Hiperparâmetros em Algoritmos

Parâmetros do Random Forest Classifier

n_estimators=100
Definição: Este parâmetro especifica o número de árvores na floresta.
Como Funciona:
Uma floresta aleatória consiste de múltiplas árvores de decisão. Cada árvore é treinada em um subconjunto diferente dos dados e suas previsões são combinadas (geralmente pela média ou votação majoritária) para produzir o resultado final.
Impacto:

- Um número maior de árvores geralmente melhora o desempenho, pois reduz o sobreajuste e varia da estimativa final. No entanto, um número maior também aumenta o tempo de treinamento e os requisitos de memória.
- 100 árvores é um valor comum e equilibrado que muitas vezes fornece um bom compromisso entre desempenho e eficiência computacional.

random_state=42
Definição: Este parâmetro controla a aleatoriedade do estimador.
Como Funciona:
Fixar o `random_state` como um inteiro (p. ex. 42) garante que o algoritmo produza os mesmos resultados em execuções distintas, desde que outros parâmetros também permaneçam constantes.
Impacto:

- É usado para tornar os resultados replicáveis. Na ciência de dados, é uma prática comum fixar sementes de aleatoriedade para fins de reprodutibilidade em experimentos e modelagem.
- **O valor 42** é uma escolha arbitrária (e uma referência humorística ao livro "O Guia do Mochileiro das Galáxias" de Douglas Adams, onde 42 é a "resposta para a vida, o universo e tudo mais"), mas qualquer número poderia ser usado.

Parâmetros do SVC

kernel='linear'
Definição:
O parâmetro kernel especifica a função de kernel a ser usada no método SVM. A função de kernel mapeia os dados de entrada de um espaço de características original para um espaço de dimensões mais altas, onde as classes podem ser separadas linearmente.
Tipos Comuns de Kernels:
Linear:

Usa um hiperplano linear para separar as classes. É eficaz em problemas onde os dados são linearmente separáveis.

Polynomial:

Usa um hiperplano polinomial, que pode modelar relações não lineares. Os parâmetros adicionais, como o grau do polinômio, podem ser ajustados.

RBF (Radial Basis Function):

Um kernel gaussiano que é muito eficaz na modelagem de relações não lineares. O parâmetro gamma é crucial e determina a largura da base da função radial.

Sigmoid:
Funciona como uma rede neural, mas raramente é usado na prática.

Impacto:
A escolha do kernel tem um grande impacto no desempenho do modelo. Se os dados não forem linearmente separáveis, usar um kernel linear pode levar a um desempenho insatisfatório.

random_state=42

Definição:
O parâmetro random_state é usado para controlar a aleatoriedade no SVM. Ele garante que resultados consistentes sejam obtidos em execuções diferentes do mesmo código.

Impacto:
O valor de random_state permite a reprodutibilidade do experimento. Usando um valor fixo, todos os aspectos aleatórios do modelo (como a seleção de subconjuntos de dados ou a inicialização de pesos) permanecerão consistentes entre execuções, tornando fácil comparar resultados.
Assim como no caso do **Random Forest Classifier**, usar um valor fixo (como 42, que é uma referência cultural) é uma prática comum na modelagem estatística e aprendizado de máquina.

Outros Hiperparâmetros Importantes

Outros hiperparâmetros que podem ser ajustados no SVC:

C:
O parâmetro de regularização que controla a penalização dos erros de classificação. Um valor maior de C significa uma penalização mais forte para erros de classificação, enquanto um valor menor prioriza uma maior margem de separação entre as classes, possibilitando mais erros.

gamma:
Usado com kernels não lineares como RBF. Controla a influência dos pontos de treinamento, onde um valor pequeno resulta em uma grande influência e um valor alto resulta em uma influência menor (decay rápido).

degree:
Específico para kernels polinomiais, este parâmetro ajusta o grau do polinômio.

A **SVM** é particularmente eficaz em espaços de alta dimensão e traz vantagens em relação a overfitting, especialmente em conjuntos de dados pequenos a médios.

SVM e SVC - Conceitos Importantes

- SVM (Support Vector Machine): É o termo geral que se refere ao algoritmo de aprendizado de máquina que usa vetores de suporte para classificação e regressão. O SVM pode ser considerado o modelo teórico ou a técnica subjacente.

