# IFT 3335 - TP2

Al Wahid Bio-Tchane (20137307)
Hanz Schepens (20189679)

December 2022

## 1 But du projet

Le but de ce projet est d'apprendre les méthodes d'apprentissages machines reliés au traitement de la langue naturel (NLP), principalement celles associés à la désambiguïsation de sens de mots. Dans ce projet nous avons abordé la vectorisation des données d'entrée, comment utiliser les algorithmes d'apprentissages automatiques et nous avons aussi exploré les processus pour marquer un texte avec des étiquettes utilisables dans le traitement du NLP.

## 2 Programmes utilisés

Dans ce projet, nous avons utilisés la librairie NLTK et SKlearn pour faire toutes les tâches requises. Nous avons pris la décision de faire un seul document python, qui est un Jupyter Notebook, car la simplicité de tout retrouver dans un seul notebook est beaucoup plus agréable que de travailler avec plusieurs scripts différents et le terminal. Ce notebook s'appelle : NLP.ipynb et il sera fourni dans le paquet que nous soumettrons.

## 3 Préparatifs

### 3.1 Familiarisation

Dans les instructions du projet, il nous était recommandé de nous familiariser avec des jeux de données plus simple pour apprendre comment les algorithmes de sklearn fonctionnent. Nous n'avons pas réalisé cette partie, car nous avons déjà eu des cours sur l'apprentissage automatique et nous sommes familiers avec cette librairie. Cependant, après avoir commencé le projet, nous avons remarqué que le traitement de la langue naturelle est très différent des techniques standard. Nous avons donc beaucoup lu sur internet pour nous familiariser avec les techniques qui sont standards dans ce domaine. Les textes annotés sont très étranges à regarder et à comprendre la première fois qu'on les inspecte, ce qui nous as poussé à essayer d'annoter le texte par nous même au lieu de prendre

le texte déjà annoté sur la base de données où nous devions télécharger le jeux de données.

## 3.2 Prétraitement

Pour commencer, nous avons décidé de prendre le jeux de données annoté, mais nous ne pouvions en faire ni queue, ni tête, donc nous avons pris le jeux de données qui était sans aucune annotation, car il était beaucoup plus facile de le lire et de comprendre ce qu'il se passait. Le texte annoté nous as montré de à quoi il devrait ressembler une fois toutes les étapes réalisés. Cela nous as permis d'étudier le chemin à suivre pour l'atteindre et nous as permis de comprendre comment on annote des textes pour le traitement de la langue naturelle.
Nous avons commencé par étudier la librairie de NLTK qui nous as servi a étiqueter le texte, mais avant de pouvoir l'utiliser, nous devions préparer le texte a être transformé. Après avoir mis le jeux de données en mémoire, nous avons dû nettoyer tout les $$ qui permettait d'identifier les phrases. Nous les avons tout simplement enlevé, sans plus. Une fois cette étape réalisé, nous avons séparé toutes les phrases, mots par mots, pour pouvoir effectuer les prochaines étapes. La première de ces étapes était d'enlever tout les "stop words" du vocabulaire anglais du texte que nous étions en train de traiter. Les stop words sont des mots qui n'apporte rien ou très peu d'information sur le contexte d'un mot ou de son sens. Après avoir enlevé les "stop words", nous étions rendu à l'étape du "stemming", tel que recommandé par le professeur. Nous avons décidé de faire le stemming après les stop words, car si les mots sont coupés à leurs racines seulement, ils ne feront pas parti des stop words et donc ne seront pas retirés du corpus. L'ordre logique s'est imposé, et nous avons utilisé le stemmer Porter, tel qu'indiqué par le professeur, de la librairie NLTK. Une fois que le corpus a été traité, nous avons rajouté les étiquettes pour le POS (part-of-speech), qui nous permet d'identifier le contexte des mots et leurs sens. Pour pouvoir tester plus tard, nous avons aussi décidé de faire 2 corpus, un avec le stemming, et une sans le stemming. Cela nous as permis de tester l'efficacité du stemming dans ce contexte. La dernière partie que nous avons fais pour le prétraitement est celle de la grammaire, qui n'était pas nécessaire dans notre situation. Nous avons fourni une petite grammaire et ensuite nous l'avons fourni à NLTK pour qu'elle nous produise un graphique de la structure de la phrase. Comme indiqué plus tôt, cela n'était pas nécessaire, mais assez important pour vérifier que nous avions fais du bon travail.

