gathering brain stroke datasets training models on them and collecting the scoring metrics

This repository contains colection of brainstroke datasets. I have colected 13 datasets. Some of them are clasification tasks and some of them with regresion. Most of the datasets are in csv files but there are also jpg and png images. Apart from colecting the datasets, I train various regresion/clasification models and mesure the scoring metrics for all the models.

For training the models I used nested crosvalidation with 10 outter and 3 inner models to tune the parametars for each model. The scoring metrics are calculated on all inner and outter folds and stored in JSON file. I also mesure the fit and predict times and store them in the same JSON file. There are also 2 versions of the JSON file: regular and lite version. The regular version has the indexes of the instances in each fold and also the y and y\_predict values (also y\_predict\_probability for classifiction models) which the lite version is laacking but is smaller in size.

The JSON format is the folowing:

[

  {

    "model": "model1",

    "outer\_loop": [

      {

        "fold\_num": 1,

        "best\_params": {"C" : 1},

        "train": {

          "fit\_time": 0.3523557186126709,

          "pred\_time": 0.0796060562133789,

          "pred\_proba\_time": 0.08306407928466797,

          "scoring\_metric\_1": 0.999776885319054,

          "scoring\_metric\_2": 0.9977578475336323,

        },

        "test": {

            ## same as train

        },

        "param\_comb": [

          {

            "params": { "C" : 0.1},

            "inner\_fold": [

              {

                "fold\_num": 1,

                "train": {

                    ## ...

                },

                "test": {

                    ## ...

                }

              },

              {

                "fold\_num": 2,

                ## ...

              },

              {

                "fold\_num": 3,

                ## ...

              },

            ]

          }

        ]

      },

      ## ...

      {

          "fold\_num" : 10,

          ## ...

      }

     ]

   }

   {

    "model": "model2",

    ## ...

   }

]

Image 1: Visual tree of the JSON file structure.

Here are the models I ran on the regression tasks and a short description about them.

1. **AdaBoost Regression:** An ensemble learning method that combines multiple weak regression models to create a strong regression model.

2. **Automatic Relevance Determination Regression:** Automatic Relevance Determination Regression, a Bayesian linear regression model that automatically selects relevant features.

3. **Bagging Regression:** A technique that builds multiple regression models on different subsets of the training data and combines their predictions.

4. **Bayesian Ridge Regression:** A Bayesian approach to linear regression that introduces regularization to prevent overfitting.

5. **Decision Tree Regression:** A regression model based on decision trees, where data is split into branches to make predictions.

6. **Elastic Net Regression:** A linear regression model that combines L1 (Lasso) and L2 (Ridge) regularization to balance feature selection and model complexity.

7. **Gaussian Process Regression:** A non-parametric regression method that models the entire distribution of possible functions to make predictions.

8. **Gradient Boosting Regression:** An ensemble technique that builds a strong regression model by iteratively adding weak regression models.

9. **Hist Gradient Boosting Regression:** A faster version of gradient boosting that uses histogram-based techniques for regression.

10. **Huber Regression:** A robust regression method that is less sensitive to outliers than traditional least squares regression.

11. **KNeighbors Regression:** A regression model that predicts values based on the average or weighted average of the k-nearest neighbors in the training data.

12. **Lasso Regression:** Linear regression with L1 regularization, which encourages sparsity in the model by shrinking some coefficients to zero.

13. **Least Absolute Deviations Regression:** A regression method that minimizes the sum of the absolute differences between predicted and actual values.

14. **Least Angle Regression:** A feature selection method that gradually adds features to the model based on their correlation with the target variable.

15. **Linear Regression:** The simplest form of regression, which fits a linear equation to the data to make predictions.

16. **LightGBM Regression:** A gradient boosting framework that uses a histogram-based learning technique for regression tasks.

17. **Multi-layer Perceptron Regression:** A neural network model with multiple layers used for regression tasks.