- SVC (Support Vector Classifier): É a implementação específica do algoritmo SVM para problemas de classificação no `scikit-learn`. Dentro do `scikit-learn`, `SVC` é uma subclasse destinada a tarefas de classificação, enquanto **SVR (Support Vector Regression)** é a subclasse usada para tarefas de regressão.

Relacionamento

- SVM é o conceito geral.
- SVC é uma implementação concreta desse conceito para classificação, disponível em bibliotecas como o `scikit-learn`.
- Em resumo, SVC é uma forma de aplicar a técnica SVM para tarefas de classificação.

Essa distinção pode ser útil para entender como o SVM funciona na teoria e na prática. Se você tiver mais perguntas ou precisar de mais esclarecimentos, fique à vontade para perguntar!

Parâmetros do MLPClassifier

1. **hidden_layer_sizes=(100,)**
    - **Definição**: Esse parâmetro especifica o número de neurônios em cada camada oculta da rede neural. A entrada é uma tupla onde cada elemento representa a quantidade de neurônios em uma camada oculta.
    - **Impacto**:
        - `hidden_layer_sizes=(100,)` indica que há uma única camada oculta com 100 neurônios. Quando temos mais de uma camada, a tupla pode ser algo como `(100, 50)` para ter uma camada oculta de 100 neurônios e outra de 50 neurônios, por exemplo.
        - O número de neurônios e camadas ocultas influencia diretamente a capacidade da rede de aprender e modelar padrões complexos. Mais neurônios em mais camadas podem permitir um melhor ajuste a dados complexos, mas também podem aumentar o risco de overfitting, especialmente se o conjunto de dados for pequeno ou não for representativo.
2. **max_iter=300**
    - **Definição**: Este parâmetro define o número máximo de iterações (épocas) para o algoritmo de otimização durante o treinamento.
    - **Impacto**:
        - Se `max_iter` for pequeno, pode resultar em um treinamento incompleto da rede, levando a um modelo que não aprendeu adequadamente os padrões nos dados (underfitting).
        - Um valor maior permite que o modelo tenha mais chances de se ajustar aos dados, mas também pode levar a mais tempo de

computação e ao risco de overfitting, especialmente se o treinamento for continuamente executado sem uma validação para interromper o processo uma vez que não houver mais melhoria.

3. **random_state=42**

    - **Definição**: Esse parâmetro controla a aleatoriedade ao inicializar pesos e selecionar subconjuntos de dados ao longo do treinamento, garantindo resultados reprodutíveis.
    - **Impacto**:
        - Definir `random_state` como um número fixo (como `42`, uma referência humorística) garante que todas as execuções do código gerem os mesmos resultados, permitindo comparações consistentes entre experimentos.
        - É uma prática comum na modelagem estatística e aprendizado de máquina para facilitar a reprodutibilidade dos resultados.

Outros Hiperparâmetros Importantes

Além dos parâmetros mencionados, existem outros hiperparâmetros que podem ser ajustados no MLP:

- **learning_rate** e **learning_rate_init**: Controlam a taxa de aprendizado do otimizador.
- **activation**: Função de ativação usada nos neurônios. Comum incluem 'relu', 'tanh', e 'logistic'.
- **solver**: O algoritmo usado para otimização do backpropagation, como 'adam', 'sgd', ou 'lbfgs'.
- **alpha**: Termo de regularização (L2) para controlar o overfitting.

O **MLP** é uma ótima escolha para uma ampla variedade de problemas de aprendizagem de máquina, especialmente quando se tem dados complexos que exigem modelagem não linear.

Estrutura do Modelo KBANN

1. **model = Sequential()**

    - **Definição**: Inicializa um modelo de rede neural sequencial. Este tipo de modelo empilha camadas de forma linear, onde a saída de uma camada é a entrada da próxima. É a forma mais simples de construir uma rede neural em Keras.

2. **model.add(Input(shape=(input_dim,)))**

    - **Definição**: Adiciona uma camada de entrada ao modelo. `input_dim` representa a quantidade de características ou variáveis de entrada (features) do conjunto de dados.

