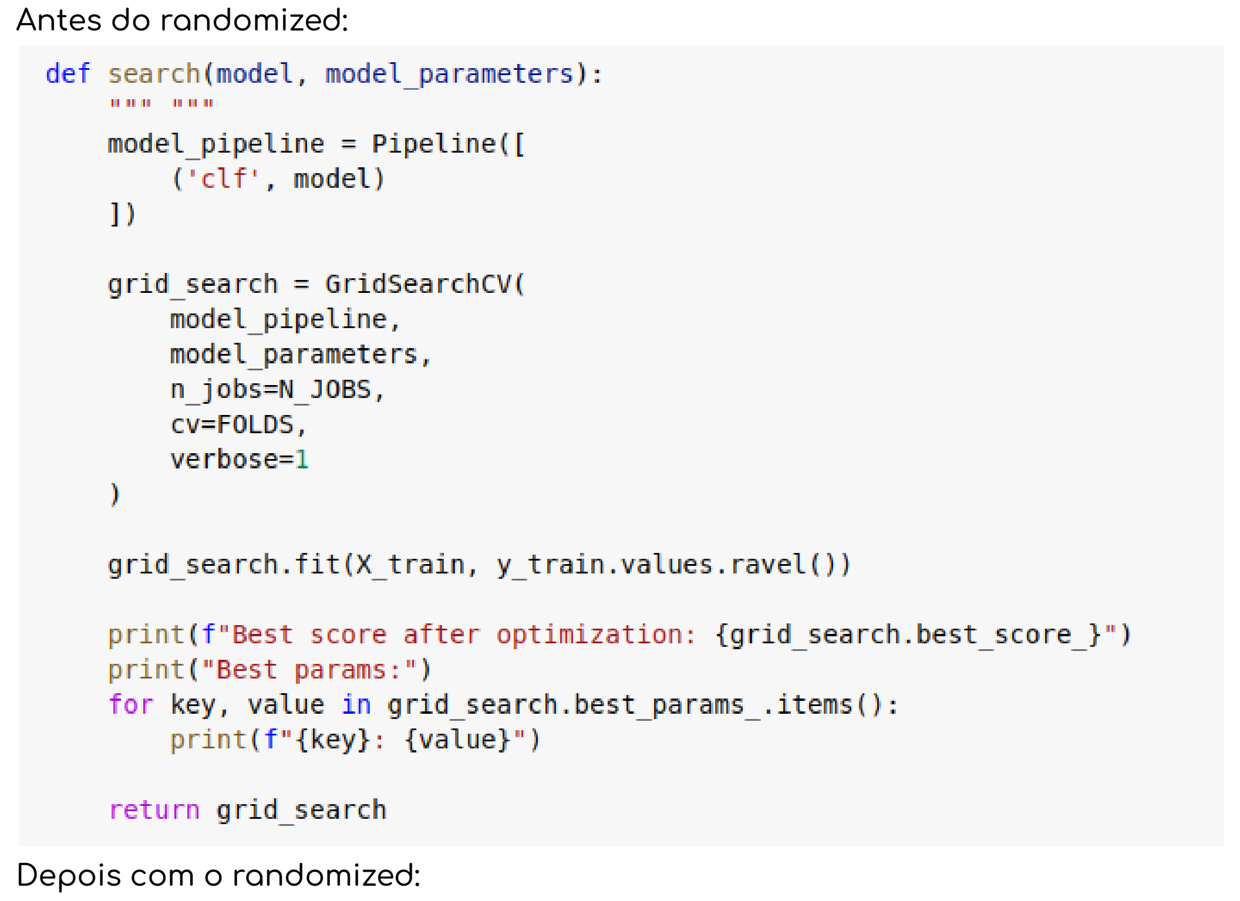
Relatório da Variação Paramétrica - I

Alunos:
André Luiz Pereira da Silva (alps2@cin.ufpe.br)
Victoria Célia César Ferreira (vccf@cin.ufpe.br)

Função de Otimização

Definimos duas funções de otimização: a grid search e a randomized. Ambas serão usadas para determinar a melhor combinação de parâmetros para cada modelo.
Uma função search é definida, que recebe uma instância de modelo a ser "variado", um dicionário de parâmetros e suas variações e uma String indicando se é grid ou randomized.
A gridsearch testa exaustivamente todas as configurações possíveis, o que garante o melhor resultado porém pode ser muito custosa em relação ao tempo.
A randomized não testa todas as configurações possíveis, não garante que seu resultado é o melhor, mas tem bons resultados, chegando perto do melhor. Pode ser vantajoso usar randomized porque ela é muito menos custosa em relação ao tempo. Se estabelece o número máximo de candidatos que serão testados. Empregamos o randomized porque alguns modelos como o random forest consumiram muito tempo.
Antes do randomized:



```
def search(model, model_parameters):
    """ """
    model_pipeline = Pipeline([
        ('clf', model)
    ])

    grid_search = GridSearchCV(
        model_pipeline,
        model_parameters,
        n_jobs=N_JOBS,
        cv=FOLDS,
        verbose=1
    )

    grid_search.fit(X_train, y_train.values.ravel())

    print(f"Best score after optimization: {grid_search.best_score_}")
    print("Best params:")
    for key, value in grid_search.best_params_.items():
        print(f"{key}: {value}")

    return grid_search
```