## 3.3 Extraction des propriétés (Feature Extraction)

Pour cette partie, nous nous sommes référé à la documentation de SKlearn pour créer la fonction qui nous as permis de faire un dictionnaire avec une fenêtre alentour de nos mots. Nous avons fais plusieurs fenêtres pour tester le code, et celles qui ne sont pas en utilisation sont commentés. Nous avons opté pour un dictionnaire au lieu d'un CountVectorizer ou du TfidfVectorizer, car nous avons trouvé plus de ressources en ligne pour le faire avec un dictionnaire, qu'avec les

deux autres, donc c'était plus simple et convenable, puisque SKlearn utilisais un dictionnaire dans leurs exemples, et cela semblait fonctionner très bien. Nous avons essayé les fenêtres de 1, 2 et de 4 pour nos tests. Respectivement, le mot précédent, le mot précédent et le prochain mot, et les deux mots précédents et les deux prochains mots. Nous avons aussi décidé d'utiliser les suffixes des mots précédents, pour tenter d'identifier des patrons. Cependant, les suffixes sont inutiles la plus part du temps lorsque nous avons déjà fais le stemming sur le texte. Une fois toutes les étapes précédentes réalisés, nous sommes passés aux algorithmes de classification.

# 4 Algorithmes de classification

Pour commencer, nous devions séparer nos données en 2 jeux de données. Un d'entraînement et un de test. Nous avons opté pour 80% pour le jeux de données d'entraînement et 20% pour le jeux de données de tests. Nous avons créé 2 méthodes pour séparer nos données, une qui implémenté par SKlearn et l'autre que nous avons implémenté nous même. Nous avons créé une méthode qui prend les données dans un ordre spécifique pour pouvoir tester plus facilement la précision des algorithmes. Cependant, nous sommes conscients que ce n'est pas recommandé pour réellement tester si les algorithmes sont précis, car normalement nous devons faire des jeux de données aléatoires pour s'assurer que les algorithmes n'ont pas simplement eu un coup de chance. La deuxième méthode qui est implémentée par SKlearn, nous permet de mettre une "seed" et de contrôler les jeux de données avec cette seed. Utiliser celle qui vous semble adéquat, par contre soyez avisé que celle de SKlearn coûte beaucoup plus chère en mémoire vive que celle que nous avons implémenté.
Une fois les jeux de données créés, il ne nous restait plus qu'a implémenter les algorithmes suivants : Naïve Bayes, Arbre de décision, Forêt aléatoire, Support vector machine (SVM) et pour terminer, le Multi layer perceptron (MLP). Ils seront détaillés dans leurs sections

## 4.1 Naïve Bayes

Nous avons fais tel que recommandé et nous avons utilisé Naïve Bayes pour faire nos tests avec les algorithmes. Nous avons été surpris à quel point cet algorithme était précis, considérant sa simplicité. Voici les résultats de nos tests avec les différents corpus :

- Corpus Stemmed
- 0.8000613936355264 (previous token)
- 0.8202316285743568 (prev1, next1)
- 0.8061904273854669 (prev1, prev2, next1, next2)
- Corpus Un-stemmed
- 0.8344778108025007 (previous token)
- 0.8446243722455673 (prev1, next1)
- 0.82781592702675 (prev1, prev2, next1, next2)

La différence ici est petite, mais remarquable. Nous pouvoir voir que lorsque le corpus est complet, Naïve Bayes donne un résultat marginalement meilleur qu'avec le corpus tronqué. Nous pouvons assumer que ces résultats sont similaires dans leurs augmentations et diminutions pour les autres algorithmes. Nous avons choisi Naïve Bayes pour représenter les autres classes, car il avait une bonne précision et que nous savions au préalable que Naïve Bayes est un système qui se porte bien au traitement de la langue naturelle, comme avec les systèmes de pourriels par exemple. La raison est que Naïve Bayes traite le document comme un sac de mot et cette hypothèse d'indépendance entre les mots est assez forte, cependant elle permet de simplifier le problème de désambiguïsation de sens car nous n'avons pas à prendre en compte le contexte des mots autant qu'auparavant et nous réduisons le problème à des probabilités. Naïve Bayes est un algorithme qui se porte très bien au projet que nous sommes en train de réaliser, par contre son plus grand défaut est le fait qu'il simplifie le problème énormément et cela ne lui permet pas vraiment d'aller chercher la plus grande précision qu'il pourrait aller chercher s'il prenait les mots dans leurs contexte comme que certain des autres algorithmes font. L'amélioration évidente pour Naïve Bayes est de ne pas tronquer les mots, car cela augmente sa précision un peu, et une autre amélioration possible serait de prendre d'avoir un plus gros corpus, car cela nous aiderait avec les probabilités et augmenterait sûrement la précision.

