
GRUPO: G_4_6

Magda Costa - up202207036

Rafael Pacheco - up202206258

Sofia Machado - up202207203

Machine Learning I (CC2008) - Practical Assignment

Índice

Introdução

[voltar ao índice]

Este trabalho tem como objetivo desenvolver e avaliar uma adaptação de um algoritmo de classificação. A análise será realizada com datasets disponíveis no OpenML, conforme sugerido.

Como sabemos, o desempenho dos algoritmos de Machine Learning (ML) pode ser significativamente comprometido por desafios inerentes aos dados do mundo real, como

atributos qualitativos com muitos valores possíveis, ruído, outliers, desbalanceamento de classes, multiclasses e sobreposição de classes.

Através da seleção de um algoritmo de classificação, vamos analisar o funcionamento teórico e prático, ao identificar uma característica dos dados que o afeta. Após esta análise, implementaremos uma alteração no algoritmo para melhorar a sua performance. Finalmente, vamos reavaliar o algoritmo modificado e comparar os resultados com a versão original.

Este estudo visa compreender melhor os desafios que os dados reais apresentam aos algoritmos de ML e como adaptar estes algoritmos para obter modelos mais precisos e eficazes.

Algoritmo Escolhido

[voltar ao índice]

Explicação KNN

[voltar ao índice]

K-Nearest Neighbors, ou simplesmente **K-nn**, é um dos algoritmos de classificação mais simples e utiliza distâncias para avaliar o grau de semelhança entre instâncias e, consequentemente, classificá-las. Para isso, serve-se dos **k objetos** mais próximos (ou seja, mais semelhantes) ao novo exemplo.

Fase de treino

Numa fase de treino, o algoritmo guarda os objetos de treino em memória. Isto significa que todos os atributos de uma instância passada ao algoritmo serão memorizados para comparação futura, ou seja, não existe propriamente uma **fase de aprendizagem**.

Fase de previsão

Quando pretendemos classificar um novo exemplo, o algoritmo **calcula as distâncias** possíveis entre a nova instância e todos os objetos de treino.

Após processar todos esses cálculos, as **distâncias são associadas aos objetos** de treino e sucessivamente **ordenadas por ordem crescente** num processo de **ranking**: a distância mais baixa tem rank 1, a segunda mais baixa tem rank 2, e assim sucessivamente.

De seguida, de acordo com o valor de k previamente estabelecido, escolherá os **k objetos com melhor ranking**. Este processo garante que os exemplos com valores de atributos **mais próximos** sejam os considerados, já que quanto **mais semelhantes** em relação à nova instância, menor será a distância entre eles, pelo que o seu ranking será mais alto.

Concluída a seleção dos k melhores objetos, segue-se a **análise da classe** a que os mesmos pertencem, pelo que a classificação da nova instância será feita com base na **classe maioritária dos k elementos escolhidos**.

Código KNN

[voltar ao índice]

Para o desenvolvimento de métodos de resolução de classes desbalanceadas, decidimos utilizar a implementação de K-Nn fornecida no guião:

- https://github.com/rushter/MLAlgorithms

No entanto após analisar o código percebemos que algumas funções não são necessárias para o seu funcionamento, uma vez que não estão implementadas. De notar que foi alterado o código de modo a que seja possivel determinar qual o número de vizinhos a analizar quando se cria KNNClassifier(). Mantivemos, no entanto, a métrica de distância como euclidiana.

1. Base Estimator

A classe BaseEstimator verifica se os dados fornecidos estão guardados no formato pretendido. Caso isso não aconteça, os dados são convertidos para arrays de numpy.

2. Implementação de K-nearest Neighbors

Estas classes servem de base para classificar ou usar regressão nos dados usando o modelo `K-nearest Neighbors`.

`KNNBase` possui valores implícitos, caso estes não sejam passados à função:

- Número de vizinhos, **k**: 5
- Métrica de distância usada pelo algoritmo automaticamente, **distance_func**: euclideana

Segue abaixo uma breve explicação das classes e das respetivas funções principais:

1. `KNNBase` -> **_predict_** :
    - Calcula **distâncias** entre o novo valor de target e os valores existentes e guarda em pares de (distância, classe)
    - Ordena pares por **'ranks'**
    - Classe final é determinada através da classe mais comum entre vizinhos (**mode**) ou a média (**average**)
1. `KNNClassifier` -> **aggregate** :
    - `Nota` : se houver empate entre a escolha de classes, a decisão é arbitrária
    - Usa 'Counter' para contar as ocorrências de cada classe
2. `KNNRegressor` -> **aggregate** :
    - Devolve a média de todos os valores da classe target

Complicações do KNN

[voltar ao índice]

De modo a que futuramente possamos escolher qual a complicação do KNN que queremos resolver, iremos numa fase inicial analisar os diferentes problemas que podemos encontrar:

- Valor de k

Embora não seja dependente da existência das complicações abaixo elaboradas (é apenas um parâmetro), o primeiro problema que encontramos quando usamos este algoritmo é a definição de um **valor ideal para k**.

Se o valor for **muito baixo**, o classificador torna-se muito sensível a exemplos muito semelhantes à nova instância, podendo fazer **overfitting**, focando-se em valores específicos que são nocivos para uma classificação mais abrangente. Para além disso, é muito mais suscetível à consideração de **outliers** no dataset, podendo tomar decisões totalmente inconsistentes com a realidade, baseadas em **valores excepcionais** que são **significativamente diferentes** da generalidade das instâncias de treino.

No entanto, se k apresentar um valor **muito elevado**, o algoritmo não só tem **tendência para escolher** a **classe maioritária** mas também focar-se em **instâncias pouco semelhantes** ao novo objeto (**generalização excessiva**), conduzindo a classificações incorretas. É fácil de perceber que uma consideração mais ampla de exemplos aumenta também a probabilidade de escolher **ruído do dataset** como opção de classificação do novo exemplo.

Posto isto, percebemos que é fundamental escolher um valor de k que apresente um bom **tradeoff** entre as complicações anteriormente referidas, com a finalidade de refinar a capacidade de classificação do algoritmo.

Apesar disto, como o foco deste projeto não é otimização de parâmetros, **decidimos não investigar** em profundidade um valor de k ótimo para os dataset em questão.

- Atributos qualitativos com muitos valores

Para além de ser necessário aplicar algum tipo de **encoding** aos valores qualitativos para fornecer ao algoritmo algum tipo de métrica de **comparação entre instâncias**, uma quantidade elevada de valores aumenta consideravelmente a exigência computacional do algoritmo aquando do cálculo de várias distâncias.

Uma das resoluções possíveis para este problema seria **redução de dimensionalidade**, o que implica, na maioria dos casos, a perda de dados ou características dos mesmos.

- Ruído e outliers

A presença de valores irrelevantes ou de outliers pode causar classificações indevidas, especialmente para um algoritmo como o K-nn, já que se **baseia apenas em semelhanças** entre instâncias e **não possui** uma fase de aprendizagem capaz de **distinguir padrões pertinentes de aleatórios** como, por exemplo, as **Support Vector Machines** (SVMs), que utilizam conceitos como hiperplanos e kernels capazes de mapear os dados para espaços de diferentes dimensões para detetar até padrões significativamente complexos, sendo por isso mais robustos a este tipo de problemas.

Para mitigar esta complicação, podemos aplicar uma análise dos valores do dataset e **remover outliers** e valores que determinamos ser **correlacionados de forma excessivamente alta ou baixa**.

- Multiclasses

Em datasets multiclasse, é realmente **desafiante encontrar termos comparativos** devido à possibilidade do **aumento da dimensionalidade** e de **dispersão de objetos** pelo espaço, bem como o aumento da complexidade de padrões que o K-nn pode não ser capaz de detetar.

Neste caso, as melhores opções de resolução ao problema seriam reduzir a dimensionalidade ou **utilizar outro algoritmo** de classificação alternativo, mais robusto a este problema, como por exemplo as SVM anteriormente mencionadas.

- Classes sobrepostas

O algoritmo K-nn será **gravemente afetado** pela presença de classes sobrepostas nos seus dados de treinamento. Este problema complica de forma significativa a **capacidade de distinção de classe** de um novo exemplo devido à **alta semelhança** entre instâncias **de classes distintas**, criando regiões onde a decisão do algoritmo pode ser verdadeiramente **ambígua e incorreta** com uma probabilidade relativamente elevada, o que **reduz consideravelmente a precisão** do algoritmo.

Se um bom **pré-processamento** dos dados onde se aplica **remoção de outliers e normalização** de valores não for capaz de reduzir este problema de forma a que o algoritmo não seja tão afetado pelo mesmo, **sugere-se** o uso de um **algoritmo de classificação alternativo** (mais uma vez, SVMs são relativamente robustas à existência de classes sobrespostas devido à sua capacidade de usar **"kernel tricks"**).

- Classes desbalanceadas

A existência de classes fortemente desbalanceadas num dataset é **extremamente nociva** para o algoritmo em questão, por fundamentar a sua classificação na classe maioritária dos k elementos com maior semelhança de atributos comparativamente com o novo objeto. Um erro muito comum em classificações de datasets com este problema é a rejeição da classe correta devido à porporção da classe maioritária, **M**, relativamente à classe minoritária, **m**. É possível que, mesmo com um valor de k ideal e otimizado, ao ter uma **classe minoritária**

com uma boa semelhança ao novo exemplo e uma **classe maioritária completamente distinta** do mesmo, os k vizinhos mais próximos sejam algo do tipo:

- m m M M M M

Onde a classe minoritária, m, **seria descartada** simplesmente por ser inferior em número, **apesar de ser a classificação correta.**

Problema Escolhido - Classes Desbalanceadas

[voltar ao índice]

Para este trabalho decidimos escolher 'Classes Desbalanceadas' como o problema a ser resolvido, uma vez que percebemos que é algo que pode causar diversos problemas ao algoritmo como foi mencionado previamente. Para além disso quando fizemos uma pesquisa mais aprofundada entre os diversos problemas, este foi aquele que mais nos apelou.

Como Resolver?

[voltar ao índice]

Ajuste de Pesos

Uma maneira de resolver este gravíssimo problema é **associar pesos a instâncias das classes**, de forma a atribuir mais importância a classes minoritárias, como tentativa de equilibrar o facto de se encontrarem em **desvantagem numérica**.

Ao ajustar os pesos das instâncias, garantimos que as amostras de classes minoritárias tenham uma **maior contribuição** na decisão final do algoritmo. Isto significa que, mesmo que as instâncias de uma classe minoritária sejam numericamente menores, elas podem exercer uma **influência proporcionalmente maior** na classificação, evitando que a classe correta seja descartada injustamente.

Por exemplo, se os k vizinhos mais próximos consistem em três instâncias da classe maioritária e três da classe minoritária, o ajuste de pesos pode garantir que as instâncias minoritárias, ao receberem pesos maiores, tenham um **impacto maior** na determinação da classe final do novo exemplo.