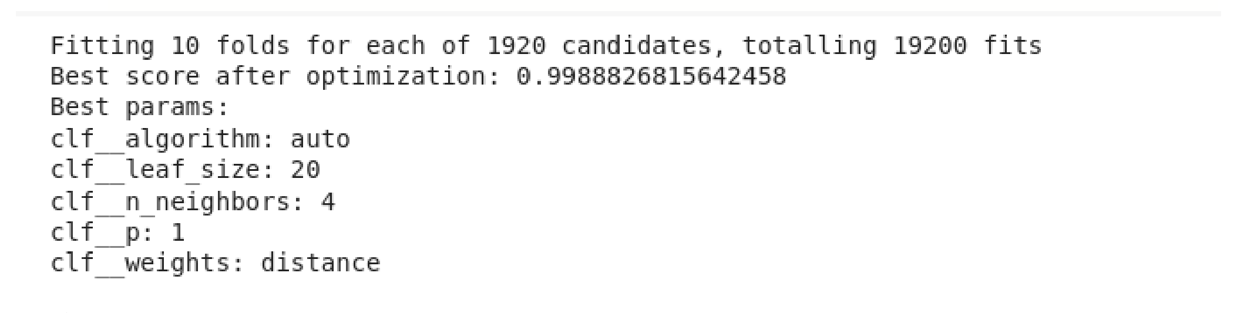
Depois com o randomized:

```
def search(model, model_parameters, search_method: str = "grid", randomized_max_candidates: int = 5000):
    """ """
    model_pipeline = Pipeline([
        ('clf', model)
    ])

    if search_method == "grid":
        search = GridSearchCV(
            model_pipeline,
            model_parameters,
            n_jobs=N_JOBS,
            cv=FOLDS,
            verbose=1
        )
    else:
        search = RandomizedSearchCV(
            model_pipeline,
            model_parameters,
            n_jobs=N_JOBS,
            cv=FOLDS,
            verbose=1,
            n_iter=randomized_max_candidates,
            random_state=RANDOM_SEED
        )

    search.fit(X_train, y_train.values.ravel())

    print(f"Best score after optimization: {search.best_score_}")
    print("Best params:")
    for key, value in search.best_params_.items():
        print(f"{key}: {value}")

    return search
```

Variando K-NN

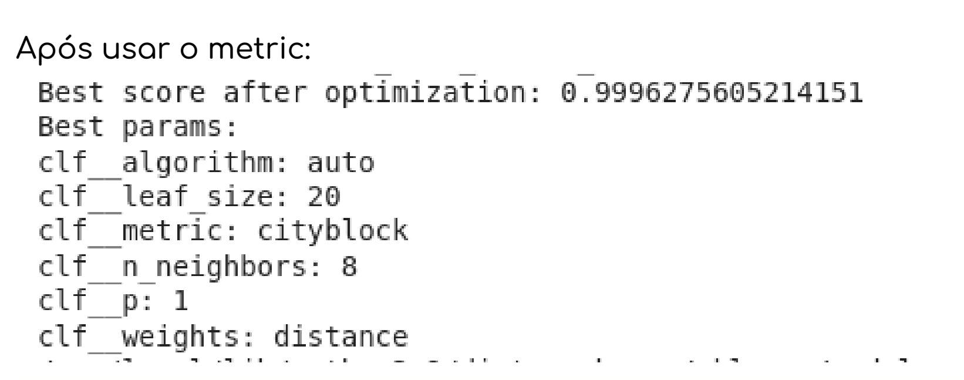
Parâmetros e ranges escolhidos para a variação:

1. n_neighbors[4,10]: número de vizinhos.
2. weights["uniform", "distance"]: função de peso usada para a previsão. No uniform todos os pontos tem o mesmo peso e no distance pontos mais próximos terão mais influência.
3. algorithm["auto", "ball_tree", "kd_tree", "brute"]: algoritmo para calcular os vizinhos mais próximos.
4. leaf_size[20,40]: parâmetro usado no BallTree ou KDTree. O default é 30, então decidimos um range intermediário.
5. metric ["cityblock","cosine","euclidean","haversine","nan_euclidean"]: métrica usada para calcular a distância.
6. P[1,3]: paramêtro usado na métrica de Minkowski. Se p = 1, vai usar a manhattan_distance e se p=2 a euclidean_distance. Qualquer outro valor vai usar a minkowski_distance.
   Antes de usar o metric:



```
Fitting 10 folds for each of 1920 candidates, totalling 19200 fits
Best score after optimization: 0.9988826815642458
Best params:
clf__algorithm: auto
clf__leaf_size: 20
clf__n_neighbors: 4
clf__p: 1
clf__weights: distance
```

Após usar o metric:

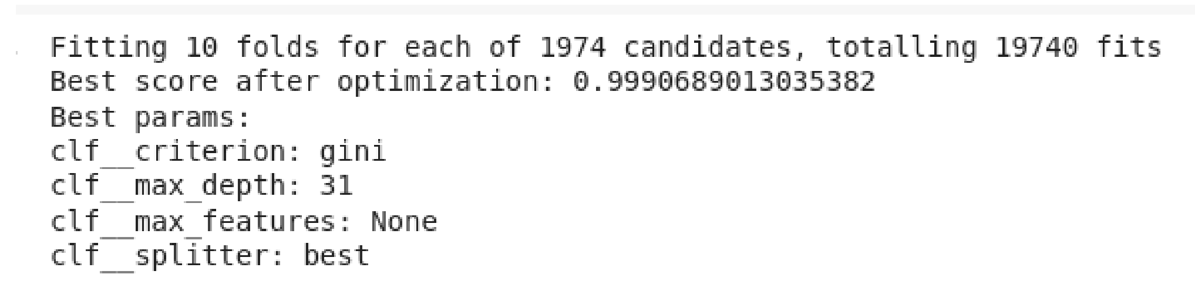


```
Best score after optimization: 0.9996275605214151
Best params:
clf__algorithm: auto
clf__leaf_size: 20
clf__metric: cityblock
clf__n_neighbors: 8
clf__p: 1
clf__weights: distance
```

Variando Decision Tree

Parâmetros e ranges escolhidos para a variação:

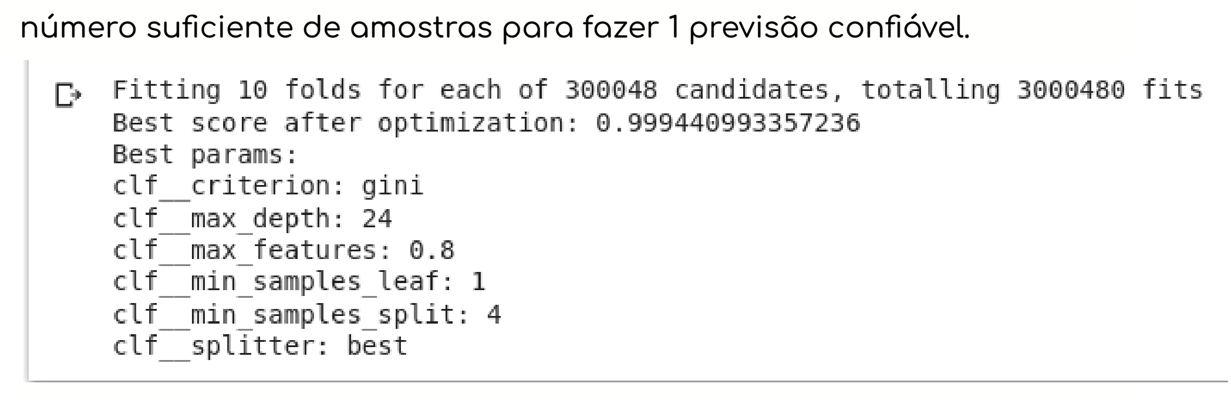
1. criterion ["gini", "entropy", "log_loss"]: Algoritmo para medição de qualidade de divisão de nós. Servem para determinar quais as melhores features para estarem mais perto do topo da árvore (ou seja, que apresentam maior ganho de informação).
2. splitter["best", "random"]: como os dados vão ser divididos a cada nó. Quando o splitter ocorre os dados são divididos em 2 ou mais subconjuntos e 1 nó para cada subconjunto é criado. A divisão dos dados procura melhor separar as diferentes classes.
3. max_features [0.2,0.4,0.6,0.8, None, "sqrt", "log2"]: máximo número de features/atributos considerados para o melhor splitter possível a cada nó da árvore. Usar muitas features resulta em overfitting e usar poucas em underfitting. Se não especificado o número máximo de features, o modelo irá treinar com todas as features.
4. max_depth[3, 50]: Máxima profundidade da árvore. Na construção de uma árvore de decisão o algoritmo recursivamente divide os dados em subconjuntos baseado nas features/atributos, até parar. Uma das formas de fazer o algoritmo parar é setando um valor máximo da profundidade da árvore, que é o max_depth. Se a árvore for muito profunda resulta em overfitting, se for pouco profunda resulta em underfitting.



```
Fitting 10 folds for each of 1974 candidates, totalling 19740 fits
Best score after optimization: 0.9990689013035382
Best params:
clf__criterion: gini
clf__max_depth: 31
clf__max_features: None
clf__splitter: best
```