## 4.2 Arbre de décision

Pour les arbres de décisions, nous avons dû limiter la profondeur de l'arbre à 20, pour éviter le sur-apprentissage et parce que l'algorithme prenait beaucoup trop de temps à compléter. Nous l'avons fait avec la fonction de coût de l'entropie et de gini, voici les résultats des deux :

— Accuracy of Decision Tree with gini : 0.6826893512350107
— Accuracy of Decision Tree with entropy : 0.7336271394895972

Pour ces résultats et tous les prochains, ils ont été fait avec seulement le "previous token" sur le corpus tronqué, car le jeux de donnée était plus petit et accélérait les calculs. Nous avons essayé de faire un GridSearchCV avec SK-learn, mais l'algorithme n'a jamais retourné de résultat, ce qui est comprenable vu la grandeur du jeux de données. Nous avons décidé de faire quelques tests à la main pour voir ce qui donnait un bon résultat et nous nous sommes arrêtés sur un profondeur maximal de 20 et 5 pour le minimum qu'il doit avoir dans un groupe pour faire un autre split. En ce qui concerne la pertinence de cette algorithme pour le problème à résoudre, nous croyons qu'il n'est pas complètement inutile, mais qu'il y a des alternatives plus performantes, comme le Multi layer perceptron ou Naïve Bayes. Son gros défaut est que nous avons beaucoup de mots qui arrive seulement une fois, ce qui veut dire que les groupes de séparation ne seront pas très homogènes et donc les fonctions de coûts (gini et entropie) auront beaucoup de misère à séparer les groupes en des groupes distincts et homogènes. Pour améliorer l'algorithme, il faudrait vraiment enlever beaucoup plus de mots que seulement les mots d'arrêts. Il faudrait vraiment focus sur cer-

tains mots spécifiques, avec beaucoup moins de mots variés pour pouvoir créer des groupes distincts. Au final, nous ne croyons pas que les arbres de décisions sont idéals, mais clairement, ils sont mieux que de tirer au hasard à pile ou face pour déterminer le sens.

### 4.3 Forêt aléatoire

Les forêts aléatoires nous ont donné des résultats similaires aux arbres de décisions, principalement car ils se ressemblent fondamentalement :

— Accuracy of Random Forest Tree with gini : 0.625909603361689

— Accuracy of Random Forest Tree with entropy : 0.6430255201393871

Un peu inférieurs, mais similaires. Les forets aléatoires ont les mêmes problèmes que les arbres de décisions, c'est à dire qu'ils ne sont pas idéales pour le traitement de ce genre de problèmes, car les données sont de grandes dimensions et il est difficile de séparer les données en des groupes distincts et homogènes. Nous pensons que la raison pour laquelle nous avons eu des résultats inférieurs aux arbres de décision, est que nous avons limiter la profondeur des arbres à 20 et que ce n'est sûrement pas assez pour traiter notre corpus efficacement. Cependant, pour éviter le sur-apprentissage, il est important de limiter le taille de l'arbre maximale, mais puisque nous n'avons pas fais de GridSearch pour les forêts aléatoires, nous avions estimé que la même taille que les arbres de décisions serait assez, mais évidemment ce ne l'était pas. Les forêts aléatoires auraient dû donner un meilleur résultat, car ils sont meilleurs pour séparer les groupes, mais évidemment, ce n'était pas le cas. Peut-être que pour améliorer l'algorithme, l'utilisation d'une plus grande fenêtre aurait aidé et sûrement que le corpus non-tronqué aurait aidé aussi. L'avantage est que nous pourrions devenir très granulaire avec cette algorithme et atteindre un niveau de précision plus élevé que les arbres de décisions, mais nous aurions besoins de beaucoup plus de données pour y arriver.

### 4.4 Support vector machine (SVM)

Les SVM nous ont donnés les pires résultats de tous les algorithmes, et de loin, principalement car les données ne sont pas linéairement séparables :

— Accuracy of Support Vector Machine with linear kernel : 0.302 (iteration = 1)

— Accuracy of Support Vector Machine with linear kernel : 0.343 (iteration = 5)

— Accuracy of Support Vector Machine with polynomial kernel : 0.346 (iteration = 1)

— Accuracy of Support Vector Machine with polynomial kernel : 0.587 (iteration = 5)

— Accuracy of Support Vector Machine with sigmoid kernel : 0.458 (iteration = 1)

— Accuracy of Support Vector Machine with sigmoid kernel : 0.57 (iteration = 5)

Nous avons essayé avec plusieurs noyaux différents et nous avons aussi essayé d'augmenter le nombre d'itérations, mais le fléau de la dimension nous as limité à un nombre très petit d'itérations, sur un très petit nombre de données, 1000 entraînement et 1000 test. Ceci nous as obligé à réduire la dimension de nos données et nous avons donc choisi PCA (Principal Component Analysis) pour exécuter cette réduction à deux dimensions seulement. Une fois que nous avons réduits les données, nous avons pu faire l'algorithme sur l'ensemble des données dans un temps raisonnable et cela nous as donnée les résultats suivants :

- Accuracy of Support Vector Machine with linear kernel and reduction of dimensionality : 0.48396023367838475 (iteration = 1000)
- Accuracy of Support Vector Machine with polynomial kernel and reduction of dimensionality : 0.5309008916675207 (iteration = 1000)
- Accuracy of Support Vector Machine with sigmoid kernel and reduction of dimensionality : 0.37265552936353386 (iteration = 1000)