Distâncias Ponderadas

Decidimos também implementar o conceito de **distâncias ponderadas**, onde cada vizinho contribui para a decisão da classe com um peso proporcional à inversa da sua distância ao novo exemplo. Desta forma, **os vizinhos mais próximos** têm um impacto maior na classificação do que os vizinhos mais afastados. Este ajuste na ponderação das distâncias permite que o algoritmo leve em consideração não apenas a proximidade absoluta, mas também a **relevância relativa** de cada vizinho.

Assim, por exemplo, se os vizinhos mais próximos consistirem em três instâncias da classe maioritária e três instâncias da classe minoritária, as instâncias da classe minoritária contribuirão mais para a decisão de classificação, mesmo que sejam menos numerosas. Isto ajuda a evitar que a classe correta seja descartada simplesmente por ser minoritária.

Em resumo, ao implementar distâncias ponderadas no algoritmo k-NN, podemos **mitigar os efeitos prejudiciais** das classes desbalanceadas, garantindo que a classificação leva em consideração tanto a proximidade dos vizinhos quanto a distribuição das classes no conjunto de dados, promovendo a criação de um modelo mais justo e preciso, capaz de lidar de maneira eficaz com conjuntos de dados desbalanceados.

Ao utilizarmos o ajuste de pesos e as distâncias ponderadas, podemos ajustar a **sensibilidade do algoritmo** ao desequilíbrio entre as classes, permitindo um melhor **equilíbrio entre a precisão** na classificação das classes minoritárias e a capacidade de **generalização** para todas as classes.

Testar o Algoritmo nos Datasets

[voltar ao índice]

1. Leitura e Análise dos Datasets

Para avaliar a eficiência das funções acrescentadas, é necessário ter um ponto de partida relativamente ao desempenho do algoritmo em questão, sem métodos de resolução dos problemas. Primeiro, passamos os dados csv para um dataframe.

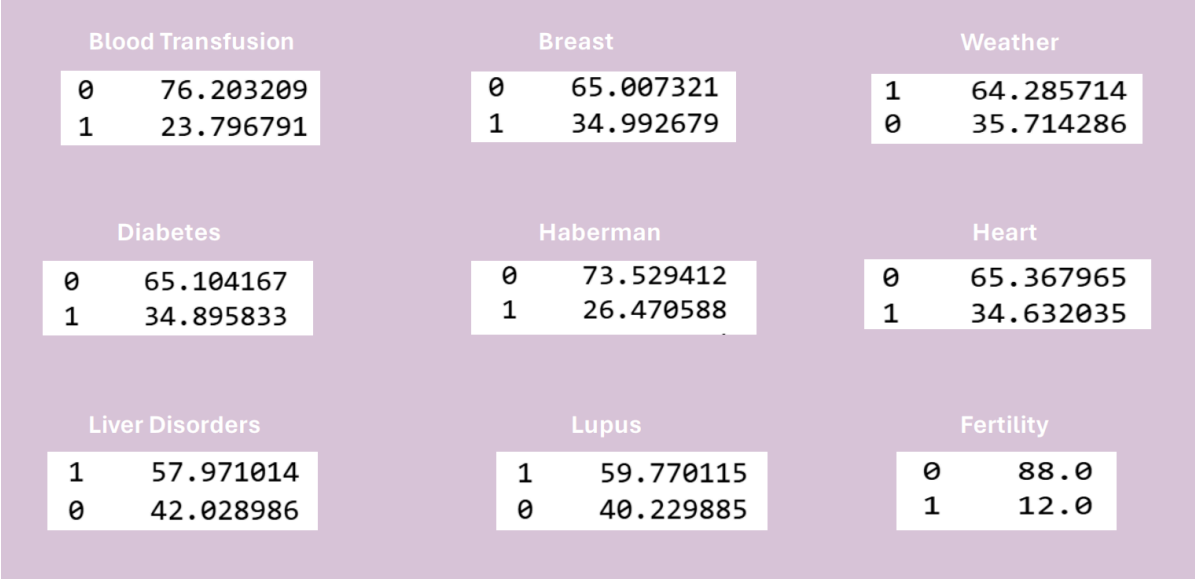
In [1]:
```
from IPython.display import Image
caminho_imagem = 'd.png'
Image(filename=caminho_imagem)
```

Out[1]:

Para uma análise mais detalhada criamos um jupyter apenas com o pré-processamento de dados dos nossos datasets. No entanto, vamos agora observar apenas as proporções dos nossos datasets.

In [2]:
```
caminho_imagem = 'p.png'
Image(filename=caminho_imagem)
```

Out[2]:



2. Divisão entre Attributes e Target

De seguida, preparamos um split para os dados e inicializar, treinar e testar o classificador **K-nn** para cada um dos datasets representados.

3. Teste do código nos datasets

Uma vez lidos e divididos em features e target usamos funções que nos permitem automatizar o processo de classificação e visualização da accuracy.

In [3]:
```
caminho_imagem = 'accuracy.png'
Image(filename=caminho_imagem)
```

Out[3]:

```
1. blood_transfusion performance:

Accuracy: 0.7777777777777778

---------------------------

2. breast_w  performance:

Accuracy: 0.9658536585365853

---------------------------

3. diabetes performance:

Accuracy: 0.7532467532467533

---------------------------

4. fertility performance:

Accuracy: 0.8666666666666667

---------------------------

5. haberman performance:

Accuracy: 0.7282608695652174

---------------------------

6. liver_disorders performance:

Accuracy: 0.6153846153846154

---------------------------

7. lupus performance:

Accuracy: 0.7777777777777778

---------------------------

8. sa_heart performance:

Accuracy: 0.6115107913669064

---------------------------

9. weather performance:

Accuracy: 0.6

---------------------------
```

Para que se possam compreender melhor os resultados devemos saber as seguintes noções:

Accuracy

Alta: Indica que uma grande proporção das previsões do modelo está correta. O modelo é geralmente eficaz em distinguir entre as classes.

Baixa: Sugere que o modelo comete muitos erros ao fazer previsões. Pode indicar que o modelo não é adequado para os dados ou está mal configurado.

Conclusão

A análise dos datasets foi realizada várias vezes durante testes por nós executados, no entanto de modo a manter o Notebook fácil de compreender apenas deixamos uma dessas execuções. Contudo, devemos ter em conta que para alguns splits que não se encontram aqui representados os valores da accuracy variaram bastante, dependendo estes da divisão dos dados.

Para além do valor anormal do dataset 'breast_w', excecionalmente alto, os resultados obtidos pelo KNN parecem estar de acordo com os datasets que apresentam classes desbalanceadas. Devemos também mencionar que nos diversos testes realizados a accuracy de 'Weather' apresentou valores muito distintos, provavelmente devido à escassez de exemplos em certos splits. Pensamos também que a anormalidade de 'breast_w' se deve ao facto das classes estarem bem divididas, como foi possivel observar no Jupyter Notebook do Tratamento de Dados.

Como Melhorar

[voltar ao índice]

Para este projeto, foram escolhidos 2 modos de lidar com o problema de **classes desbalanceadas** que identificámos em todos os datasets em questão, provenientes de OpenML:

- **Ajuste de peso das classes**
- **Ajuste da distância**

Deste modo, vamos implementar estas medidas e analisar a sua eficiência.

`Notas Importantes:`

- É possível que os datasets recolhidos **apresentem outros tipos de complicações**, tais como classes sobrepostas ou atributos irrelevantes. **Segundo as instruções do guião e após confirmação com docente**, vamos apenas explicar que apesar de as nossas funções estarem corretamente implementadas, os resultados podem não ser ideais, devido às complicações mencionadas anteriormente.

Melhoria do Algoritmo

[voltar ao índice]

4.1. Ajuste de Peso

De forma a equilibrar o impacto das instâncias de ambas as classes, é calculado, `durante o processo de treino do modelo`, que peso dar a cada classe, sendo que é dado um **peso maior** às instâncias da classe **minoritária**, de modo a que esta passe a ter uma maior

relevância e influência. Após serem escolhidos os k vizinhos mais próximos usamos os pesos para nos ajudar a decidir qual a classificação correta.

4.2. Ajuste da distância

Neste método utilizamos a mesma técnica de cálculo de pesos utilizada no 'Ajuste de peso das classes', mas em vez de estes serem atribuidos após a escolha dos k vizinhos mais próximos, os pesos são multiplicados pelas distâncias a que os dados se encontram uns dos outros, de modo a influenciar a relevância de todos os dados, sendo assim os k vizinhos escolhidos tendo em conta o peso e a distância.

Nesta parte do projeto o objetivo é melhorar o código do KNN para o problema das classes desbalanceadas.

1. Melhoria K-nearest Neighbors - Pesos

[voltar a Melhoria do Algoritmo]

Relativamente a BaseEstimator **não serão realizadas quaisquer alterações** uma vez que esta classe tem como único propósito ver se os dados estão no formato correto para serem utilizados e caso não estejam convertê-los para tal.

Para que posteriormente possam ser realizados testes com o KNN sem melhoria e após realizar alterações, serão mudados os nomes das classes, para que não haja dúvidas de quais estão a ser chamadas. Reforçamos que BaseEstimator não sofreu alterações, logo será utilizado tanto para o KNN melhorado como para o antigo, não havendo, portanto, a necessidade de alterar o seu nome.

Mudança dos nomes:

Antigo		Novo
KNNBase	->	KNNBase_P
KNNClassifier	->	KNNClassifier_P
KNNRegressor	->	KNNRegressor_P

Com o objetivo de lidar com as classes desbalanceadas, vamos aplicar atribuições de pesos a classes, com inspiração no método da **Ponderação Equilibrada de Classes (BWC)**.

-> Ponderação Equilibrada de Classes: é uma técnica utilizada em problemas de classificação para lidar com conjuntos de dados desbalanceados. É uma abordagem que visa **equilibrar o impacto das diferentes classes**, atribuindo **pesos apropriados durante o processo de classificação**, de modo que classes **menos frequentes tenham um peso maior** do que as classes mais frequentes. A seguinte fórmula proporciona uma boa métrica para avaliar qual o peso a ser atribuido às classes:

```
Peso da classe = Número total de amostras / (Número de
classes * Número de amostras daquela classe)
```

Decidimos utilizar o BWC para a atribuição de pesos uma vez que esta técnica os ajusta de forma proporcional e equilibrada, pois considera não só a frequência de cada classe como o número total de classes. Para além disso, a fórmula normaliza os pesos de modo a que seja possivel evitar pesos excessivamente grandes ou pequenos, mantendo a influência de cada classe numa escala mais uniforme, algo que não aconteceria se usassemos a Ponderação Inversa da Frequência de Classe (IFW).