18. **Ordinal Ridge Regression:** A variant of ridge regression designed for ordinal regression, where the target variable has ordered categories.

19. **Orthogonal Matching Pursuit Regression:** A sparse regression method that selects a subset of the most important features to make predictions.

20. **Passive Aggressive Regression:** A linear regression model that updates its parameters in an aggressive manner when prediction errors occur.

21. **RANSAC Regression:** A robust regression method that fits a model to the inliers in the data while ignoring outliers.

22. **Random Forest Regression:** An ensemble of decision tree regressors that averages their predictions to reduce overfitting.

23. **Ridge Regression:** Linear regression with L2 regularization, which prevents overfitting by penalizing large coefficients.

24. **SGD Regression:** Stochastic Gradient Descent regression, which optimizes a linear regression model using stochastic gradient descent.

25. **Support Vector Regression:** A regression technique that uses support vector machines to find the best-fitting hyperplane.

26. **Theil Sen Regression:** A robust linear regression method that estimates the slope and intercept of a line using median-based statistics.

27. **Tweedie Regression:** A regression model based on the Tweedie distribution, which is useful for modeling data with different types of error distributions.

28. **XGBoost Regression:** An optimized gradient boosting library that is widely used for regression tasks.

Some of the models are very simular even in some caces can get the same result but the purpose was to get veriaty and extensiveness.

**Scoring metrics-regression:**

The models were tested on the following metrics:

1. **Mean Absolute Error (MAE):** measures the **average absolute difference between the model's predictions and the actual values**. Lower values indicate better performance.

2. **Mean Squared Error (MSE):** calculates the **average of the squared differences between predictions and actual values**. Squaring the errors penalizes larger errors more than MAE, making it sensitive to outliers.

3. **Root Mean Squared Error (RMSE):** RMSE is the **square root of MSE**. It is commonly used because it shares the same unit of measurement as the target variable, making it easier to interpret.

4. **Median Absolute Error (MedAE):** MedAE is the median of the a**bsolute differences between predictions and actual values**. It is less sensitive to outliers compared to MAE and is useful when dealing with skewed data.

5**. Mean Percentage Error (MPE):** expresses the **average percentage difference between predictions and actual values**. It can help assess the model's bias in terms of percentage.

6. **Mean Absolute Percentage Error (MAPE):** MAPE is similar to MAE but expressed as a percentage of the actual values. It measures the **model's average percentage error**, making it interpretable and useful for comparing models.

7. **Symmetric Mean Absolute Percentage Error (SMAPE):** SMAPE is another percentage-based metric that accounts for both overestimation and underestimation errors. It provides a **symmetric view of the model's performance**. (considers errors in both directions, whether the model overestimates or underestimates the target variable, equally.)

8. **Relative Squared Error (RSE):** measures the **proportion of error variance relative to the total variance in the data**. It helps in understanding how much of the variability is explained by the model.

9. **Theil's U (U-statistic):** assesses the **relative performance of a model compared to a naive or benchmark model.** It is valuable for evaluating if a model adds value beyond a simple reference point.

10. **Mean Error (ME):** calculates the **average difference between predictions and actual values**. It provides information about the model's overall bias.

11**. Adjusted R-squared:** Adjusted R-squared is a modified version of the R-squared metric that considers the number of predictors in a regression model. It helps in **understanding the model's goodness of fit while penalizing for unnecessary complexity.**

12. **Explained Variance Score:** This metric quantifies the **proportion of variance in the target variable** that is explained by the model. It is particularly useful in situations where you want to assess how well the model captures variability.

13. **Jarque-Bera Test Statistic:** assesses **whether the residuals from a regression model follow a normal distribution**. It's essential for checking the assumption of normality in linear regression.

14. **Kolmogorov-Smirnov Statistic:** **evaluates the goodness-of-fit of a model's predictions to a given distribution**, often used for assessing the distributional assumptions of data.