Nous avons pu remarquer une augmentation significative des performances de l'algorithme avec ce simple changement. Nous ne pouvons que supposer ce qu'il se passerait si nous pouvions exécuter les premiers résultats sur plus d'itérations, car cela nous prendrait trop de temps avec la librairie SKlearn et nous devrions plutôt utiliser les librairies qui permettent l'utilisation d'un GPU pour faire les calculs. Le noyau polynomial semble être celui avec le plus de précision pour nos deux batteries de tests et il donne même un meilleur résultat dans les premiers tests, ce qui nous indique à penser que la réduction de dimension nous as fais perdre beaucoup d'information sur nos données, car nous n'avons pas un meilleur résultat après la réduction et avec plus d'itérations. Le plus gros défaut de cette algorithme pour la résolution de ce problème est que les données ne sont pas linéairement séparable, et donc l'algorithme continuera sans fin, à moins de lui imposer une limite. Évidemment, dans un contexte où les données sont linéairement séparable, c'est alors la plus grande force de cette algorithme, par contre il n'est pas très approprié pour ce travail. Les autres algorithmes qui sont capable de définir des frontières de décisions plus précises et détaillés sont avantagés ici, tandis que le SVM linéaire génère un hyperplan pour sa frontière de décision et puisque les données sont de très grande dimension, il est dur de trouver sur quel plan il faudrait les projeter pour trouver un ensemble linéairement séparable. Il n'y a pas grand chose à faire de plus pour améliorer cette algorithme malheureusement, peut-être que des données avec une moins grande dimension serait préférable ou encore changer de librairie pour traiter le problème pourrait nous aider à gérer les données dans leurs dimensions originales.

## 4.5 Multi layer perceptron (MLP)

Pour le MLP, nous avons fait un GridSearchCV, sur seulement les 1000 premières données du jeux de données, avec la librairie SKlearn, pour tester plusieurs hyperparamètres différents. Nous avons fourni le document des résultats plus bas, car il est assez gros, mais pour résumer les meilleurs paramètres que nous avons trouvé : activation : tanh, alpha : 0.1, hidden layer sizes : (100,),

learning rate : constant, max iter : 500, solver : adam
Nous avons été surpris pour le paramètre des couches cachés, car nous étions sous l'impression qu'un réseau plus profond serait préférable pour résoudre ce problème, mais au contraire, une seule couche de 100 noeuds était meilleurs que les autres options que nous avons essayés. Ces hyperparamètres nous ont données une précision de 0.8328379624884699, ce qui est très bien considérant que nous lui avons fourni qu'une fenêtre de 1 alentour des mots. Le MLP est très approprié pour le traitement de la langue naturelle, après tout, c'est le standard de l'industrie de travailler avec des réseaux de neurones récurrents pour résoudre ce problème, et ils sont dans la même famille. Le niveau de détails que les réseaux de neurones peuvent accomplir est assez impressionnant et ils se portent bien aux données de hautes dimensions. Nous avons l'impression que pour améliorer l'efficacité de l'algorithme nous devrions lui donné une plus grand fenêtre alentour des mots et cela lui donnerait plus de contexte pour comprendre le sens des mots. Nous croyons que les fenêtres de taille de la phrase pourrait être appropriés pour les MLP, car ils donnerait le contexte complet, avec le moins d'ambiguïté possible. Une autre amélioration que l'on pourrait apporter serait de changer le MLP pour le RNN, car le RNN à la notion temporelle qui peut aider lorsque le sens d'un mot est dans une phrase précédente. Sa plus grande faiblesse est la disparition/explosion du gradient, car nous pourrions créer un réseau profond pour améliorer la qualité des prédictions, mais nous ne serions pas capable, car la descente de gradient se ferait mal.

# 5 Conclusion et Réflexion

En résumé, nous avons essayé quelques techniques pour améliorer le traitement du texte avec les algorithmes. Nous avons essayé d'utiliser les mots d'arrêts (stop words) et la troncature des mots (stemming) et le second semble avoir donnée des résultats pire lorsqu'il est appliqué sur le corpus. Nous avons aussi essayé plusieurs tailles de fenêtres différentes sur notre corpus et nous avons aussi essayé de réduire nos données avec l'algorithme des SVM, ce qui nous as fais gagner en temps énormément, mais perdre en précision. Nous avons comparé la précision de plusieurs algorithmes, et quelques uns se sont démarqués des autres, comme Naïve Bayes et perceptron a multi-couches, qui ont eut des performances très bonnes, considérant que nous n'avons pas essayé d'optimiser les algorithmes le plus que nous le pouvions. Certains des algorithmes ne se portaient pas très bien aux tâches relié au traitement de la langue naturelle, tel que les SVM et les arbres de décisions. Les SVM en particulier, ne sont pas très pratique lorsque les données ne sont pas linéairement séparable et ici elles ne le sont pas du tout. Ce que nous avons trouvé d'intéressant dans ce projet, fût la découverte de comment une texte doit être préparé pour qu'il soit compatible avec les algorithmes d'apprentissage automatique. C'était très enrichissant d'apprendre de nouvelles librairies sur un sujet assez complexe qui est utilisé un peu partout de nos jours. Cela nous as permis de mieux comprendre comment tout le processus d'annotation, d'extraction des propriétés et de prétraitement

fonctionne. Pour terminer, ce fut un projet qui nous as permis de comprendre pourquoi certains algorithmes se prêtent bien au NLP et pourquoi d'autres ne s'y prêtent pas bien, ainsi que les améliorations possible que nous aurions pu faire aux algorithmes. Le projet nous as aussi permis de valider nos connaissances sur l'apprentissage automatique, ce qui devrait nous être très utile dans le futur pour nos prochains projets.