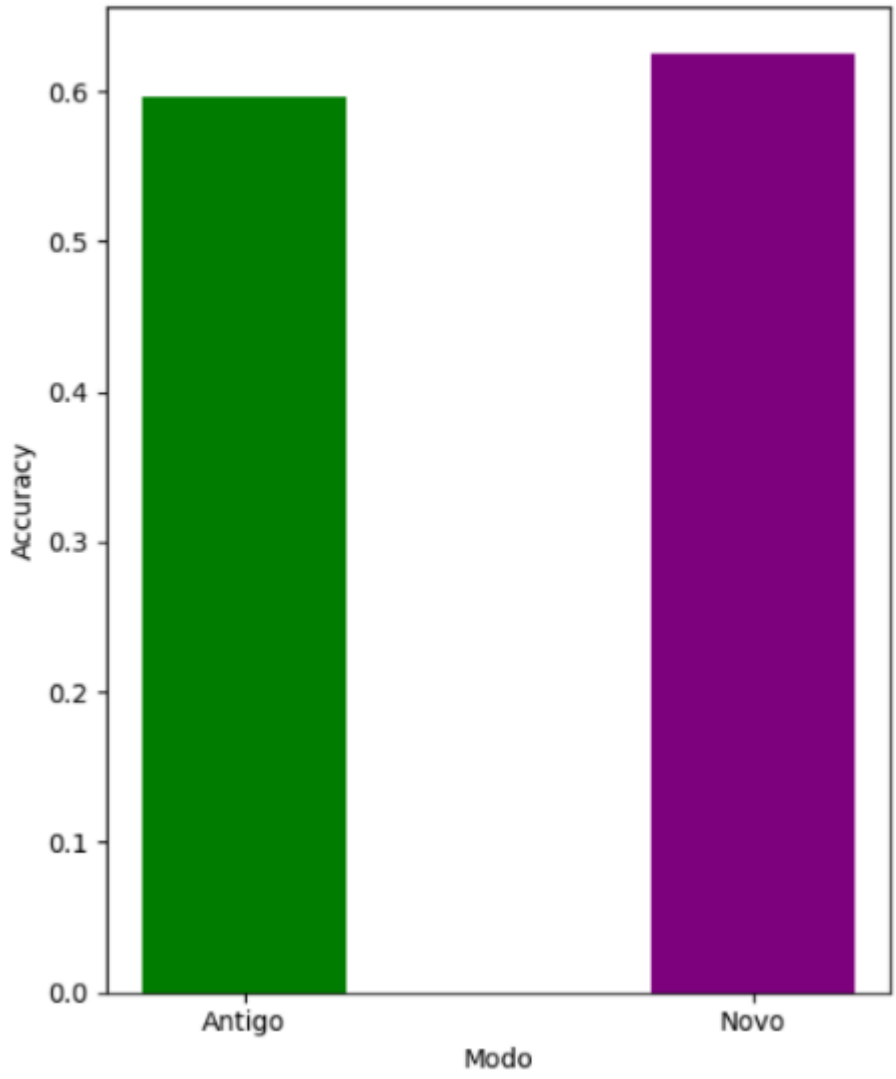
Vamos então fazer alterações ao código incluindo a ponderação de pesos. De um modo geral o funcionamento será o seguinte:

```
- Quando se usa 'fit' a função vai ser ativada em KNNClassifier_P,
mas vai buscar na mesma o 'fit' no BaseEstimator, ou seja, até aqui
não haveriam alterações. A única diferença é que no 'fit' do
KNNClassifier_P vão ser também calculados os pesos que se devem dar
a cada classe usando a função calculate_class_weights que funciona
de acordo com o BWC.

- Durante a classificação, ou seja, quando se ativa a função
predict, o código irá funcionar normalmente, isto é, calcula quais
os pontos mais próximos ao que estamos a avaliar. No entanto,
quando passar para 'aggregate' do KNNClassifier_P, em vez de
escolher o 'label' mais comum, irá funcionar do seguinte modo:

    1. Calcula o número de vezes que cada 'label' aparece
    2. Multiplica o peso de cada classe pelo número de vezes que
esta apareceu
    3. Escolhe o label com maior pontuação.
```

De modo a perceber se o algoritmo melhorou, fizemos um teste rápido com apenas uma iteração.

In [4]:
```
caminho_imagem = 'tpesos.png'
Image(filename=caminho_imagem)
```

Out[4]:
```
0.5961538461538461
0.625
```



2. Melhoria K-nearest Neighbors - Distâncias Ponderadas

[voltar a Melhoria do Algoritmo]

Iremos agora testar a técnica das **Distâncias Ponderadas**. Tal como anteriormente o 'BaseEstimator' não irá sofrer nenhuma alteração, no entanto iremos mudar os nomes às restantes classes para que não haja dúvidas de quais estamos a utilizar.

Mudança dos nomes:

Antigo		Novo
KNNBase	->	KNNBase_D
KNNClassifier	->	KNNClassifier_D
KNNRegressor	->	KNNRegressor_D

Como funciona:

```
- Começamos por contar as ocorrências de cada classe no conjunto de
dados. Isto geralmente é feito contando quantas vezes cada classe
aparece nos rótulos de destino.
```

- Em seguida, é calculado um peso para cada classe. Utilizamos também o BWC.

- Durante a classificação, as distâncias entre o ponto a ser classificado e os pontos de treinamento são calculadas. Essas distâncias são então ponderadas pelos pesos das classes, de modo que as distâncias para classes menos frequentes tenham um impacto maior na classificação final.

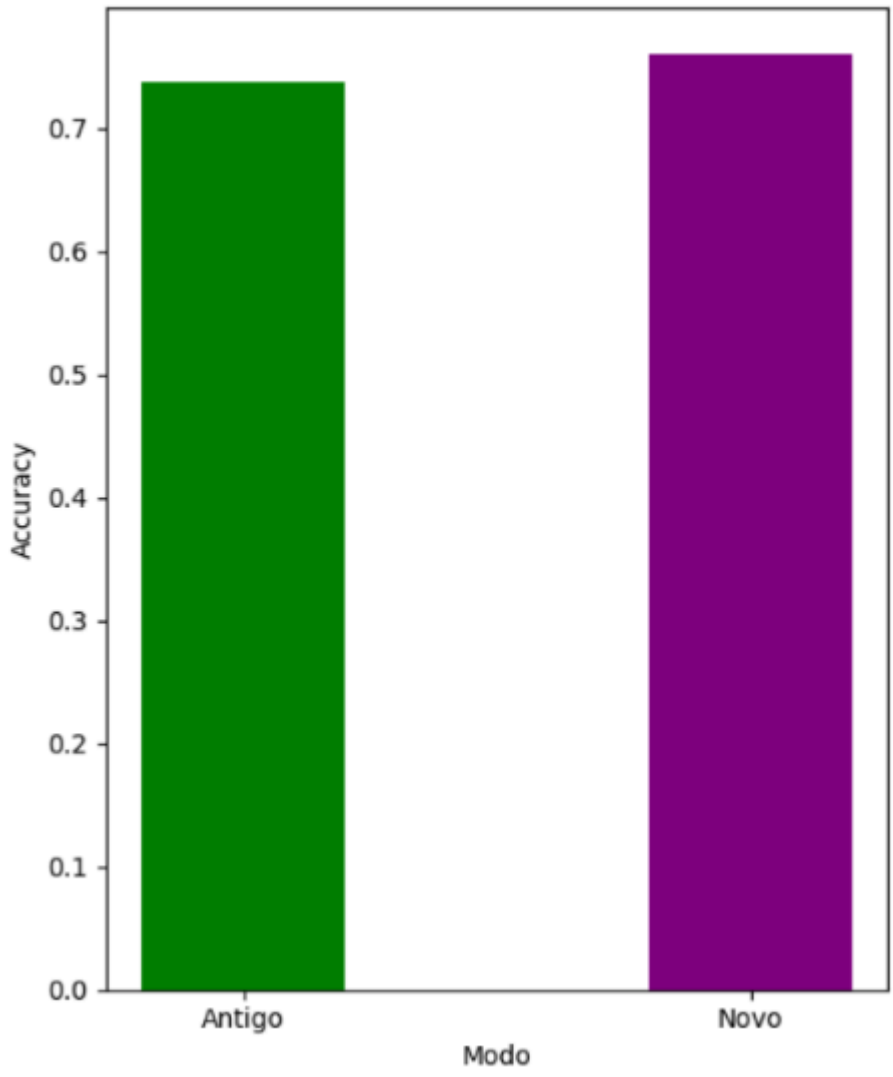
Distancia * Peso da sua classe

- As distâncias ponderadas são organizadas e escolhem-se as k menores. Entre essas escolhe-se a classificação que se repetir mais vezes.

Novamente, de modo a perceber se o algoritmo melhorou, vamos fazer um teste rápido com apenas uma iteração.

In [5]:
```
caminho_imagem = 'tdistancias.png'
Image(filename=caminho_imagem)
```

Out[5]:
```
0.7377777777777778
0.76
```



Avaliação do Algoritmo

[voltar ao índice]

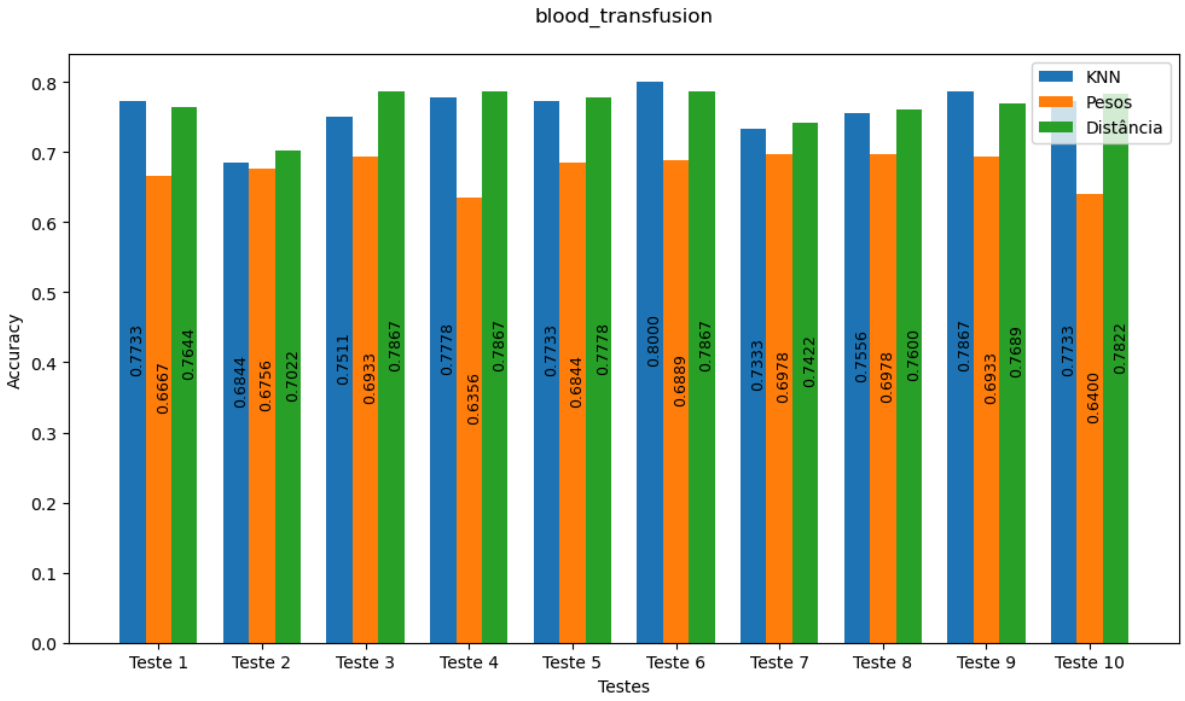
Para analisar graficamente o desempenho de todos os KNN, tanto no original como nos modificados, utilizamos dois tipos de gráficos que nos permitem tirar conclusões. Destacamos que para cada teste é criado um split dos dados aleatório, no entanto, dentro de cada teste é utilizado o mesmo 'split' de modo a garantir que os resultados sejam obtidos corretamente, sendo que essa divisão é utilizada nos três algoritmos.