15. **R-squared (Coefficient of Determination):** measures the proportion of the variance in the dependent variable that is explained by the independent variables in a regression model. It is a widely used metric for regression model evaluation.

I used the metric Mean Squared Error as the metric for the grid search to choose the best parameter combination.

For the regression models these are the tried parameter combinations:

AdaBoost Regression:

* 'n\_estimators': values [50, 100, 200, 400, 600], (Number of weak learners (base estimators).)
* 'learning\_rate': values [0.01, 0.1, 1.0], (Shrinkage parameter to control the contribution of each estimator. Small value means each tree in the ensemble has a minor impact on the final prediction lead to gradual convergence of the algorithm.)
* 'loss': values ['linear', 'square', 'exponential'], (Loss function to be used when updating weights.)
* 'estimator': values [DecisionTreeRegressor(max\_depth=1), DecisionTreeRegressor(max\_depth=3), DecisionTreeRegressor(max\_depth=7)], (Base estimator. Simpler models can reduce overfitting.)

ARDRegression:

* 'max\_iter': values [50, 100, 200, 400, 600], (Maximum number of iterations for optimization.)
* 'alpha\_1': values np.logspace(-6, -3, 4), ( Control how many important features the model selects. Larger values lead to stronger regularization.)
* 'alpha\_2': values np.logspace(-6, -3, 4), (Control how much the coefficients of all features should be shrunk towards zero.. Larger values lead to stronger regularization.)
* 'lambda\_1': values np.logspace(-6, -3, 4), (How much individual feature coefficients can vary. Larger values lead to stronger regularization.)
* 'lambda\_2': values np.logspace(-6, -3, 4), (Controlling the average size of all coefficients. Larger values lead to stronger regularization. )

Bagging Regression:

* 'n\_estimators': values [10, 50, 100, 200, 400], (Number of base estimators (bags). Larger values lead to stronger regularization.)
* 'estimator': values [None, LinearRegression(), Ridge(alpha=1.0), Lasso(), DecisionTreeRegressor()], (Base estimator to use. )
* 'max\_samples': values [0.7, 0.85, 1.0], (Fraction of samples used for fitting each bag. Larger values lead to stronger regularization.)

Bayesian Ridge Regression:

* 'max\_iter': values [50, 100, 200, 400, 600], (Maximum number of iterations for optimization.)
* 'alpha\_1': values np.logspace(-6, -3, 4), ( Control how many important features the model selects. Larger values lead to stronger regularization.)
* 'alpha\_2': values np.logspace(-6, -3, 4), (Control how much the coefficients of all features should be shrunk towards zero.. Larger values lead to stronger regularization.)
* 'lambda\_1': values np.logspace(-6, -3, 4), (How much individual feature coefficients can vary. Larger values lead to stronger regularization.)
* 'lambda\_2': values np.logspace(-6, -3, 4), (Controlling the average size of all coefficients. Larger values lead to stronger regularization. )

Decision Tree Regression:

* 'criterion': values ['squared\_error', 'friedman\_mse', 'absolute\_error'], (Function used to measure the quality of a split at each node.)
* 'max\_depth': values [1, 2, 3, 5, 7, 10, 15, 20, 25, 30, None], (Maximum depth of the tree. None means unlimited depth.)
* 'min\_samples\_split': values [2, 5, 10, 15, 20], (Minimum samples required to split an internal node.)
* 'max\_features': values ['log', 'sqrt', 0.1, 0.2, 0.25, 0.33, 0.5], #Maximum number of features to consider when splitting a node during tree construction. None: can use all available features.

Elastic Net Regression:

* 'alpha': values np.logspace(-3, 1, 5), (Combined L1 and L2 regularization strength.)
* 'l1\_ratio' : [0, 0.2, 0.5, 0.7, 1],
* (Mix between L1 and L2 regularization. 0: Ridge, 1: Lasso.)
* 'max\_iter': values [50, 100, 300, 500, 1000, 1500], (Maximum number of optimization iterations.)