# 6 Annexe

Voici tout les paramètres essayés pour le perceptron a multi-couches :

```
Best parameters found:
 {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'max_iter': 500, 'solver': 'adam'}
0.555 (+/-0.002) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'max_iter': 100, 'solver': 'sgd'}
0.686 (+/-0.008) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'max_iter': 100, 'solver': 'adam'}
0.605 (+/-0.011) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'max_iter': 200, 'solver': 'sgd'}
0.681 (+/-0.010) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'max_iter': 200, 'solver': 'adam'}
0.668 (+/-0.044) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'max_iter': 500, 'solver': 'sgd'}
0.699 (+/-0.005) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'max_iter': 500, 'solver': 'adam'}
0.555 (+/-0.002) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'max_iter': 100, 'solver': 'sgd'}
0.688 (+/-0.025) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'max_iter': 100, 'solver': 'adam'}
0.600 (+/-0.009) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'max_iter': 200, 'solver': 'sgd'}
0.688 (+/-0.012) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'max_iter': 200, 'solver': 'adam'}
0.672 (+/-0.003) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'max_iter': 500, 'solver': 'sgd'}
0.695 (+/-0.017) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'max_iter': 500, 'solver': 'adam'}
0.555 (+/-0.002) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'max_iter': 100, 'solver': 'sgd'}
0.692 (+/-0.008) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'max_iter': 100, 'solver': 'adam'}
0.595 (+/-0.029) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'max_iter': 200, 'solver': 'sgd'}
0.700 (+/-0.005) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'max_iter': 200, 'solver': 'adam'}
0.676 (+/-0.014) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'max_iter': 500, 'solver': 'sgd'}
0.687 (+/-0.020) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'max_iter': 500, 'solver': 'adam'}
0.555 (+/-0.002) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'max_iter': 100, 'solver': 'sgd'}
0.689 (+/-0.017) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'max_iter': 100, 'solver': 'adam'}
0.608 (+/-0.009) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'max_iter': 200, 'solver': 'sgd'}
0.694 (+/-0.014) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'max_iter': 200, 'solver': 'adam'}
0.679 (+/-0.004) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'max_iter': 500, 'solver': 'sgd'}
0.693 (+/-0.016) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'max_iter': 500, 'solver': 'adam'}
0.521 (+/-0.049) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'max_iter': 100, 'solver': 'sgd'}
0.701 (+/-0.015) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'max_iter': 100, 'solver': 'adam'}
0.563 (+/-0.002) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'max_iter': 200, 'solver': 'sgd'}
0.699 (+/-0.015) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'max_iter': 200, 'solver': 'adam'}
0.643 (+/-0.012) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'max_iter': 500, 'solver': 'sgd'}
0.696 (+/-0.008) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'max_iter': 500, 'solver': 'adam'}
0.504 (+/-0.001) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'max_iter': 100, 'solver': 'sgd'}
0.700 (+/-0.013) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'max_iter': 100, 'solver': 'adam'}
0.564 (+/-0.004) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'max_iter': 200, 'solver': 'sgd'}
0.695 (+/-0.007) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'max_iter': 200, 'solver': 'adam'}
0.643 (+/-0.005) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'max_iter': 500, 'solver': 'sgd'}
0.694 (+/-0.004) for {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'max_iter': 500, 'solver': 'adam'}
0.552 (+/-0.010) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'max_iter': 100, 'solver': 'sgd'}
0.693 (+/-0.013) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'max_iter': 100, 'solver': 'adam'}
0.596 (+/-0.033) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'max_iter': 200, 'solver': 'sgd'}
0.705 (+/-0.016) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'max_iter': 200, 'solver': 'adam'}
0.672 (+/-0.024) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'max_iter': 500, 'solver': 'sgd'}
0.698 (+/-0.011) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'max_iter': 500, 'solver': 'adam'}
0.557 (+/-0.007) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'max_iter': 100, 'solver': 'sgd'}
0.704 (+/-0.012) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'max_iter': 100, 'solver': 'adam'}
0.602 (+/-0.007) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'max_iter': 200, 'solver': 'sgd'}
0.692 (+/-0.019) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'max_iter': 200, 'solver': 'adam'}
0.680 (+/-0.019) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'max_iter': 500, 'solver': 'sgd'}
0.710 (+/-0.011) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'max_iter': 500, 'solver': 'adam'}
0.554 (+/-0.004) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'max_iter': 100, 'solver': 'sgd'}
0.691 (+/-0.017) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'max_iter': 100, 'solver': 'adam'}
0.607 (+/-0.017) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'max_iter': 200, 'solver': 'sgd'}
0.700 (+/-0.026) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'max_iter': 200, 'solver': 'adam'}
0.681 (+/-0.016) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'max_iter': 500, 'solver': 'sgd'}