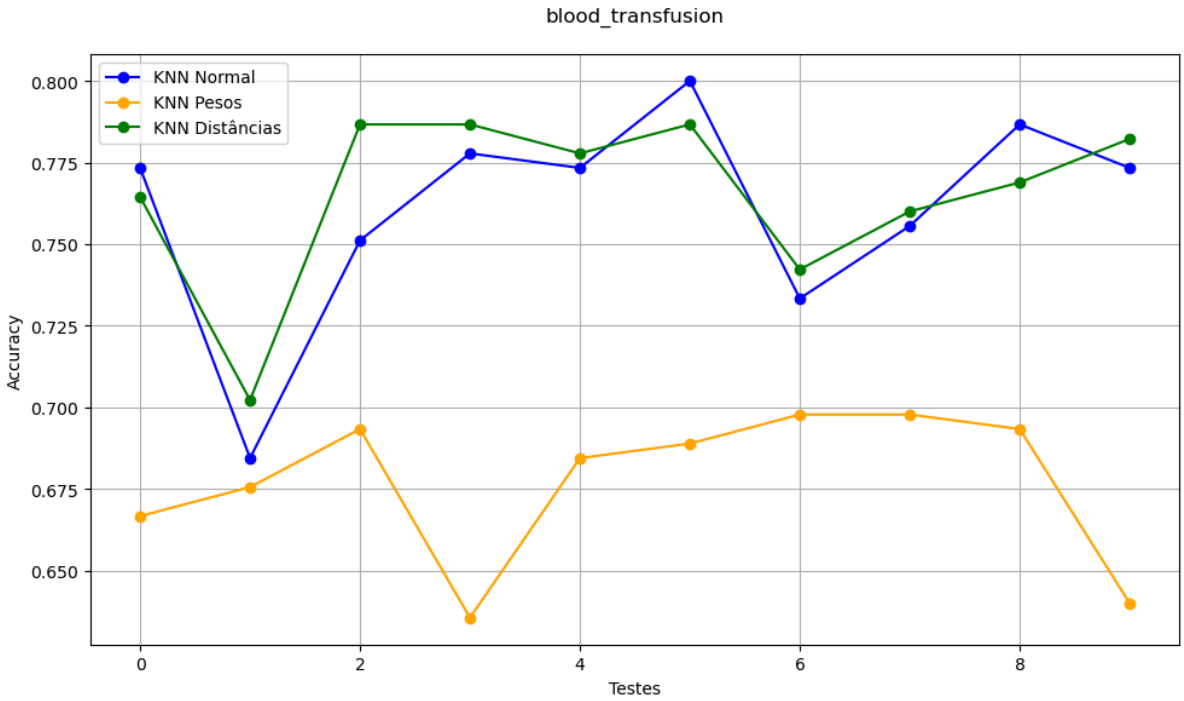
Testes

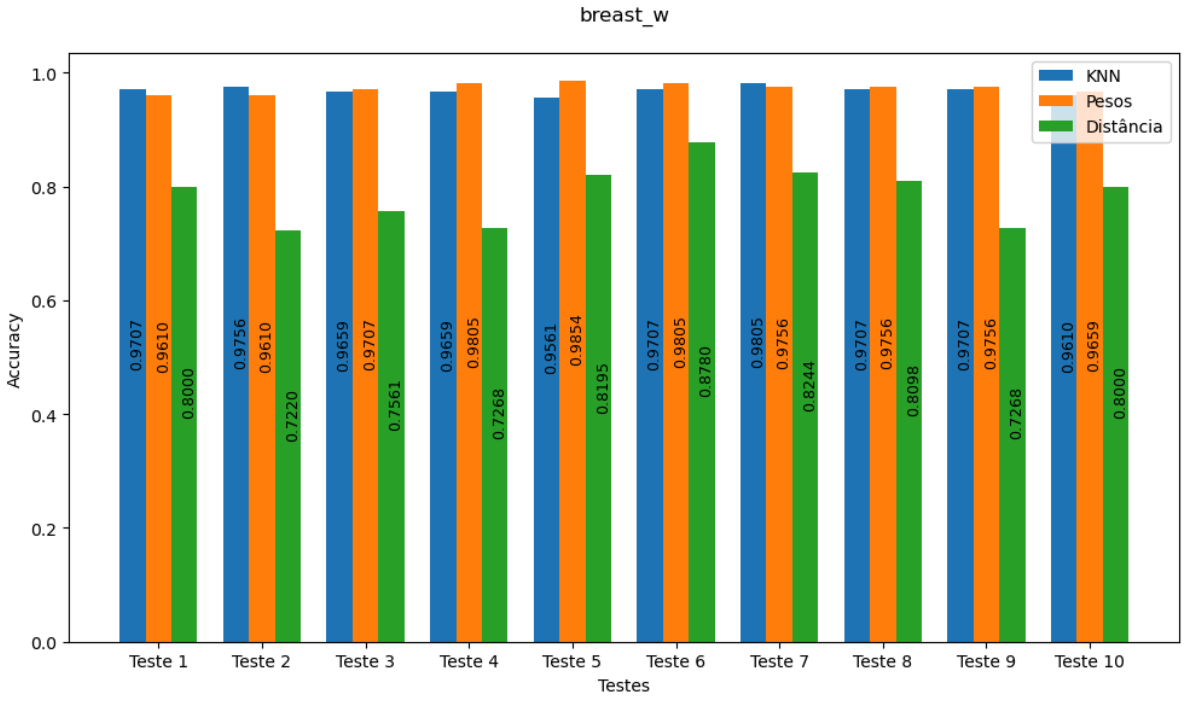
[voltar a Avaliação do Algoritmo]

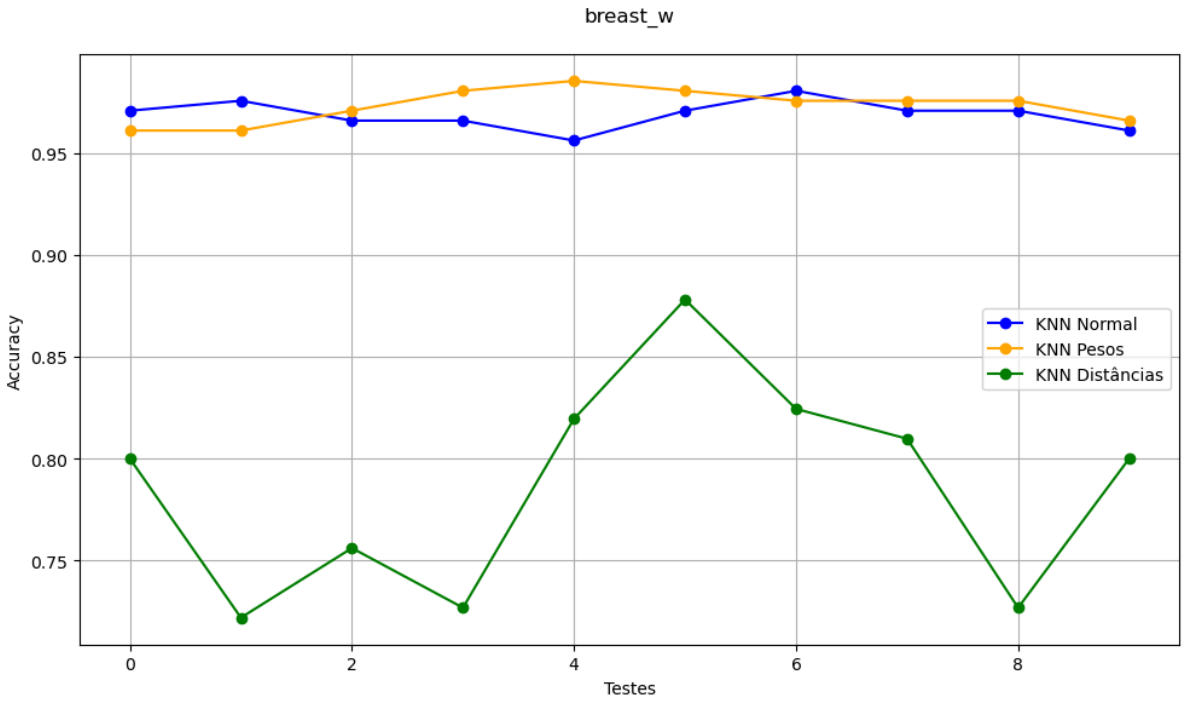
Durante as fases de teste foram realizadas mais do que 10 iterações, contudo decidimos deixar aquela que consideramos que retrata uma melhor representação dos efeitos dos algoritmos para datasets que podem revelar mais que uma complicação em simultâneo.