Gaussian Process Regression:

* 'kernel': values [RBF(), Matern()], (Kernel function to model the covariance of the Gaussian process.)
* 'n\_restarts\_optimizer': values [1, 3, 5, 10], (Number of restarts for the optimizer to find the best kernel parameters.)
* 'alpha': values np.logspace(-3, 1, 5), (Regularization parameter for the Gaussian process. Larger value lead to stronger regularization)

Gradient Boosting Regression:

* 'n\_estimators': values [50, 100, 200, 400, 600], (Number of boosting stages.)
* 'learning\_rate': values [0.01, 0.1, 1.0], (Shrinkage parameter to control the contribution of each estimator. Small value means each tree in the ensemble has a minor impact on the final prediction lead to gradual convergence of the algorithm.)
* 'max\_depth': values [1, 3, 5, 7, 10, 15, 20], (Maximum depth of individual decision trees.)
* 'min\_samples\_split': values [2, 5, 10, 15, 20], (Minimum samples required to split an internal node.)
* 'subsample': values [0.7, 0.85, 1.0], (Fraction of samples used for fitting the trees.)
* 'max\_features': values ['log', 'sqrt', 0.1, 0.2, 0.25, 0.33, 0.5], (Maximum number of features to consider for a split.)

Hist Gradient Boosting Regression:

* 'n\_estimators': values [50, 100, 200, 400, 600], (Number of boosting stages (iterations). Larger values lead to risk of overfitting.)
* 'max\_depth': values [1, 2, 3, 5, 7, 10, 15, 20, 25, 30, None], (Maximum depth of the trees. None: no maximum depth. Smaller values lead to stronger regularization. )
* 'min\_samples\_leaf': values [2, 5, 10, 15, 20], (Minimum samples required to be at a leaf node. Larger values lead to stronger regularization.)
* 'learning\_rate': values [0.01, 0.1, 1.0], (Shrinkage parameter to control the contribution of each estimator. Smaller values lead to stronger regularization.)
* 'loss': values ['least\_squares', 'least\_absolute\_deviation'], (Loss function to be optimized.)
* (least\_squares: the sum of the squared differences between the predicted values and the actual target values,)
* (least\_absolute\_deviation: known as L1 loss (i.e. measures the sum of the absolute differences between the predicted values and the actual target values).)

Huber Regression:

* 'epsilon': values [1.0, 1.5, 2.0], (Loss parameter. Larger value lead to more resistant to outliers.)
* 'alpha': values np.logspace(-3, 1, 5), (L2 regularization term. Larger values lead to stronger regularization.)
* 'max\_iter': values [50, 100, 200, 400, 600], (Maximum number of iterations.)

KNeighbors Regression:

* 'n\_neighbors': values list(range(1, 12, 2)), (Number of neighbors to consider. Larger values make the model less sensitive to noise but smoother.)
* 'weights': values ['uniform', 'distance'], (Weight function used in prediction. 'uniform' treats all neighbors equally, 'distance' weights by inverse of distance.)
* 'algorithm': values ['auto', 'ball\_tree', 'kd\_tree', 'brute'], (Algorithm used to compute nearest neighbors.)
* 'p': values [1, 2], (Minkowski distance metric parameter. 1 is Manhattan distance, 2 is Euclidean distance.)

Lasso Regression:

* 'alpha': values np.logspace(-3, 1, 5), (Regularization strength (L1 regularization). Smaller values lead to weaker regularization.)
* 'max\_iter': values [None, 50, 100, 300, 500, 1000, 1500], (Maximum number of optimization iterations. If None the model takes the default for each solver.)

Least Absolute Deviations Regression:

* 'alpha': values np.logspace(-3, 1, 5), (Regularization strength (L1 regularization). Larger values lead to stronger regularization.)
* 'max\_iter': values [50, 100, 200, 400, 600], (Maximum number of iterations.)
* 'positive': values [True, False], (True: constrain coefficients to be positive, False: no constraints lead to greater flexibility to the model.)