0.706 (+/-0.009) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'max_iter': 500, 'solver': 'adam'}
0.555 (+/-0.002) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'max_iter': 100, 'solver': 'sgd'}
0.701 (+/-0.013) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'max_iter': 100, 'solver': 'adam'}
0.604 (+/-0.004) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'max_iter': 200, 'solver': 'sgd'}
0.701 (+/-0.013) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'max_iter': 200, 'solver': 'adam'}
0.674 (+/-0.019) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'max_iter': 500, 'solver': 'sgd'}
0.703 (+/-0.013) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'max_iter': 500, 'solver': 'adam'}
0.504 (+/-0.001) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'max_iter': 100, 'solver': 'sgd'}
0.712 (+/-0.001) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'max_iter': 100, 'solver': 'adam'}
0.563 (+/-0.002) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'max_iter': 200, 'solver': 'sgd'}
0.712 (+/-0.008) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'max_iter': 200, 'solver': 'adam'}
0.642 (+/-0.010) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'max_iter': 500, 'solver': 'sgd'}
0.715 (+/-0.021) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'max_iter': 500, 'solver': 'adam'}
0.516 (+/-0.018) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'max_iter': 100, 'solver': 'sgd'}
```

```
0.712 (+/-0.008) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'max_iter': 100, 'solver': 'adam'}
0.563 (+/-0.002) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'max_iter': 200, 'solver': 'sgd'}
0.710 (+/-0.015) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'max_iter': 200, 'solver': 'adam'}
0.644 (+/-0.018) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'max_iter': 500, 'solver': 'sgd'}
0.714 (+/-0.022) for {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'max_iter': 500, 'solver': 'adam'}
0.538 (+/-0.047) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'max_iter': 100, 'solver': 'sgd'}
0.693 (+/-0.005) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'max_iter': 100, 'solver': 'adam'}
0.593 (+/-0.031) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'max_iter': 200, 'solver': 'sgd'}
0.704 (+/-0.020) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'max_iter': 200, 'solver': 'adam'}
0.676 (+/-0.031) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'max_iter': 500, 'solver': 'sgd'}
0.701 (+/-0.004) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'max_iter': 500, 'solver': 'adam'}
0.555 (+/-0.002) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'max_iter': 100, 'solver': 'sgd'}
0.702 (+/-0.028) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'max_iter': 100, 'solver': 'adam'}
0.603 (+/-0.010) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'max_iter': 200, 'solver': 'sgd'}
0.696 (+/-0.019) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'max_iter': 200, 'solver': 'adam'}
0.664 (+/-0.031) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'max_iter': 500, 'solver': 'sgd'}
0.705 (+/-0.013) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'max_iter': 500, 'solver': 'adam'}
0.555 (+/-0.002) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'max_iter': 100, 'solver': 'sgd'}
0.695 (+/-0.011) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'max_iter': 100, 'solver': 'adam'}
0.596 (+/-0.051) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'max_iter': 200, 'solver': 'sgd'}
0.701 (+/-0.010) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'max_iter': 200, 'solver': 'adam'}
0.677 (+/-0.040) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'max_iter': 500, 'solver': 'sgd'}
0.708 (+/-0.012) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'max_iter': 500, 'solver': 'adam'}
0.560 (+/-0.009) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'max_iter': 100, 'solver': 'sgd'}
0.697 (+/-0.011) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'max_iter': 100, 'solver': 'adam'}
0.602 (+/-0.016) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'max_iter': 200, 'solver': 'sgd'}
0.701 (+/-0.029) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'max_iter': 200, 'solver': 'adam'}
0.679 (+/-0.027) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'max_iter': 500, 'solver': 'sgd'}
0.709 (+/-0.036) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'max_iter': 500, 'solver': 'adam'}
0.521 (+/-0.049) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'max_iter': 100, 'solver': 'sgd'}
0.719 (+/-0.012) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'max_iter': 100, 'solver': 'adam'}
0.563 (+/-0.002) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'max_iter': 200, 'solver': 'sgd'}
0.716 (+/-0.015) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'max_iter': 200, 'solver': 'adam'}
0.640 (+/-0.008) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'max_iter': 500, 'solver': 'sgd'}
0.719 (+/-0.017) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'max_iter': 500, 'solver': 'adam'}
0.516 (+/-0.035) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'max_iter': 100, 'solver': 'sgd'}
0.715 (+/-0.009) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'max_iter': 100, 'solver': 'adam'}
0.563 (+/-0.002) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'max_iter': 200, 'solver': 'sgd'}
0.719 (+/-0.015) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'max_iter': 200, 'solver': 'adam'}
0.638 (+/-0.016) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'max_iter': 500, 'solver': 'sgd'}