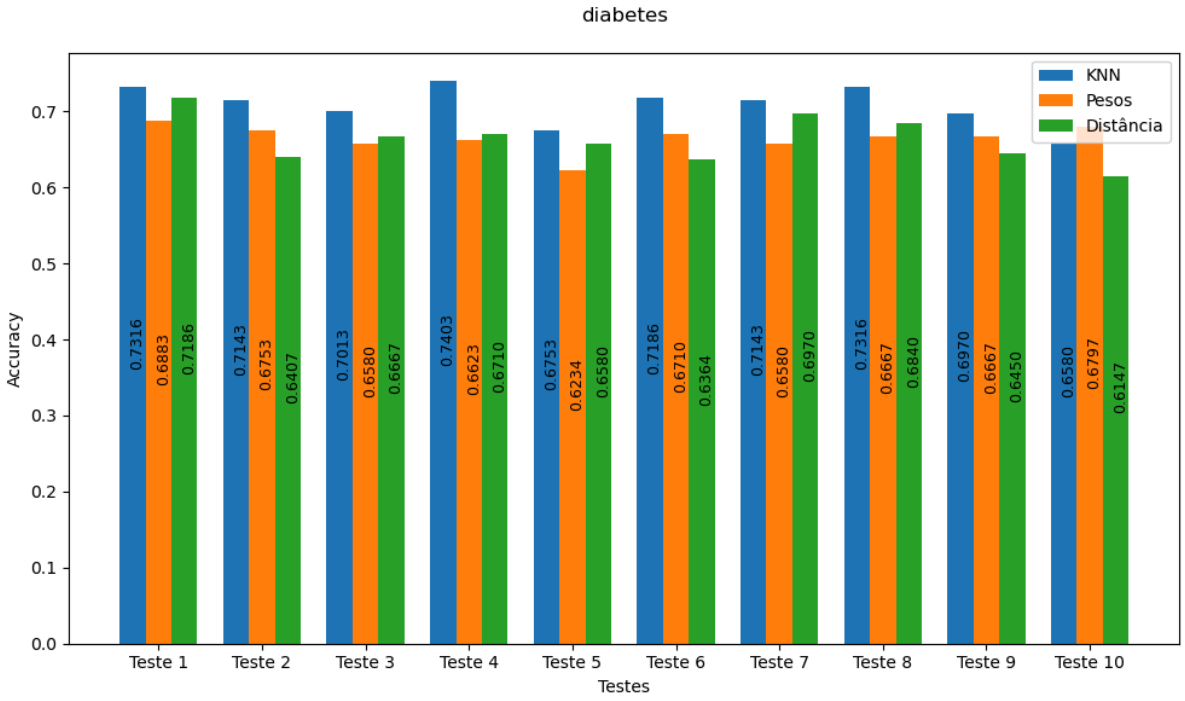
In [34]:
```
test(split_data, ds, n_tests = 10)
```

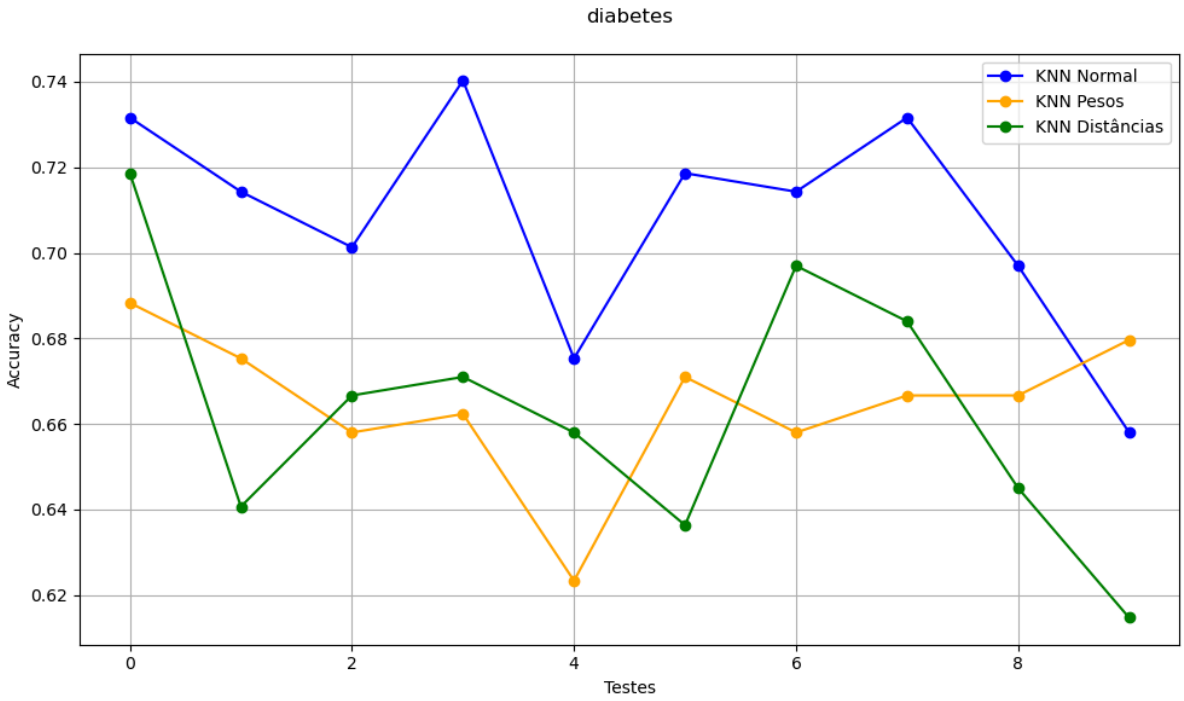


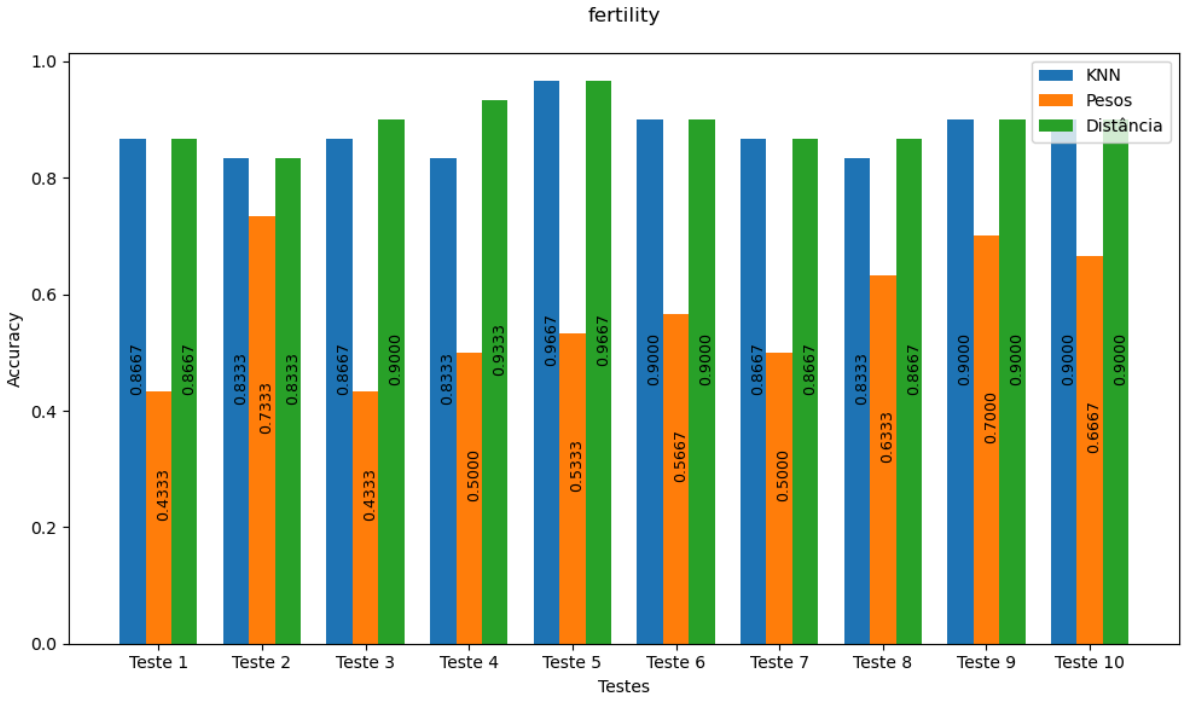

blood_transfusion
KNN Normal
KNN Pesos
KNN Distâncias
Accuracy
Testes
0.800
0.775
0.750
0.725
0.700
0.675
0.650
0
2
4
6
8


breast_w
KNN
Pesos
Distância
Accuracy
Testes
Teste 1
Teste 2
Teste 3
Teste 4
Teste 5
Teste 6
Teste 7
Teste 8
Teste 9
Teste 10
0.9707 0.9610 0.8000
0.9756 0.9610 0.7220
0.9659 0.9707 0.7561
0.9659 0.9805 0.7268
0.9561 0.9854 0.8195
0.9707 0.9805 0.8780
0.9805 0.9756 0.8244
0.9707 0.9756 0.8098
0.9707 0.9756 0.7268
0.9610 0.9659 0.8000

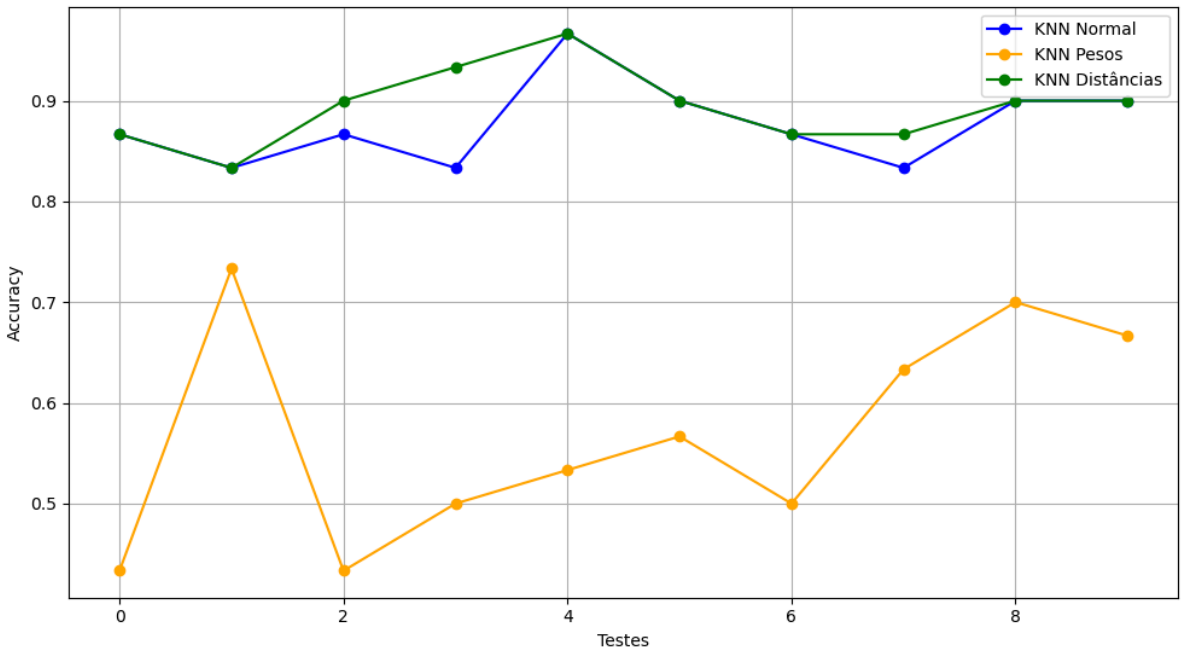

breast_w
Accuracy
Testes
KNN Normal
KNN Pesos
KNN Distâncias


diabetes
Accuracy
Testes
KNN
Pesos
Distância
Teste 1
Teste 2
Teste 3
Teste 4
Teste 5
Teste 6
Teste 7
Teste 8
Teste 9
Teste 10