Least Angle Regression:

* 'eps': values [1e-5, 1e-4, 1e-3], (L2 regularization parameter. Smaller values lead to stronger regularization.)

Linear Regression:

* No parameters tweaked.

LightGBM Regression:

* 'n\_estimators': values [50, 100, 200, 400, 600], (Number of boosting stages. Larger values may lead to better performance but longer training times.)
* 'learning\_rate': values [0.01, 0.1, 1.0], (Larger values shrinks the contribution of each tree, which can help prevent overfitting but may require more trees for similar predictive power.)
* 'max\_depth': values [1, 2, 3, 5, 7, 10, 15, 20, 25, 30], (Maximum depth of individual trees. Larger values can capture more complex relationships and can lead to overfitting if too large.)
* 'subsample': values [0.7, 0.85, 1.0], (Fraction of samples used for fitting trees. A larger value means using more data for training.)
* 'colsample\_bytree': values ['log', 'sqrt', 0.1, 0.2, 0.25, 0.33, 0.5], (Fraction of features used for fitting trees. A larger value increases diversity but may lead to overfitting if set too high.)

MLP Regressor:

* 'hidden\_layer\_sizes': values [(50,), (100,), (150,), (200,), (250,)], (Number of neurons in each hidden layer. Larger value lead to more complex)
* 'activation': values ['identity', 'logistic', 'tanh', 'relu'], (Activation function for hidden layers. 'identity': values returns its input as-is, 'relu': values Rectified Linear Unit)'solver': values ['lbfgs', 'sgd', 'adam'], (Optimization algorithm.)
* 'alpha': values np.logspace(-3, 1, 5), (L2 regularization term. Larger value lead to stronger regularization)
* 'learning\_rate': values ['constant', 'invscaling', 'adaptive'], (Learning rate schedule for weight updates.)
* 'learning\_rate\_init': values [0.001, 0.01, 0.1], (Initial learning rate.)
* 'max\_iter': values [50, 100, 200, 400, 600], (Maximum number of iterations.)

Ordinal Ridge Regression:

* 'alpha': values np.logspace(-3, 1, 5), (Regularization strength (L2 regularization). Larger values lead to stronger regularization.)
* 'max\_iter': values [50, 100, 200, 400, 600], (Maximum number of iterations.)
* 'solver': values ['auto', 'svd', 'cholesky', 'lsqr', 'sparse\_cg', 'sag', 'saga'], (Solver algorithm. 'lsqr': values Least Squares, 'sparse\_cg': values Conjugate Gradient, 'sag': values Stochastic Average Gradient Descent, 'saga': values sag with Adaptive Regularization.)

Orthogonal Matching Pursuit Regression:

* No parameters tweaked.

Passive Aggressive Regression:

* 'C': values [0.1, 0.5, 1, 2, 10, 100], (Regularization parameter. Smaller values lead to stronger regularization.)
* 'max\_iter': values [50, 100, 200, 400, 600], (Maximum number of iterations.)
* 'shuffle': values [True, False], (Whether to shuffle the training data at each iteration.)

RANSAC Regression:

* 'base\_estimator': values [None, LinearRegression(), Ridge(alpha=1.0), Lasso()], (Base estimator for RANSAC.)
* 'min\_samples': values [None, 0.1, 0.25, 0.5], (Minimum samples required to fit a model. None: no minimum requirement.)
* 'max\_trials': values [50, 100, 200, 400, 600], (Maximum number of RANSAC iterations.)
* 'loss': values ['absolute\_loss', 'squared\_loss'], (Loss function to use.)
* 'residual\_threshold': values [None, 0.5, 1.0], (Threshold for considering a data point as an inlier.)