0.718 (+/-0.021) for {'activation': 'tanh', 'alpha': 0.1, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'max_iter': 500, 'solver': 'adam'}
0.431 (+/-0.004) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'max_iter': 100, 'solver': 'sgd'}
0.686 (+/-0.026) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'max_iter': 100, 'solver': 'adam'}
0.471 (+/-0.059) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'max_iter': 200, 'solver': 'sgd'}
0.687 (+/-0.017) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'max_iter': 200, 'solver': 'adam'}
0.617 (+/-0.022) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'max_iter': 500, 'solver': 'sgd'}
0.693 (+/-0.012) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'max_iter': 500, 'solver': 'adam'}
0.431 (+/-0.004) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'max_iter': 100, 'solver': 'sgd'}
0.692 (+/-0.019) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'max_iter': 100, 'solver': 'adam'}
0.455 (+/-0.070) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'max_iter': 200, 'solver': 'sgd'}
0.684 (+/-0.015) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'max_iter': 200, 'solver': 'adam'}
0.624 (+/-0.012) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'max_iter': 500, 'solver': 'sgd'}
0.674 (+/-0.024) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'max_iter': 500, 'solver': 'adam'}
0.431 (+/-0.004) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'max_iter': 100, 'solver': 'sgd'}
0.689 (+/-0.017) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'max_iter': 100, 'solver': 'adam'}
0.480 (+/-0.067) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'max_iter': 200, 'solver': 'sgd'}
0.684 (+/-0.020) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'max_iter': 200, 'solver': 'adam'}
0.617 (+/-0.011) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'max_iter': 500, 'solver': 'sgd'}
0.670 (+/-0.015) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'max_iter': 500, 'solver': 'adam'}
0.431 (+/-0.004) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'max_iter': 100, 'solver': 'sgd'}
0.682 (+/-0.021) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'max_iter': 100, 'solver': 'adam'}
0.431 (+/-0.004) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'max_iter': 200, 'solver': 'sgd'}
0.685 (+/-0.010) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'max_iter': 200, 'solver': 'adam'}
0.603 (+/-0.034) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'max_iter': 500, 'solver': 'sgd'}
0.681 (+/-0.047) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'max_iter': 500, 'solver': 'adam'}
0.451 (+/-0.059) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'max_iter': 100, 'solver': 'sgd'}
0.695 (+/-0.015) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'max_iter': 100, 'solver': 'adam'}
0.557 (+/-0.007) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'max_iter': 200, 'solver': 'sgd'}
0.688 (+/-0.021) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'max_iter': 200, 'solver': 'adam'}
0.621 (+/-0.021) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'max_iter': 500, 'solver': 'sgd'}
0.685 (+/-0.027) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'max_iter': 500, 'solver': 'adam'}
0.431 (+/-0.004) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'max_iter': 100, 'solver': 'sgd'}
0.694 (+/-0.017) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'max_iter': 100, 'solver': 'adam'}
0.555 (+/-0.002) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'max_iter': 200, 'solver': 'sgd'}
0.694 (+/-0.029) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'max_iter': 200, 'solver': 'adam'}
0.618 (+/-0.011) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'max_iter': 500, 'solver': 'sgd'}
0.684 (+/-0.032) for {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'max_iter': 500, 'solver': 'adam'}
0.431 (+/-0.004) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'max_iter': 100, 'solver': 'sgd'}
0.679 (+/-0.017) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'max_iter': 100, 'solver': 'adam'}
0.458 (+/-0.067) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'max_iter': 200, 'solver': 'sgd'}
0.696 (+/-0.023) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'max_iter': 200, 'solver': 'adam'}
0.613 (+/-0.034) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'max_iter': 500, 'solver': 'sgd'}
0.694 (+/-0.020) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'max_iter': 500, 'solver': 'adam'}
0.431 (+/-0.004) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'max_iter': 100, 'solver': 'sgd'}
0.692 (+/-0.015) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'max_iter': 100, 'solver': 'adam'}
0.504 (+/-0.001) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'max_iter': 200, 'solver': 'sgd'}
0.694 (+/-0.008) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'max_iter': 200, 'solver': 'adam'}
0.614 (+/-0.015) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'max_iter': 500, 'solver': 'sgd'}
0.694 (+/-0.009) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'max_iter': 500, 'solver': 'adam'}
0.431 (+/-0.004) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'max_iter': 100, 'solver': 'sgd'}
0.689 (+/-0.016) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'max_iter': 100, 'solver': 'adam'}
```

```
0.431 (+/-0.004) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'max_iter': 200, 'solver': 'sgd'}
0.696 (+/-0.025) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'max_iter': 200, 'solver': 'adam'}
0.609 (+/-0.033) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'max_iter': 500, 'solver': 'sgd'}
0.692 (+/-0.025) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'max_iter': 500, 'solver': 'adam'}