- **Impacto**: Define a estrutura do tensor de entrada e especifica quais dimensões de dados a rede neural receberá na entrada.

3. **model.add(Dense(128, activation='relu'))**

   - **Definição**: Adiciona uma camada totalmente conectada (densa) com 128 neurônios. A função de ativação `relu` (Rectified Linear Unit) é aplicada a cada neurônio.
   - **Impacto**: Cada neurônio nessa camada aprenderá a representar padrões ou características de entrada, e a função ReLU ajuda a introduzir não linearidade, permitindo que a rede aprenda relações complexas nos dados.

4. **model.add(Dropout(0.5))**

   - **Definição**: Adiciona uma camada de dropout, que desativa aleatoriamente 50% dos neurônios da camada anterior durante o treinamento.
   - **Impacto**: O Dropout é uma técnica de regularização que ajuda a prevenir o overfitting ao forçar a rede a não depender excessivamente de neurônios individuais. Isso promove uma generalização melhor do modelo.

5. **model.add(Dense(64, activation='relu'))**

   - **Definição**: Adiciona outra camada densa com 64 neurônios e a função de ativação ReLU.
   - **Impacto**: Esta camada continua a processar os dados, aprendendo representações mais complexas à medida que os dados avançam pela rede.

6. **model.add(Dropout(0.5))**

   - **Definição**: Mais uma camada de dropout com 50% de neurônios desativados.
   - **Impacto**: Isso ajuda a manter a rede robusta e reduzir o risco de overfitting, especialmente em camadas ocultas onde pode haver muitas complexidades.

7. **model.add(Dense(32, activation='relu'))**

   - **Definição**: Uma terceira camada densa com 32 neurônios, também usando ReLU.
   - **Impacto**: Continuação do aprendizado de representações mais refinadas e complexas dos dados.

8. **model.add(Dense(len(np.unique(y_encoded)), activation='softmax'))**

    - **Definição**: Camada de saída que tem um número de neurônios igual ao número de classes únicas no conjunto de dados de saída. A função de ativação **`softmax`** é usada.
    - **Impacto**: A camada softmax transforma as saídas em probabilidades que somam a 1, sendo útil para tarefas de classificação multi-classe.

Compilação do Modelo

- **model.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=['accuracy'])**
    - **loss='sparse_categorical_crossentropy'**: Função de perda usada para problemas de classificação onde as classes são inteiros. É apropriada quando as saídas não são codificadas one-hot.
    - **optimizer='adam'**: Adam é um algoritmo de otimização eficiente que ajusta as taxas de aprendizado durante o treinamento.
    - **metrics=['accuracy']**: Define a métrica a ser monitorada durante o treinamento e a avaliação, neste caso, a acurácia do modelo.

activation='relu'

- **ReLU (Rectified Linear Unit)** é uma função de ativação comumente usada em redes neurais. Ela é definida como $f(x) = \max(0, x)$, o que significa que retorna 0 quando a entrada é negativa e retorna o valor da entrada quando é positiva.
- **Por que usar ReLU?**
    - **Simplicidade**: A função é computacionalmente eficiente.
    - **Sparse Activation**: Em redes profundas, ela tende a ativar apenas uma parte dos neurônios, tornando o modelo mais eficiente e menos propenso a overfitting.
    - **Mitigação do Problema do Desvanecimento do Gradiente**: ReLU ajuda a evitar esse problema, onde os gradientes se tornam muito pequenos durante a retropropagação, levando a um aprendizado lento ou estagnado.

`. epochs=50`

- **Epochs** referem-se ao número total de vezes que o modelo irá passar por todo o conjunto de dados de treinamento durante o treinamento. Cada época consiste em uma série de iterações nas quais o modelo ajusta os pesos para minimizar a função de perda.
- **Por que 50 epochs?** O número de épocas pode ser ajustado dependendo do problema e do tamanho do conjunto de dados.
    - Um número pequeno pode resultar em underfitting, pois o modelo não terá tempo suficiente para aprender.
    - Um número muito alto pode levar ao overfitting, onde o modelo começa a aprender padrões que são apenas específicos aos dados de treinamento e não generaliza bem para dados novos. Normalmente utiliza-se alguma estratégia para monitorar e parar o treinamento quando a performance em dados de validação começa a decair.