Random Forest Regression:

* 'n\_estimators': values [50, 100, 200, 400, 600], (Number of trees in the forest. Larger values lead to stronger regularization. )
* 'max\_depth': values [1, 2, 3, 5, 7, 10, 15, 20, 25, 30], (Maximum depth of the trees. None means no maximum depth. Deeper trees can capture more complex patterns but may overfit. Smaller values lead to stronger regularization.)
* 'min\_samples\_split': values [2, 5, 10, 15, 20], (Minimum samples required to split an internal node. Larger values help prevent overfitting. Larger values lead to stronger regularization. )
* 'max\_features': values ['log', 'sqrt', 0.1, 0.2, 0.25, 0.33, 0.5], #Maximum number of features to consider for a split. Smaller values reduce model complexity. Smaller values lead to stronger regularization.

Ridge Regression:

* 'alpha': values np.logspace(-3, 1, 5), (Regularization strength (L2 regularization). Smaller values lead to weaker regularization.)
* 'solver' : ['auto', 'svd', 'cholesky', 'lsqr', 'sparse\_cg', 'sag', 'saga', 'lbfgs'], (Algorithm for optimization.)
* 'max\_iter': values [50, 100, 200, 400, 600], (Maximum number of optimization iterations. If None the model takes the default for each solver.)

SGD Regression:

* 'loss': values ['squared\_loss', 'huber', 'epsilon\_insensitive'], (Loss function to use for optimization.)
* 'penalty': values ['l1', 'l2', 'elasticnet'], (Penalty term for regularization.)
* 'alpha': values np.logspace(-3, 1, 5), (Regularization strength. Larger values lead to stronger regularization. )
* 'max\_iter': values [50, 100, 200, 400, 600], (Maximum number of iterations.)

Support Vector Regression:

* 'kernel': values ['linear', 'rbf', 'poly' , 'sigmoid'], (Kernel function for mapping data to a higher-dimensional space. Functions: Linear, Radial basis function (RBF), Polynomial.)
* 'C': values [0.1, 0.5, 1, 2, 10, 100], (Regularization parameter. Larger values allow for more flexible decision boundaries but may overfit.)
* 'epsilon': values [0.01, 0.1, 0.5], (Epsilon parameter in the SVR model. Larger value results in a wider tolerance zone.)
* 'degree': values [2, 3, 4], (Degree of the polynomial kernel (used with 'poly' kernel).)
* 'gamma': values ['scale', 'auto'] + [0.001, 0.01, 0.1, 1, 10], (Kernel coefficient for 'rbf', 'poly', and 'sigmoid' kernels. Smaller gamma values lead to smoother decision boundaries witch can overfit the data.)

(gamma = scale => gamma = 1/n\_features, gamma = auto => gamma = 1/n\_samples.)

Theil Sen Regression:

* 'max\_iter': values [50, 100, 200, 400, 600], (Maximum number of iterations.)
* Tweedie Regression:
* 'power': values [0, 1, 2], (Tweedie power parameter.)
* 'alpha': values np.logspace(-3, 1, 5), (Regularization strength (L2 regularization). Larger values lead to stronger regularization. )
* 'solver': values ['newton-cholesky', 'lbfgs'], (Solver algorithm.)
* 'max\_iter': values [50, 100, 200, 400, 600], (Maximum number of iterations.)

XGBoost Regression:

* 'n\_estimators': values [50, 100, 200, 400, 600], (Number of boosting stages. Larger values may lead to better performance but longer training times.)
* 'learning\_rate': values [0.01, 0.1, 1.0], (Shrinkage parameter to control learning rate. Smaller values reduce overfitting.)
* 'max\_depth': values [1, 3, 5, 7, 10, 15, 20], (Maximum depth of individual trees. Larger values can capture more complex relationships and can lead to overfitting if too large.)
* 'subsample': values [0.7, 0.85, 1.0], (Fraction of samples used for fitting trees. Smaller values reduce overfitting risk. )
* 'colsample\_bytree': values ['log', 'sqrt', 0.1, 0.2, 0.25, 0.33, 0.5], (Fraction of features used for fitting trees. A larger value increases diversity but may lead to overfitting if set too high.)