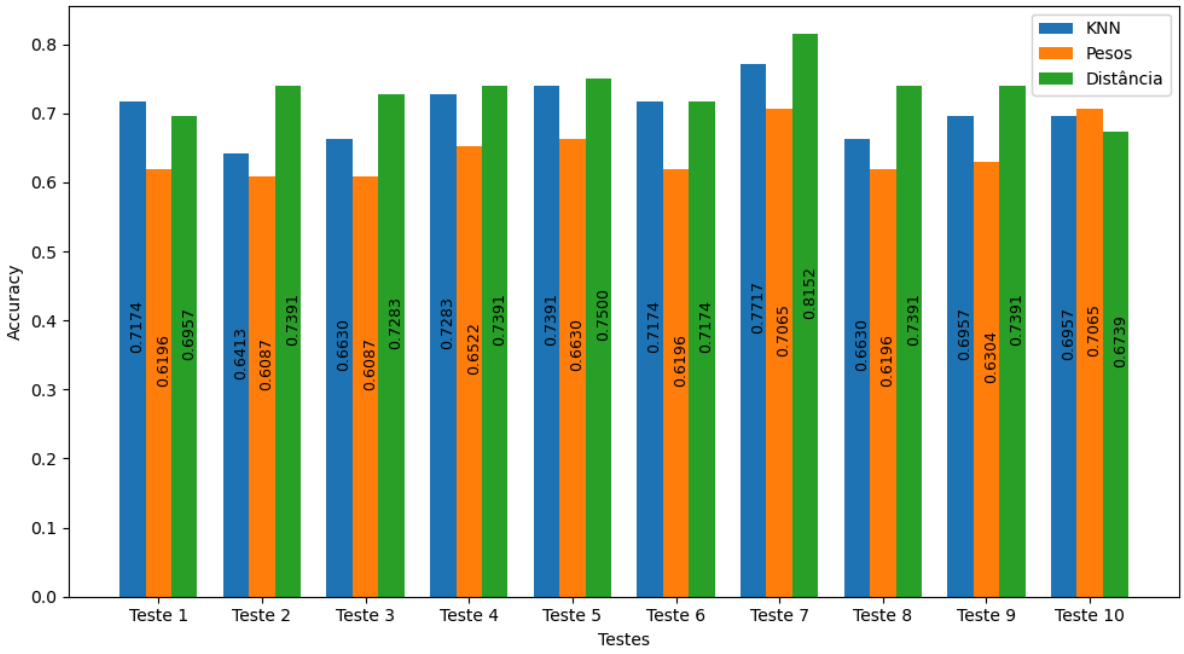

diabetes
KNN Normal
KNN Pesos
KNN Distâncias
Accuracy
Testes
0.74
0.72
0.70
0.68
0.66
0.64
0.62
0
2
4
6
8


fertility
KNN
Pesos
Distância
Accuracy
Testes
Teste 1
Teste 2
Teste 3
Teste 4
Teste 5
Teste 6
Teste 7
Teste 8
Teste 9
Teste 10
0.8667
0.4333
0.8667
0.8333
0.7333
0.8333
0.8667
0.4333
0.9000
0.8333
0.5000
0.9333
0.9667
0.5333
0.9667
0.9000
0.5667
0.9000
0.8667
0.5000
0.8667
0.8333
0.6333
0.8667
0.9000
0.7000
0.9000
0.9000
0.6667
0.9000