Foram adicionados:

5. min_samples_split [2, 10]: o número mínimo de amostras necessárias para dividir um nó interno. Se o número de amostras é menor que o min_samples_split o nó não se dividirá mais e se tornará 1 folha. Pode ajudar a prevenir o overfitting pois força o algoritmo a considerar apenas features com um número suficiente de amostras para fazer 1 divisão.
6. min_samples_leaf [1,20] o número mínimo de amostras em uma folha. Se o número de amostras em uma folha é menor que o min_samples_leaf o nó se juntará com o seu nó vizinho, ou com nó pai se for o único filho. O processo de junção continua até todas as folhas terem no mínimo o número de amostras em min_samples_leaf. Um valor muito alto para min_samples_leaf pode levar a underfitting e um valor muito baixo pode levar a overfitting. O objetivo é assegurar que cada folha tem um número suficiente de amostras para fazer 1 previsão confiável.

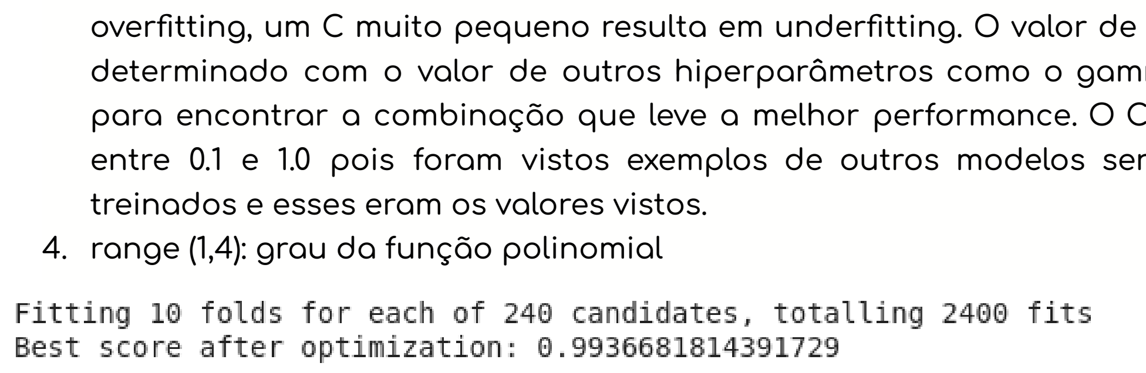


```
Fitting 10 folds for each of 300048 candidates, totalling 3000480 fits
Best score after optimization: 0.999440993357236
Best params:
clf__criterion: gini
clf__max_depth: 24
clf__max_features: 0.8
clf__min_samples_leaf: 1
clf__min_samples_split: 4
clf__splitter: best
```

Variando SVM

Parâmetros e ranges escolhidos para a variação:

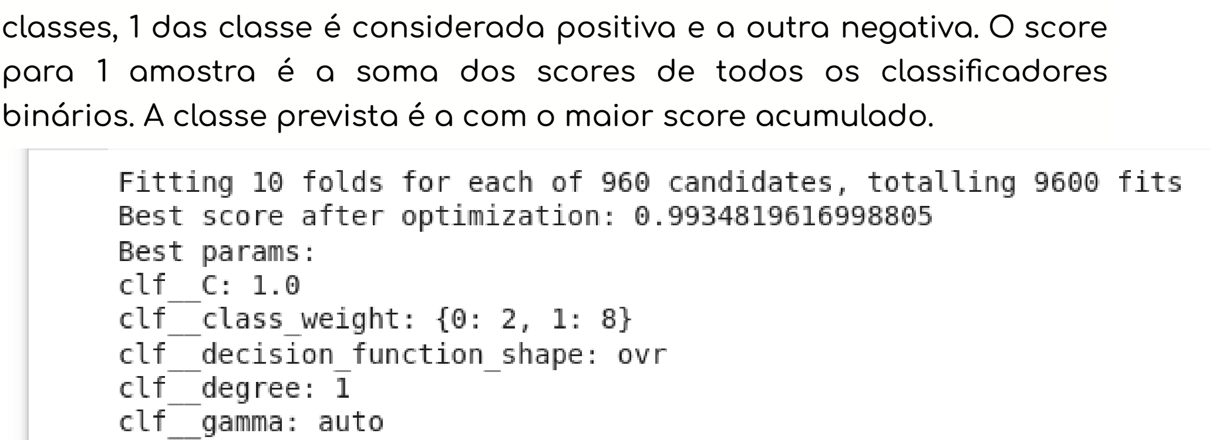
1. kernel["linear", "poly", "rbf", "sigmoid"]: O kernel é uma função matemática para transformar dos dados do input do seu espaço de features original em uma espaço de maior dimensão, para facilitar a separação de classes usando um classificador linear. A função de kernel recebe 2 vetores como input e retorna um valor escalar que mede a distância entre eles no espaço transformado. Essa medida de semelhança/distância é usada para encontrar o hiperplano que separa as classes diferentes no espaço transformado, com a maior distância possível entre os 2 pontos mais próximos de cada classe diferente.
   Linear: o espaço original;
   Polinomial: uma função polinomial sobre os dados de entradas;
   Gaussiana: uma função gaussiana é usada para medir a similaridade entre os vetores de entrada;
   Sigmóide: uma função sigmóide sobre os dados de entradas.
2. gamma["scale", "auto"]: o kernel coefficient (coeficiente do kernel). Controla a forma e largura da função de kernel. Define o range de influência de 1 único exemplo de treinamento na fronteira de decisão. Se o gamma é menor, a fronteira é mais simples e a influência de cada exemplo do treinamento é maior. Se gamma é maior, a fronteira de decisão é mais complexa e a influência de cada exemplo de treinamento é menor.
3. C[0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0]: o regularization parameter (parâmetro de regularização). O C controla o trade-off entre maximizar a margem e minimizar o erro de classificação. O C determina a penalidade por classificar de forma errada os exemplos de treinamento. Se C é pequeno, a penalidade é menor e a margem maior, se C é maior a penalidade é maior e a margem menor. Um C muito grande resulta em overfitting, um C muito pequeno resulta em underfitting. O valor de C é determinado com o valor de outros hiperparâmetros como o gamma, para encontrar a combinação que leve a melhor performance. O C foi entre 0.1 e 1.0 pois foram vistos exemplos de outros modelos sendo treinados e esses eram os valores vistos.
4. range (1,4): grau da função polinomial



```
Fitting 10 folds for each of 240 candidates, totalling 2400 fits
Best score after optimization: 0.9936681814391729
Best params:
clf__C: 1.0
clf__degree: 1
clf__gamma: auto
clf__kernel: rbf
```

Foram adicionados:

5. class_weight[0: 0.2, 1: 0.8] é usado quando há mais exemplos de 1 classe que de outra. Quando isso acontece o modelo pode se enviesar para a classe majoritária e ter uma performance pior na classe minoritária. Quando fizemos a análise exploratória de dados notamos que esse é o caso para o nosso problema. O class_weight atribui 1 peso para cada classe baseado na frequência que ela aparece nos dados. O SVM usa esse peso para ajustar a penalidade dada a cada classe no treinamento. Quanto maior o peso, maior a penalidade.
6. decision_function_shape["ovr","ovo"]: determina a forma da função de decisão. Em problemas de classificação binária, a função de decisão retorna um score (pontuação) para cada amostra indicando qual classe é a classe prevista. Se o score é >=0 é da classe positiva, se o score é <0 é da classe negativa. Em problemas com múltiplas classes, a função de decisão retorna um score para cada classe, e a classe prevista é a com o maior score. decision_function_shape controla o cálculo do score para problemas com múltiplas classes:
   "ovr" (one-vs-rest) existe um classificador SVM binário para cada classe, e as amostras de cada classe são consideradas da classe positiva e as outras amostras da classe negativa. O score para 1 amostra é o maior score dentre todos classificadores binários.
   "ovo" (one-vs-one) existe um classificador SVM binário para cada par de classes, 1 das classe é considerada positiva e a outra negativa. O score para 1 amostra é a soma dos scores de todos os classificadores binários. A classe prevista é a com o maior score acumulado.



```
Fitting 10 folds for each of 960 candidates, totalling 9600 fits
Best score after optimization: 0.9934819616998805
Best params:
clf__C: 1.0
clf__class_weight: {0: 2, 1: 8}
clf__decision_function_shape: ovr
clf__degree: 1
clf__gamma: auto
clf__kernel: rbf
```