0.431 (+/-0.004) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'max_iter': 100, 'solver': 'sgd'}
0.679 (+/-0.030) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'max_iter': 100, 'solver': 'adam'}
0.521 (+/-0.047) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'max_iter': 200, 'solver': 'sgd'}
0.686 (+/-0.020) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'max_iter': 200, 'solver': 'adam'}
0.622 (+/-0.013) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'max_iter': 500, 'solver': 'sgd'}
0.688 (+/-0.014) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'max_iter': 500, 'solver': 'adam'}
0.431 (+/-0.004) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'max_iter': 100, 'solver': 'sgd'}
0.703 (+/-0.015) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'max_iter': 100, 'solver': 'adam'}
0.555 (+/-0.002) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'max_iter': 200, 'solver': 'sgd'}
0.709 (+/-0.014) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'max_iter': 200, 'solver': 'adam'}
0.616 (+/-0.012) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'max_iter': 500, 'solver': 'sgd'}
0.715 (+/-0.008) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'max_iter': 500, 'solver': 'adam'}
0.431 (+/-0.004) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'max_iter': 100, 'solver': 'sgd'}
0.702 (+/-0.023) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'max_iter': 100, 'solver': 'adam'}
0.555 (+/-0.002) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'max_iter': 200, 'solver': 'sgd'}
0.706 (+/-0.010) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'max_iter': 200, 'solver': 'adam'}
0.618 (+/-0.024) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'max_iter': 500, 'solver': 'sgd'}
0.715 (+/-0.013) for {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'max_iter': 500, 'solver': 'adam'}
0.431 (+/-0.004) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'max_iter': 100, 'solver': 'sgd'}
0.690 (+/-0.014) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'max_iter': 100, 'solver': 'adam'}
0.472 (+/-0.118) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'max_iter': 200, 'solver': 'sgd'}
0.700 (+/-0.016) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'max_iter': 200, 'solver': 'adam'}
0.616 (+/-0.027) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'max_iter': 500, 'solver': 'sgd'}
0.691 (+/-0.029) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'constant', 'max_iter': 500, 'solver': 'adam'}
0.431 (+/-0.004) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'max_iter': 100, 'solver': 'sgd'}
0.696 (+/-0.015) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'max_iter': 100, 'solver': 'adam'}
0.479 (+/-0.072) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'max_iter': 200, 'solver': 'sgd'}
0.695 (+/-0.011) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'max_iter': 200, 'solver': 'adam'}
0.610 (+/-0.030) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'max_iter': 500, 'solver': 'sgd'}
0.694 (+/-0.006) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'max_iter': 500, 'solver': 'adam'}
0.431 (+/-0.004) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'max_iter': 100, 'solver': 'sgd'}
0.697 (+/-0.018) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'max_iter': 100, 'solver': 'adam'}
0.479 (+/-0.072) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'max_iter': 200, 'solver': 'sgd'}
0.701 (+/-0.017) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'max_iter': 200, 'solver': 'adam'}
0.614 (+/-0.030) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'max_iter': 500, 'solver': 'sgd'}
0.698 (+/-0.013) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'max_iter': 500, 'solver': 'adam'}
0.431 (+/-0.004) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'max_iter': 100, 'solver': 'sgd'}
0.697 (+/-0.045) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'max_iter': 100, 'solver': 'adam'}
0.479 (+/-0.072) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'max_iter': 200, 'solver': 'sgd'}
0.705 (+/-0.024) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'max_iter': 200, 'solver': 'adam'}
0.598 (+/-0.038) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'max_iter': 500, 'solver': 'sgd'}
0.697 (+/-0.024) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'adaptive', 'max_iter': 500, 'solver': 'adam'}
0.431 (+/-0.004) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'max_iter': 100, 'solver': 'sgd'}
0.709 (+/-0.009) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'max_iter': 100, 'solver': 'adam'}
0.555 (+/-0.002) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'max_iter': 200, 'solver': 'sgd'}
0.716 (+/-0.012) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'max_iter': 200, 'solver': 'adam'}
0.617 (+/-0.013) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'max_iter': 500, 'solver': 'sgd'}
0.717 (+/-0.012) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'max_iter': 500, 'solver': 'adam'}
0.431 (+/-0.004) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'max_iter': 100, 'solver': 'sgd'}
0.710 (+/-0.019) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'max_iter': 100, 'solver': 'adam'}
0.555 (+/-0.002) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'max_iter': 200, 'solver': 'sgd'}
0.709 (+/-0.010) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'max_iter': 200, 'solver': 'adam'}
0.610 (+/-0.014) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'max_iter': 500, 'solver': 'sgd'}
0.717 (+/-0.020) for {'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (100,), 'learning_rate': 'adaptive', 'max_iter': 500, 'solver': 'adam'}
```