`Dense(128, activation='relu')`

- **Dense** define uma camada densa (totalmente conectada) da rede neural, onde cada neurônio da camada anterior está conectado a cada neurônio da camada atual.
- O número `128` refere-se ao número de neurônios na camada. Isso significa que a camada terá 128 unidades que irão processar a entrada.
- A função de ativação deste lote é `ReLU`, conforme explicado anteriormente. Isso significa que os 128 neurônios aplicarão a função ReLU às suas entradas.

`X_train.shape[1]`

- Essa expressão refere-se à dimensão dos dados de entrada. **`X_train`** é uma matriz (ou um array NumPy) que contém os dados de treinamento.
- **`X_train.shape`** retorna uma tupla que representa a forma do array. O primeiro elemento **`(shape[0])`** é o número de amostras (neste caso, quantas sequências de DNA existem em X_train), e o segundo elemento **`(shape[1])`** é o número de características ou características de entrada (ou seja, quantos recursos foram extraídos das sequências).
- Portanto, **`X_train.shape[1]`** retorna o número total de características usadas pela rede neural.

`model.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=['accuracy'])`

- **model.compile** é um método usado para configurar o processo de treinamento para a rede neural.
  - **loss='sparse_categorical_crossentropy'**:
    - Essa é a função de perda utilizada para medir o desempenho do modelo. É apropriada para problemas de classificação multi-classe onde as classes são inteiros (etiquetas inteiras). O 'sparse' refere-se ao fato de que os rótulos de classe podem ser representados como inteiros em vez de vetores one-hot.
  - **optimizer='adam'**:
    - Adam (Adaptive Moment Estimation) é um algoritmo de otimização que combina as vantagens do AdaGrad e do RMSProp. Ele ajusta a taxa de aprendizado para cada parâmetro individualmente e trabalha bem na prática para muitos problemas de aprendizado.
  - **metrics=['accuracy']**:
    - Aqui, você está especificando que deseja acompanhar a precisão do modelo durante o treinamento e a avaliação. A precisão avalia a proporção de previsões corretas em relação ao total de previsões.

Como a linha **`Dense(256, activation='relu', kernel_regularizer=l2(0.01))`**, e especialmente o hiperparâmetro **`kernel_regularizer=l2(0.01)`** têm impacto no algoritmo:

Componentes da Linha

1. **Dense(256)**:
    - Esta parte da linha define uma camada densa (camada totalmente conectada) com **256 neurônios**. Cada neurônio nesta camada terá conexões com todos os neurônios da camada anterior.
2. **activation='relu'**:
    - Aqui, você está especificando a função de ativação para os neurônios dessa camada. **ReLU** (Rectified Linear Unit) é uma função de ativação muito utilizada que transforma valores negativos em zero, enquanto mantém os valores positivos. Isso permite que a rede aprenda comportamentos não lineares de maneira mais eficiente em comparação com funções como sigmoid ou tanh.
3. **kernel_regularizer=l2(0.01)**:
    - O que este parâmetro faz é aplicar **regularização L2** na camada. A regularização L2 penaliza os grandes valores dos pesos ao adicionar um termo à função de perda. O termo de penalidade é a soma dos quadrados dos pesos multiplicada por um fator (neste caso, **`0.01`**).

Regularização L2

1. **Prevenção de Overfitting**:
    - A principal função da regularização L2 é ajudar a prevenir o overfitting. Quando a rede se ajusta muito aos dados de treinamento (ou seja, os pesos tornam-se muito grandes), isso pode levar a um desempenho ruim em novos dados não vistos. A regularização L2 penaliza pesos maiores, levando a uma rede neural que aprende características dos dados sem ficar excessivamente complexa.
2. **Estabilidade Durante o Treinamento**:
    - O fator de regularização (`0.01`) controla a força da penalidade. Um valor mais alto resultará em uma penalização mais forte, mas se for muito alto, pode levar a uma rede que não consegue aprender suficientemente os padrões nos dados.
3. **Interpretação do L2**:
    - L2 se refere à soma dos quadrados dos pesos. Assim, a penalização L2 ajuda a manter os pesos menores, o que, por sua vez, pode levar a uma rede que generaliza melhor para dados novos.