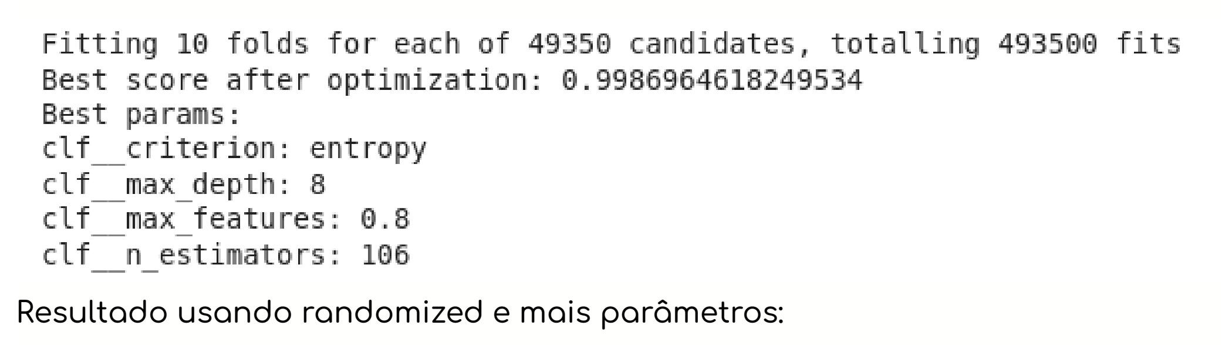
Variando Random Forest

Parâmetros e ranges escolhidos para a variação:

1. n_estimators [50:150]: Número de árvores de decisão presentes na floresta.
2. criterion ["gini", "entropy", "log_loss"]: Algoritmo para medição de qualidade de divisão de nós. Servem para determinar quais as melhores features para estarem mais perto do topo da árvore (ou seja, que apresentam maior ganho de informação).
3. max_features ["sqrt", "log2", None]: Número de atributos a serem considerados na hora de dividir.
4. max_depth [3:50]: Profundida máxima da árvore. O próprio SKLearn recomenda 3 como mínimo. O máximo depende muito das situações, mas via de regra, uma árvore mais profunda representa um classificador com risco de overfitting. Manter uma profundidade balanceada pode garantir um modelo mais genérico que consegue lidar com registros novos (diferentes dos que foram usado para o treino) sem muitos problemas.
5. min_samples_split [2..20]: Número mínimo de amostras necessárias para dividir um nó interno (que não são folhas).
6. min_samples_leaf [1..20]: Número mínimo de amostras necessárias para considerarmos um nó como folha. Aumentar esse número produz árvores mais generalistas, com profundidades menores.

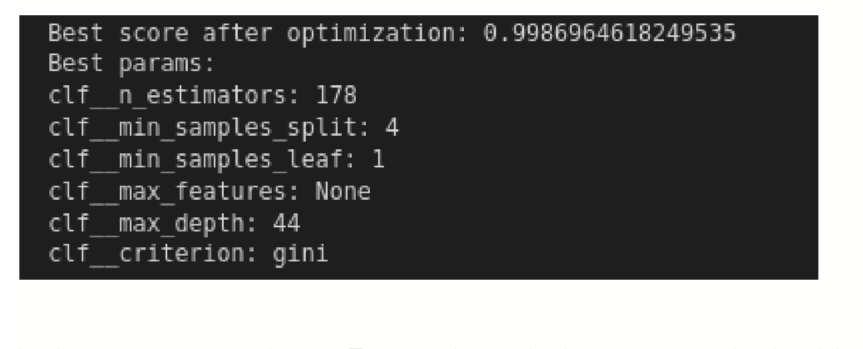
Da entrega I para a II, foram adicionados os parâmetros min_samples_split e min_samples_leaf. Originalmente essa busca foi feita usando gridsearch, o que resultou num tempo de execução de mais de 10 horas! Mas com o random search, limitamos o número de candidatos para 5000, economizando muito tempo e chegando num resultado semelhante.

Resultado original:



```
Fitting 10 folds for each of 49350 candidates, totalling 493500 fits
Best score after optimization: 0.9986964618249534
Best params:
clf__criterion: entropy
clf__max_depth: 8
clf__max_features: 0.8
clf__n_estimators: 106
```

Resultado usando randomized e mais parâmetros:



```
Best score after optimization: 0.9986964618249535
Best params:
clf__n_estimators: 178
clf__min_samples_split: 4
clf__min_samples_leaf: 1
clf__max_features: None
clf__max_depth: 44
clf__criterion: gini
```

Variando Rede Neural MLP

Parâmetros e ranges escolhidos para a variação:

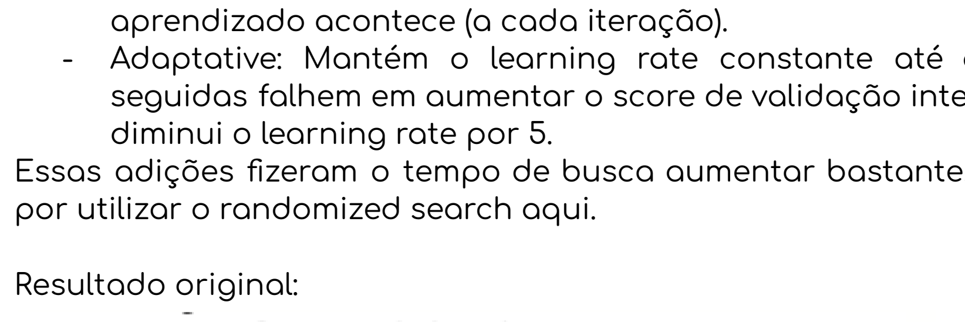
1. hidden_layer_sizes [(100,), (50, 50,), (33, 33, 34,), (25, 25, 25, 25,)]: Quantidade de camadas internas do perceptron (e o número de neurons em cada uma)
2. Activation ["identity", "logistic", "tanh", "relu"]: Função de ativação para os neurons.
3. Solver ["lbfgs", "sgd"]: De acordo com o sklearn, o solver "adam" funciona melhor com datasets maiores (na casa das dezenas de milhares). Como nosso dataset tem uma escala menor, seu uso não é recomendado, pois as alternativas convergem mais rapidamente e performam melhor.
4. Alpha: [0.0001, 0.0002, 0.0003, ..., 0.0009]: Termo de Regularização L2.
5. Learning rate ["constant", "invscaling", "adaptative"]: Taxa de aprendizado que define as atualizações de peso.
6. max_iter [50..1000]: Número de iterações a ocorrem internamente durante o treinamento. Maiores números significam mais tempo de processamento, e possivelmente mais precisão, já que o modelo terá mais tentativas para convergir numa solução.

Da entrega I para a II, foram adicionados os parâmetros Alpha, learning rate e max_iter.
O learning rate, em especial, foi adcionado por ser considerado um dos parâmetros mais importantes de uma MLP, pois define a taxa de aprendizado, ou seja, o quanto o modelo é impactado em cada atualização recorrente dos pesos internos. Um valor muito baixo pode resultar em treinamentos longos e pouco conclusivos (falhas de convergência), enquanto um valor muito alto pode resultar em convergência demasiadamente alta, em um conjunto de pesos que não é ótimo.
O learning rate pode assumir os seguintes valores:

- Constant: Valor constante durante todo o processo (0.001, por padrão).
- Invscaling: Gradualmente diminui o learning rate a medida que o aprendizado acontece (a cada iteração).
- Adaptative: Mantém o learning rate constante até que 2 iterações seguidas falhem em aumentar o score de validação interno. Nesse caso, diminui o learning rate por 5.

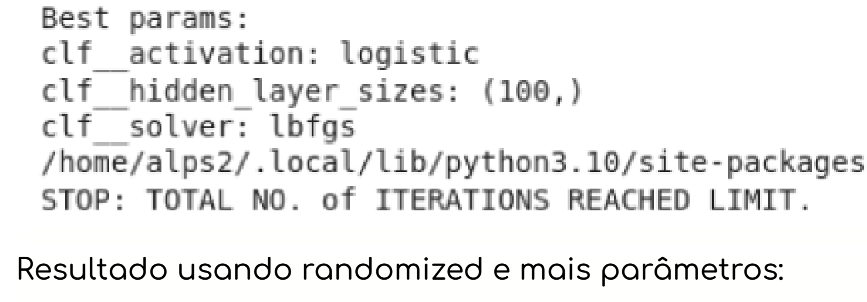
Essas adições fizeram o tempo de busca aumentar bastante, então optamos por utilizar o randomized search aqui.

Resultado original:



```
Best score after optimization: 0.9983240223463687
Best params:
clf__activation: logistic
clf__hidden_layer_sizes: (100,)
clf__solver: lbfgs
/home/alps2/.local/lib/python3.10/site-packages/s
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.
```

Resultado usando randomized e mais parâmetros:



```
Best score after optimization: 0.998510242085661
Best params:
clf__solver: lbfgs
clf__max_iter: 200
clf__learning_rate: invscaling
clf__hidden_layer_sizes: (100,)
clf__alpha: 0.0006
clf__activation: relu
```

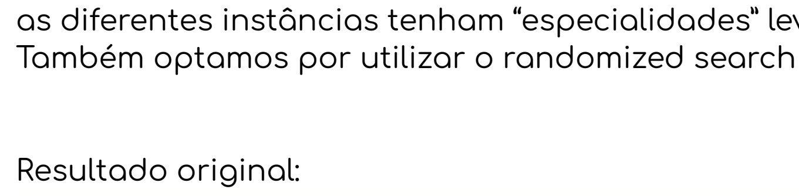
Variando Comitê de redes neurais Artificiais

