manhattan. Abaixo é possível ver o código implementado:

```python
import numpy as np

def knn_regression(dataset, query, res, k):
    # Calcular a distância de query para cada padrão do dataset
    distance = np.zeros(dataset.shape[0])
    for i in range(dataset.shape[0]):
        # Distância Euclidiana
        # distance[i] = np.sqrt(np.sum((dataset[i] - query)**2))
        # Distância Manhattan
        distance[i] = np.sum(np.abs(dataset[i] - query))

    # Retornar os k mais próximos
    idx = np.argsort(distance)[:k]
    k_nearest_res = res[idx]

    # Calcular a média dos valores dos k vizinhos mais próximos
    mean_value = np.mean(k_nearest_res)
    return mean_value
```

Na figura 7 podemos ver os resultados obtidos a partir da implementação do KNN contínuo com distância Manhattan. é possível notar que as melhores escolhas para esse banco de dados são os que tem um conjunto de teste entre 10 e 20%. não é possível visualizar mas a combinação que obteve o menor erro foi usando conjunto de teste de 10% com k=5. Vale ressaltar que os dados na coluna y estão variado em um milhão a cada valor.

Figura 7: KNN contínuo Implementado Manhattan



Fonte: Autoria própria

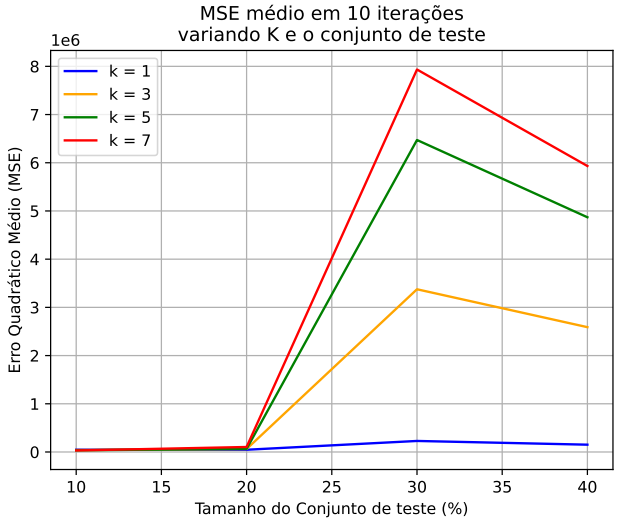
2.8 KNN contínuo Literatura

Já para compararmos o desempenho do algorítmo testado com um da literatura, escolhemos o KNeighborsRegressor presente na biblioteca sklearn. Por padrão ele trabalha com distâncias euclidianas. Abaixo é possível ver a implementação:

```
def knn_regression_literatura(dataset, query, res, k):
    knn = KNeighborsRegressor(n_neighbors=k)
    knn.fit(dataset, res)
    y_pred = knn.predict([query])
    return y_pred[0]
```

Na figura 8 podemos ver os resultados obtidos a partir da implementação da literatura. é possível notar que as melhores escolhas para esse banco de dados são os que tem um conjunto de teste entre 10 e 20%. não é possível visualizar mas a combinação que obteve o menor erro foi usando conjunto de teste de 10% com k=5.

Figura 8: KNN contínuo Literatura



Fonte: Autoria própria

3 Conclusão

Neste trabalho, exploramos e aplicamos os algoritmos KNN (K-Nearest Neighbors) e DWNN (Distance-Weighted Nearest Neighbors) para tarefas de classificação e regressão. Esses algoritmos são baseados no princípio da vizinhança, onde os exemplos de treinamento mais próximos ao exemplo de teste são utilizados para realizar as previsões.

No caso da classificação, o KNN mostrou-se eficaz ao atribuir uma classe ao exemplo de teste com base na maioria das classes presentes nos vizinhos mais próximos. A escolha do valor de K é crucial, pois influencia a precisão do modelo. Observamos que um valor de K muito baixo pode levar a uma classificação sensível a ruídos, enquanto um valor muito alto pode levar a um viés excessivo.

Para a tarefa de regressão, o KNN pode ser adaptado para prever valores numéricos em vez de classes. Utilizando a abordagem de KNN regressão, os valores de saída são obtidos através da média ou da ponderação dos valores das amostras vizinhas. Isso permite estimar um valor contínuo com base nos vizinhos mais próximos. A escolha adequada do valor de K é fundamental para obter previsões precisas.

Além disso, exploramos o algoritmo DWNN, que atribui pesos diferentes aos vizinhos com base na distância. Isso permite que exemplos mais próximos tenham uma influência maior nas previsões. Essa abordagem é particular-

mente útil quando a relevância dos exemplos próximos varia conforme a distância.

Ao longo do trabalho, também destacamos a importância da etapa de pré-processamento de dados, que inclui normalização, tratamento de valores ausentes e seleção de características relevantes. Essas etapas têm um impacto significativo no desempenho e na precisão dos modelos KNN e DWNN.

Em resumo, os algoritmos KNN e DWNN são técnicas simples e intuitivas, amplamente utilizadas em problemas de classificação e regressão. Eles são versáteis, podendo ser aplicados em diversas áreas, como medicina, finanças e reconhecimento de padrões. No entanto, é importante ajustar os parâmetros adequadamente e considerar as características específicas do conjunto de dados para obter resultados confiáveis e precisos.

Referências

[1] Russell, S. J., & Norvig, P. (2010). *Artificial Intelligence: A Modern Approach*. Pearson Education.