Escolha do Parâmetro de Regularização - l2

- **Ajuste do Valor**:
    - A escolha de **`0.01`** é uma escolha comum, mas pode ser ajustada conforme necessário. Experimentar com diferentes valores pode ser benéfico. Um menor valor pode ser menos penalizador, enquanto um maior pode resultar em uma rede menos capaz de overfit.

Conclusão

Incluir a **regularização L2** na camada, como em **`kernel_regularizer=l2(0.01)`**, é uma prática poderosa para manter a estabilidade do aprendizado, melhorar a generalização e controlar o overfitting.

Treinamento do Modelo

- **kbann_model.fit(X_train, y_train, epochs=50, batch_size=64, verbose=1)**
    - **X_train e y_train**: Dados de entrada e rótulos correspondentes para treinamento.
    - **epochs=50**: O número de passagens completas através do conjunto de dados de treinamento. Cada época permite que o modelo aprenda mais sobre os dados.
    - **batch_size=64**: O número de amostras a serem usadas antes de atualizar os pesos no modelo. Um tamanho de lote menor evita sobrecarga da memória, mas pode tornar o treinamento mais ruidoso.
    - **verbose=1**: Define o nível de detalhes durante o treinamento. Um valor de 1 mostra a barra de progresso.

Avaliação do Modelo

- **kbann_y_pred = np.argmax(kbann_model.predict(X_test), axis=-1)**

Validação Cruzada com KFold

A validação cruzada é uma técnica de avaliação de modelos que ajuda a estimar a performance de um modelo em dados não vistos, por meio da divisão do conjunto de dados em vários subconjuntos. O KFold é uma das abordagens mais comuns de validação cruzada.

Validação Cruzada com KFold

1. **Divisão dos dados**: O conjunto de dados é dividido em 'k' subconjuntos (ou "folds").
2. **Treinamento e validação**: Para cada um dos 'k' folds, o modelo é treinado usando os dados de 'k-1' folds e, em seguida, avaliado no fold restante. Esse processo é repetido 'k' vezes, com cada fold sendo usado como conjunto de validação uma vez.
3. **Média dos resultados**: As métricas de desempenho (como precisão, recall, etc.) são então médias para obter uma estimativa robusta do desempenho do modelo.

Parâmetro `folds=20`

Quando se menciona **`folds=20`**, isso significa que o conjunto de dados será dividido em 20 partes iguais. Cada parte (ou fold) será usada como conjunto de validação uma vez, enquanto o restante dos dados é usado para treinamento. Uma maior quantidade de folds pode resultar em uma estimativa mais precisa da performance do modelo, pois aumenta a

quantidade de validação e permite que o modelo seja testado em diferentes subconjuntos de dados.

No entanto, uma maior quantidade de folds também pode aumentar o tempo de computação, já que o modelo precisa ser treinado múltiplas vezes. Portanto, o valor de folds é um hiperparâmetro que pode ser otimizado dependendo do tamanho do conjunto de dados e da complexidade do modelo.

Aplicação no KBANN

No contexto do KBANN (Knowledge-Based Artificial Neural Networks), a validação cruzada com múltiplos folds (como 20) pode ser usada para avaliar a capacidade de generalização da rede neural treinada com conhecimento prévio, ajudando a evitar overfitting e a melhorar a robustez do modelo.

Matriz de Confusão

Uma matriz de confusão é uma tabela que é frequentemente usada para descrever o desempenho de um modelo de classificação em um conjunto de dados de teste, onde as verdadeiras classes são conhecidas. Ela mostra como as previsões se comparam às classes reais.

A matriz de confusão é estruturada da seguinte maneira:

	Predito		
	IE	**EI**	**NEITHER**
IE	**[V]**	**[FP]**	**[FN]**
EI	**[FN]**	**[V]**	**[FP]**
NEITHER	**[FN]**	**[FP]**	**[V]**

Onde:

V = Verdadeiro → Número de casos em que a classe foi prevista corretamente.

FP = Falso Positivo → Número de casos em que a classe foi prevista, mas era incorreta (erro do modelo)

FN = Falso Negativo → Número de casos em que a classe não foi prevista quando deveria ter sido (erro do modelo).