Como o algoritmo Bagging é um ensemble e pode receber vários classificadores, decidimos instanciar o MLP com o resultado de sua otimização, para garantir maior qualidade do comitê.
Parâmetros e ranges escolhidos para a variação:

1. n_estimators [5:15]: Número de classificadores presentes no comitê.
2. max_samples [0.2:1]: Número de registros a serem extraídos de X na hora de treinar cada classificador do comitê.
3. max_features [0.2:1]: Número de atributos a serem considerados na hora de treinar cada classificador do comitê.
4. bootstrap [True, False].
5. bootstrap_features [True, False].

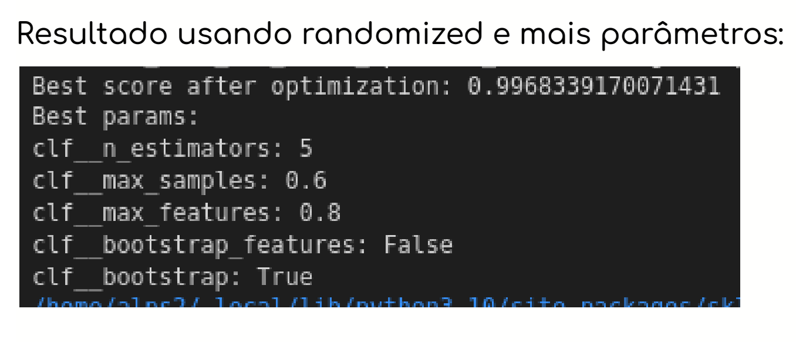
Da entrega I para a II, foram adicionados os parâmetros bootstrap e bootstrap_features, que controlam amostras dos conjuntos de treinos e de suas features, para treinamento das instâncias dos modelos que compõe o comitê.
O bootstrap_features, em especial, é interessante pois teoricamente faria com que os diferentes modelos foquem em features diferentes, possibilitando que as diferentes instâncias tenham “especialidades” levemente diferentes entre si.
Também optamos por utilizar o randomized search aqui, para poupar tempo.

Resultado original:



```
Best score after optimization: 0.9975784485394257
Best params:
clf__bootstrap: False
clf__bootstrap_features: False
clf__max_features: 0.8
clf__max_samples: 0.8
clf__n_estimators: 14
```

Resultado usando randomized e mais parâmetros:



```
Best score after optimization: 0.9968339170071431
Best params:
clf__n_estimators: 5
clf__max_samples: 0.6
clf__max_features: 0.8
clf__bootstrap_features: False
clf__bootstrap: True
```

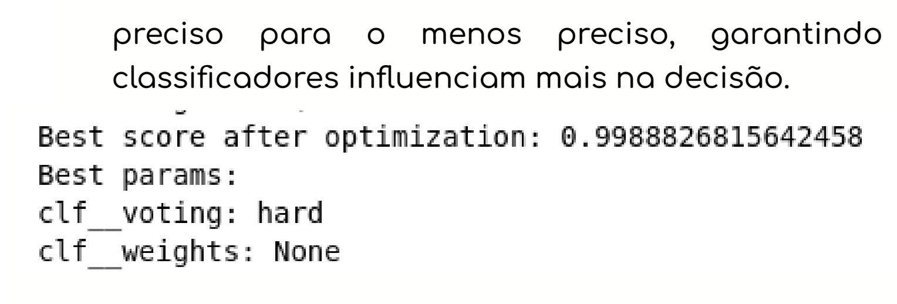
Variando Comitê Heterogêneo

Como o algoritmo VotingClassifier é um ensemble e pode receber vários classificadores diferentes, decidimos instanciá-los já com os melhores parâmetros provenientes de asuas otimizações. Os classificadores escolhidos para o VotingClassifier foram:

- MLP
- Random Forest
- SVM
- KNN

Parâmetros e ranges escolhidos para a variação:

1. voting ["hard", "soft"]: Regra para definir vencedor da votação (maioria simples com hard, probabilidade de classes com soft).
2. weights [None, [2.5, 2, 1.5, 1]]: pesos atribuídos para cada classificador. No caso de None, o peso é uniforme. Caso contrário, os pesos passados são atribuídos. Declaramos os classificadores em ordem do mais preciso para o menos preciso, garantindo que os melhores classificadores influenciam mais na decisão.



```
Best score after optimization: 0.9988826815642458
Best params:
clf__voting: hard
clf__weights: None
```