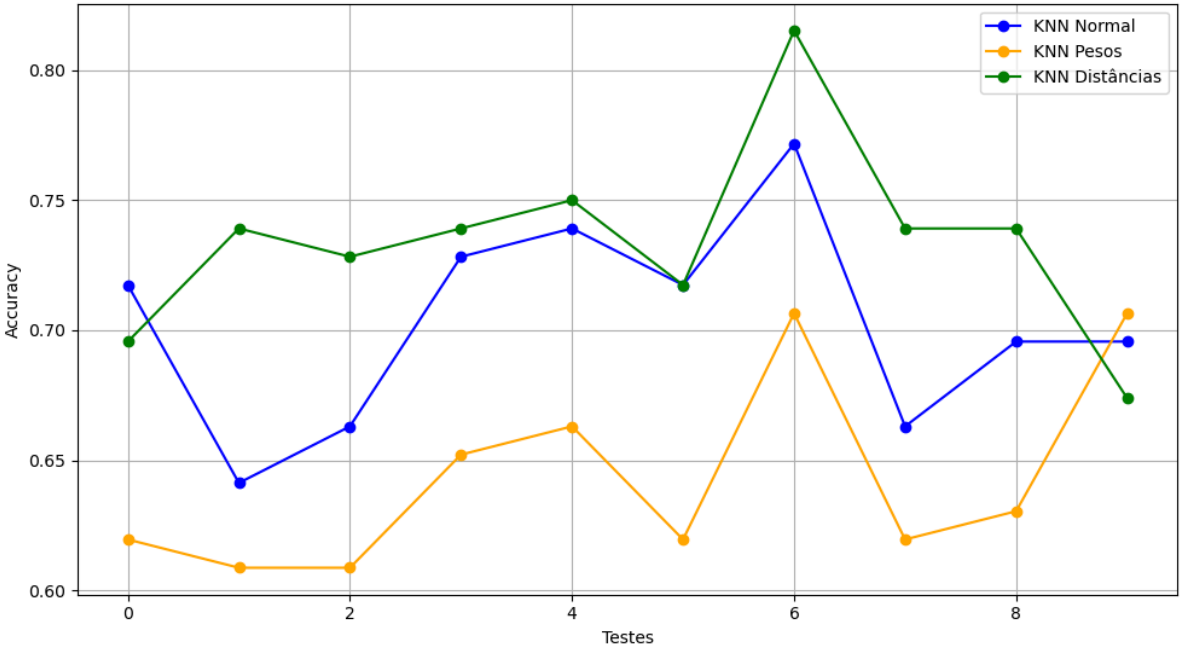
fertility



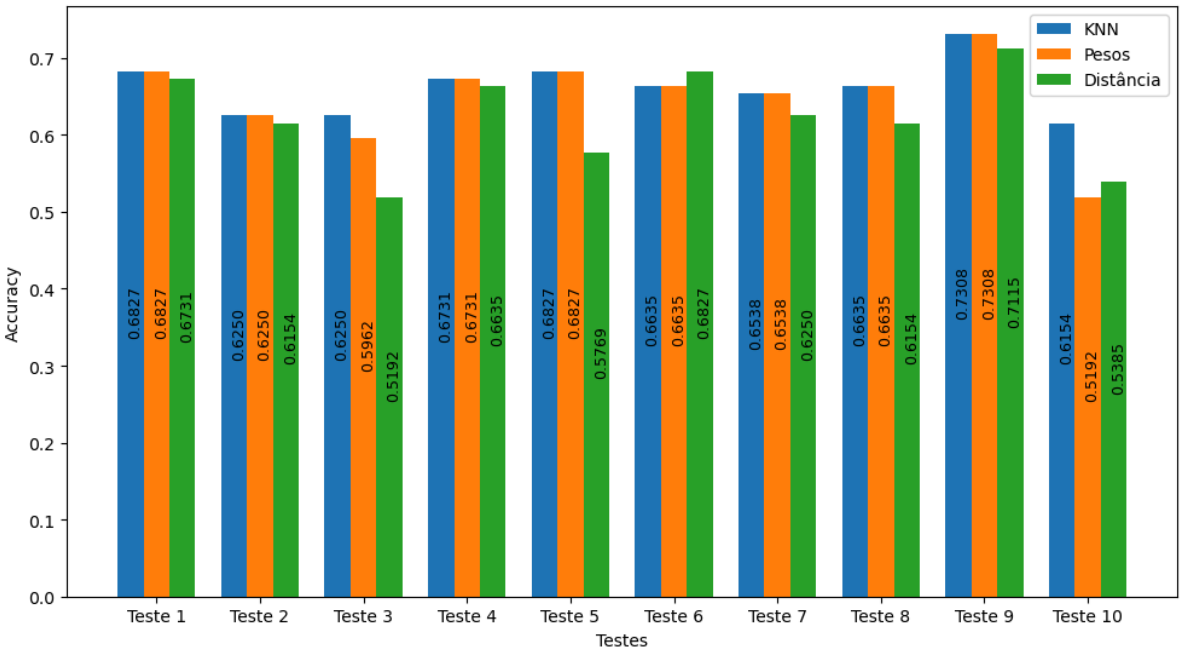
haberman



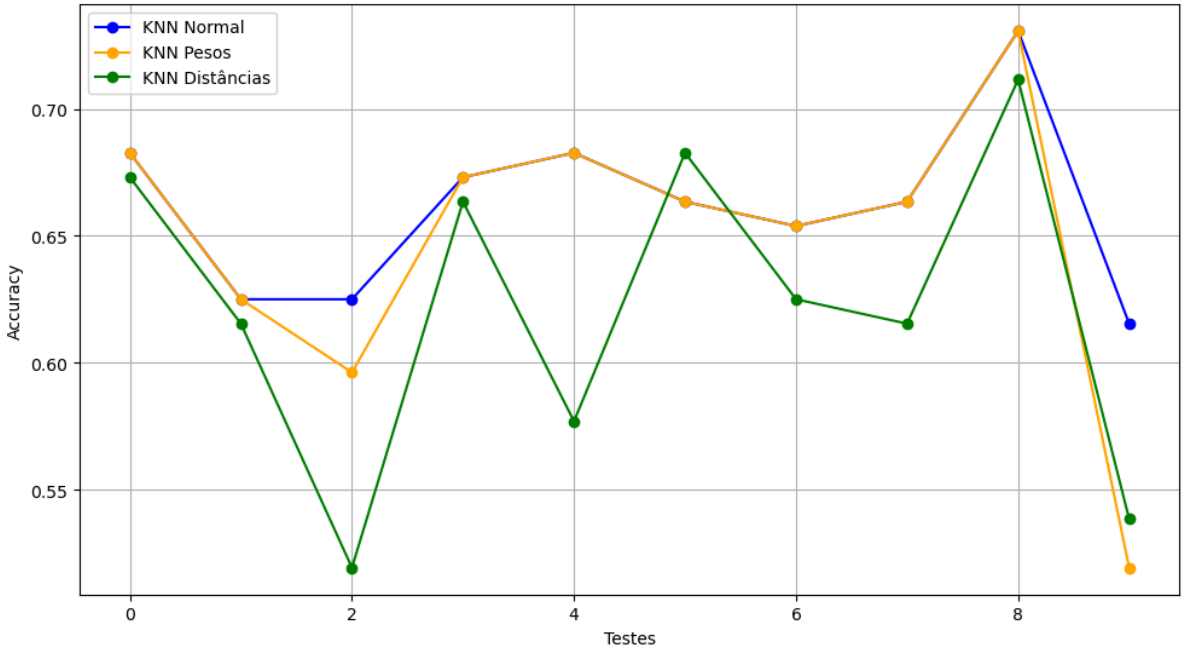
haberman



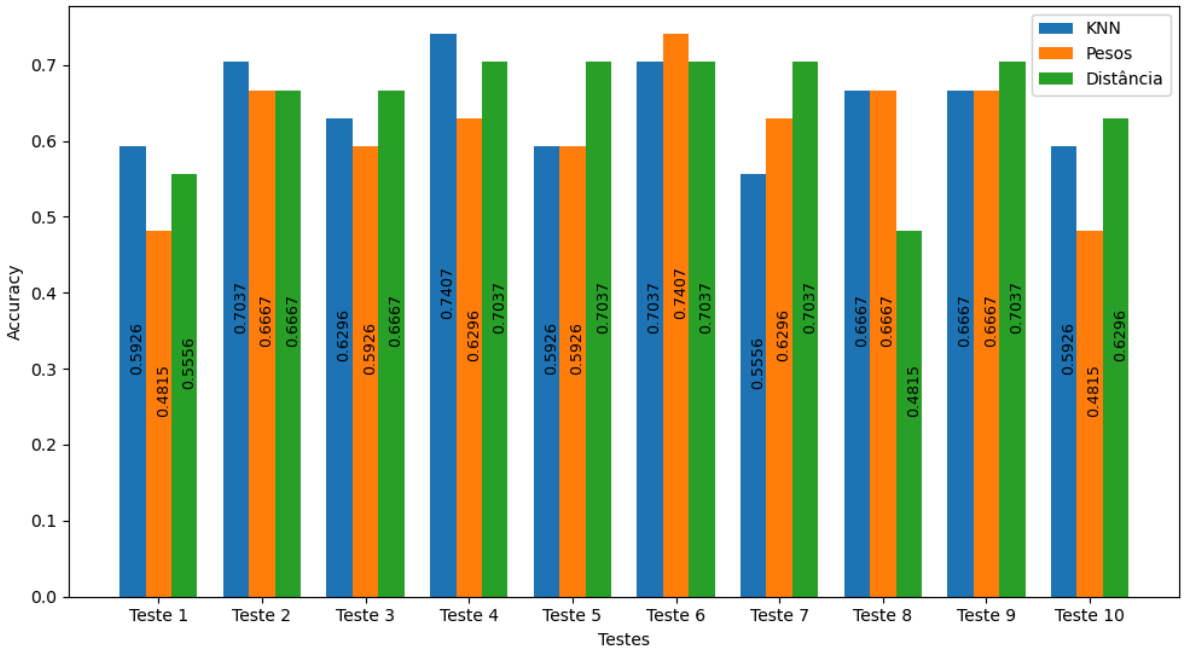
liver_disorders

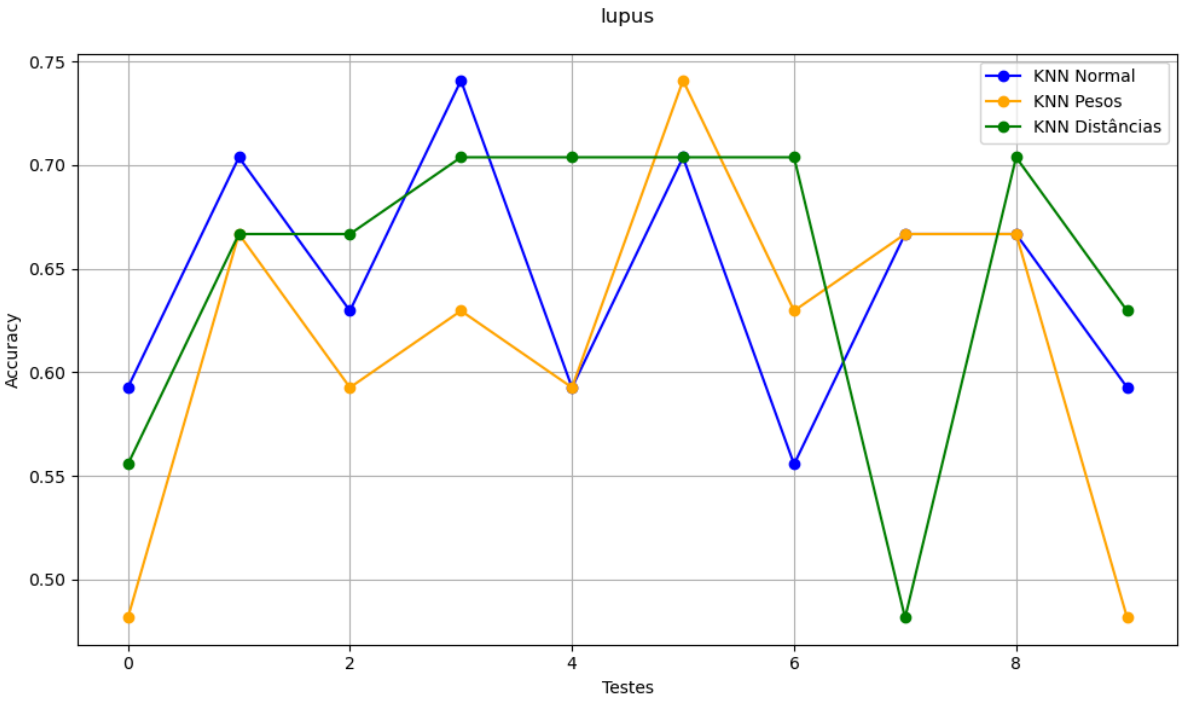


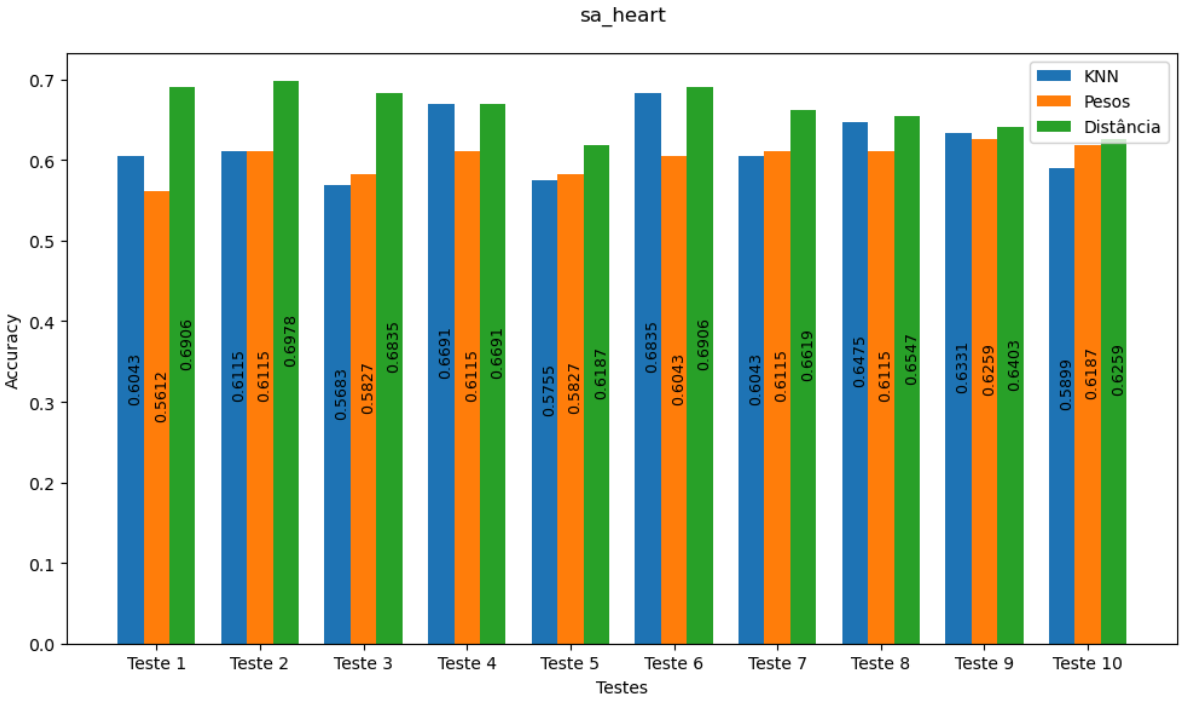
liver_disorders

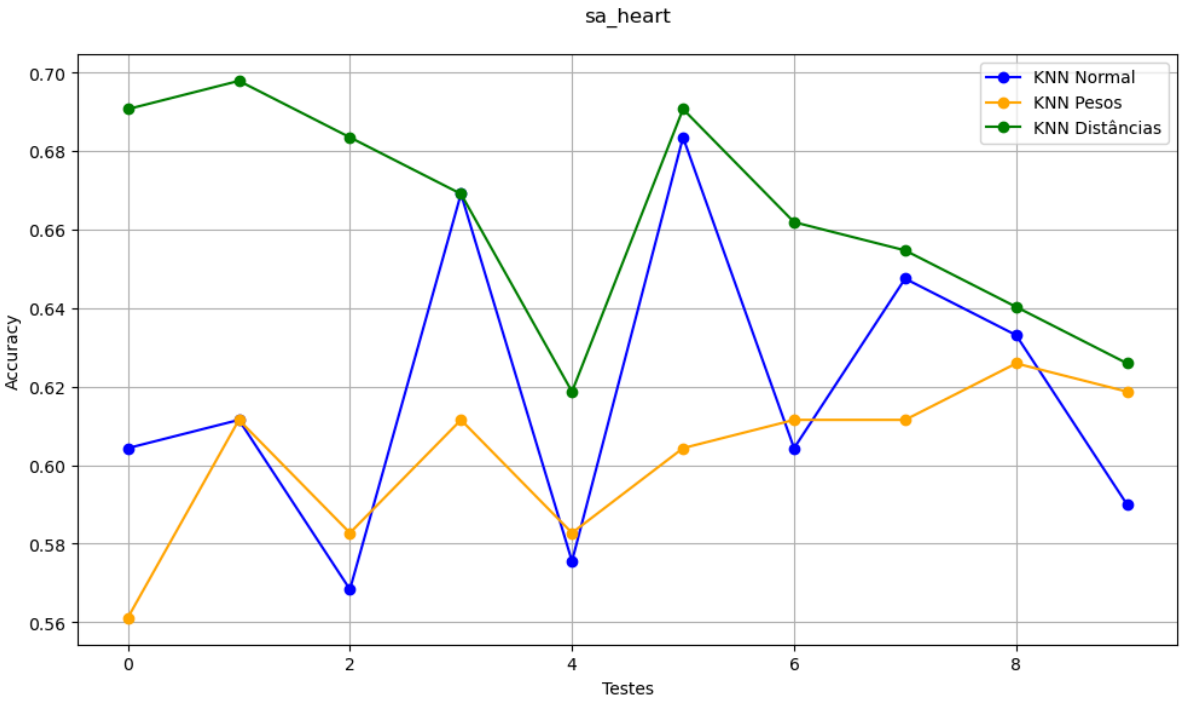


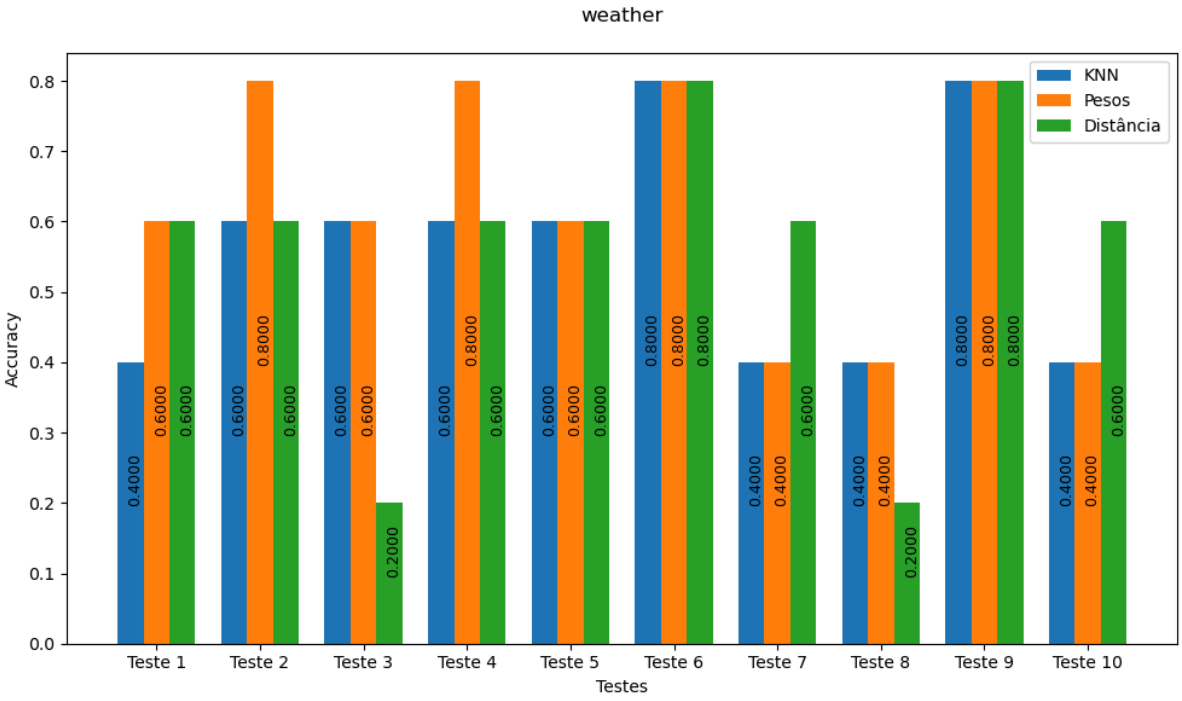
lupus

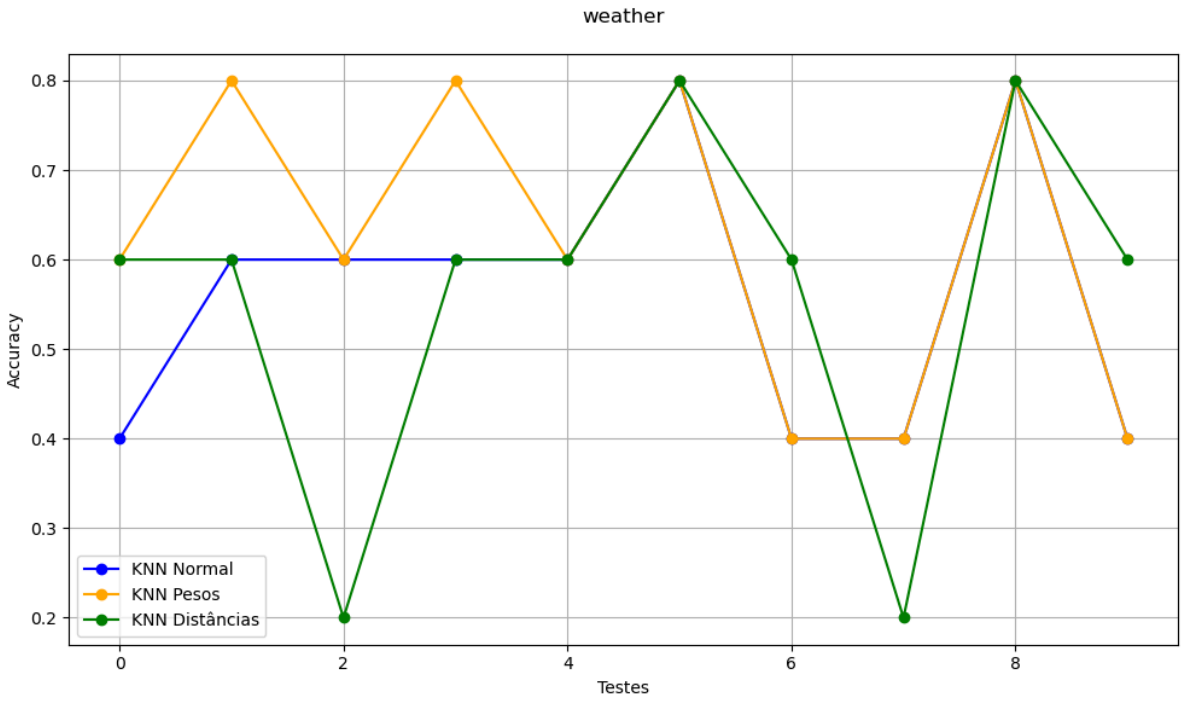



lupus
KNN Normal
KNN Pesos
KNN Distâncias
Accuracy
Testes
0.75
0.70
0.65
0.60
0.55
0.50
0
2
4
6
8


sa_heart
KNN
Pesos
Distância
Accuracy
Testes
Teste 1
Teste 2
Teste 3
Teste 4
Teste 5
Teste 6
Teste 7
Teste 8
Teste 9
Teste 10
0.6043 0.5612 0.6906
0.6115 0.6115 0.6978
0.5683 0.5827 0.6835
0.6691 0.6115 0.6691
0.5755 0.5827 0.6187
0.6835 0.6043 0.6906
0.6043 0.6115 0.6619
0.6475 0.6115 0.6547
0.6331 0.6259 0.6403
0.5899 0.6187 0.6259


sa_heart
KNN Normal
KNN Pesos
KNN Distâncias
Accuracy
Testes
0.70
0.68
0.66
0.64
0.62
0.60
0.58
0.56
0
2
4
6
8


weather
KNN
Pesos
Distância
Accuracy
Testes
Teste 1
Teste 2
Teste 3
Teste 4
Teste 5
Teste 6
Teste 7
Teste 8
Teste 9
Teste 10
0.4000
0.6000
0.6000
0.6000
0.8000
0.6000
0.6000
0.6000
0.2000
0.6000
0.8000
0.6000
0.6000
0.6000
0.6000
0.8000
0.8000
0.8000
0.4000
0.4000
0.6000
0.4000
0.4000
0.2000
0.8000
0.8000
0.8000
0.4000
0.4000
0.6000



Análise dos Resultados

[voltar a Avaliação do Algoritmo]

Ao analisarmos os gráficos percebemos que os algoritmos que sofreram melhorias apresentaram resultados mais satisfatórios em comparação ao KNN original, isto é, demonstraram-se melhores.

No entanto, em muitos casos as 'Distâncias Ponderadas' obtiveram melhores resultados que os 'Pesos', o que nos levou a questionar porque é que tal acontecia. De modo a compreender quais obtiveram melhores resultados observe a seguinte tabela:

Distâncias Ponderadas	Pesos
Blood Transfusion	Breast_w
Fertility	
Haberman	
Lupus	
Sa_Heart	

A escolha entre distâncias ponderadas e peso de classes depende do contexto específico. Em geral, distâncias ponderadas são melhores quando a distribuição local dos dados é crucial para a classificação, enquanto o peso de classes é mais eficaz para corrigir desequilíbrios globais. Ao observamos graficamente os nossos datasets chegamos à conclusão que em muitos deles havia algum tipo de sobreposição, isto é, as classes encontram-se muito 'misturadas', logo, aplicar medidas que alteram a posição dos dados acaba por amenizar muitas das sobreposições encontradas.

Observamos também que existem alguns casos excecionais, tais como:

- No dataset **'Diabetes'** o knn original demonstrou-se melhor que os restantes algoritmos, sendo que a sua complicação dominante é a de **classes sobrepostas**. Fomos capazes de determinar isto após confirmar que a proporção entre classes não é extremamente desbalanceada contrariamente à forte sobreposição dos dados claramente representada nos scaterplots. Por este motivo, como não existe uma distinção clara entre classes, atribuir pesos aos vizinhos pode na realidade afastá-los em vez de ajudar a detetar qual a resposta correta.

- **'Liver Disorders'** representa um dataset de controlo, ou seja, não é propriamente desbalanceado, tal escolha foi feita com o intuito de mostrar que nem sempre é benéfico usar as otimizações que desenvolvemos. Neste caso, as 'distâncias ponderadas' provaram afetar negativamente o processo de classificação do algoritmo, o que seria de esperar uma vez que este pode afastar ou aproximar valores indevidamente, principalmente se os dados estiverem muito próximos uns dos outros, como vimos nos gráficos. Contudo, a utilização de 'pesos' não demonstrou classificações muito diferentes das do knn original, o que também seria de prever tendo em conta o seu funcionamento. Após escolher os k valores mais próximos, a atribuição de pesos é praticamente insignificante, tal acontece já que os seus valores são proporcionais à frequência dos dados.

- A escassez dos dados, aliada ao seu desbalanceamento, fazem com que o dataset **'Weather'** seja dificil de classificar. A aplicação de medidas de resolução do problema revelou-se, por isso, especialmente eficaz.

Tendo em conta as análises realizadas concluimos que os knn melhorados, seja com pesos ou distâncias ponderadas, apenas obtêm melhores resultados se as classes forem desbalanceadas, dependendo adicionalmente da disposição dos dados.

Conclusão

[voltar ao índice]

Através da elaboração deste projeto, fomos capazes de desenvolver competências úteis para futuras análises de dados, otimizações de algoritmos já conhecidos e aprimorar a nossa capacidade de reconhecimento de complicações de datasets e possíveis soluções. Posto isto, conseguimos:

- Identificar os vários tipos de complicações que podem afetar o algoritmo escolhido, **K-nn**, e sucessivamente decidir atacar o problema das **classes desbalanceadas**;

- Proceder a um cuidadoso tratamento dos datasets recolhidos da plataforma indicada, OpenML, que se encontra no jupyter 'Tratamento de Dados';

- Recolher informações sobre o número de exemplos para cada classe em todos os dataset, identificando que se encontram **desbalanceadas**;

- Realizar uma análise visual de distribuições de classes usando scatterplots para **identificar possíveis complicações não esperadas**;

- Desenvolver otimizações de K-nn, nomeadamente adição de pesos e de distâncias ponderadas ao algoritmo;
- Criar uma função de pesos inspirada em BWC (Ponderação Equilibrada de Classes) e implementar no nosso código;
- Elaborar gráficos de fácil compreensão que mostram as diferenças de performance para os algoritmos implementados.