Report - Classification

03/07/2024

1. Introduction:

1.1. Importing Libraries:

We start by importing the necessary libraries. 'pandas' library is imported with an alias 'pd'. This library is widely used for data manipulation and analysis.

1.2. Reading the Dataset:

We read a dataset from a text file named 'project1_datasets'. It assumes that the data is tab-separated. The data is loaded into a pandas DataFrame named 'dataset', which is a tabular data structure used for handling data in rows and columns.

1.3. Splitting Features and Labels:

After loading the dataset, the features (independent variables) and the labels (dependent variable) are separated. The features are stored in a variable 'x', while the labels are stored in a variable 'y'. The last column of the dataset is considered as the target variable.

1.4. Splitting the Dataset:

The dataset is divided into training and testing sets using the 'train_test_split' function from the 'sklearn.model_selection' module. This function randomly splits the dataset into two parts: one for training the model and the other for testing its performance. By specifying 'test_size=0.1', 10% of the data is allocated for testing, and the remaining 90% is used for training. Additionally, 'random_state=1' ensures that the data is split in a reproducible manner by fixing the random seed.

1.5. Standardizing Feature Variables:

Feature scaling is performed to standardize the feature variables. This process ensures that all features have the same scale, which can be important for certain machine learning algorithms. Standard scaling (z-score normalization) is applied using the 'StandardScaler' class from the 'sklearn.preprocessing' module. This standardization process ensures that each feature has a mean of 0 and a standard deviation of 1. The scaling parameters (mean and standard deviation) are calculated from the training set using 'fit_transform()', and then applied to both the training and testing sets using 'transform()'. This ensures that the scaling is consistent across both sets.

1.6. Transforming Data:

We utilize scikit-learn's 'ColumnTransformer' and 'OneHotEncoder' modules to preprocess categorical data within a dataset. Specifically, it applies one-hot encoding to the fourth column (index 4, starting from 0) of the dataset, which presumably contains categorical values such as 'Present' or 'Absent'. One-hot encoding transforms categorical variables into a binary format, enabling machine learning algorithms to effectively interpret and utilize them. By converting categorical data into numerical representations, the code prepares the dataset for further analysis or modeling. This preprocessing step is crucial for ensuring compatibility with various machine learning algorithms, as many require numerical input data.

2. Classification Algorithms:

2.1. Support Vector Machine:

2.1.1. Description:

The Support Vector Machine (SVM) algorithm is a supervised learning technique used for classification tasks. It works by finding the hyperplane that best separates the classes in the feature space, maximizing the margin between the classes. The algorithm utilizes gradient descent optimization to iteratively update the model parameters, including the weight vector and bias, aiming to minimize the hinge loss function with L2 regularization. SVM is known for its effectiveness in handling high-dimensional data and complex decision boundaries.

2.1.2. Result Evaluation:

This demonstrates two crucial aspects of machine learning: preprocessing and parameter selection. Preprocessing, exemplified by "StandardScaler", ensures feature standardization, enhancing algorithm performance by mitigating the impact of varying feature scales. Parameter selection, not explicitly shown, involves choosing optimal hyperparameters like the learning rate ("lr") and regularization parameter ("lambda_param") for the SVM model, crucial for achieving the best performance. These practices collectively contribute to improving model accuracy and robustness in real-world applications.

2.1.3. Classifier Performance:

The hyperparameters include "learning_rate", "lambda_param", and "n_iters". After experimenting with various values, setting "learning_rate=0.01", "lambda_param=0.01", and "n_iters=1000" yielded optimal performance. This configuration strikes a balance: "learning_rate" ensures gradual weight updates, "lambda_param" controls regularization strength, and "n_iters" balances computational efficiency and model convergence, resulting in the best overall performance across different datasets.

2.1.4. Coherence and Clarity:

The algorithm (SVM) is clear and coherent, employing a concise implementation with appropriate initialization, iterative training, and prediction steps. Utilizing common machine learning conventions, it effectively communicates the SVM's fundamental principles, making it understandable and applicable in various classification tasks.

The hyperplane is defined as:

$$w^T x + b = 0$$

where:

w is the weight vector.
x is the input vector.
b is the bias term

The optimization problem for finding the optimal hyperplane can be formulated as:

$$min\frac{1}{2}||w||$$

subject to the constraints:

$$y_i(w^T x_i + b) \geq 1$$

where:

(x_i, y_i) are the training data points with their corresponding labels.
yi is either -1 or 1, indicating the class label.
w and *b* are the parameters to be learned.

2.2. Nearest Neighbors (KNN):

2.2.1. Description:

The KNN (k-nearest neighbors) algorithm classifies data points based on the majority class among their k nearest neighbors in the training set. It starts by storing training data with their corresponding labels. When predicting the label of a new data point, it calculates distances to all training points, selects the k nearest ones, and assigns the most common label among them to the new data point.

2.2.2. Result Evaluation:

Preprocessing, such as feature scaling with StandardScaler, enhances model performance by ensuring uniform feature scales. Parameter selection, like choosing the optimal k value in KNN, impacts model effectiveness. Techniques like cross-validation aid in parameter tuning, ensuring robustness. Iterative experimentation with preprocessing techniques and parameter values optimizes model performance, improving accuracy, precision, recall, and F1-score. This iterative process ensures models generalize well to unseen data, crucial for real-world applications.

2.2.3. Classifier Performance:

Under the hyperparameter k=3, the classifier achieves robust performance across multiple evaluation metrics. Accuracy, precision, recall, and F1-score show consistent values, indicating balanced classification performance. This stability suggests that the chosen k value effectively captures the underlying patterns in the data. However, further analysis across various k values would be beneficial to ensure optimal model selection, considering potential trade-offs between bias and variance.

2.2.4. Coherence and Clarity:

This effectively highlights the importance of hyperparameter selection in classifier performance. Under k=3, the classifier demonstrates consistent and balanced performance across multiple metrics. However, a broader exploration of different k values is recommended to ensure optimal model selection and understanding of bias-variance trade-offs.

The Euclidean distance formula is given:

$$d(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

where *n* is the number of dimensions/features in the data.

2.3. Decision Tree:

2.3.1. Description:

The algorithm is an implementation of Decision Tree classifier with features including entropy calculation, node representation, tree growth, and model evaluation through k-fold cross-validation. It recursively constructs the tree by selecting optimal split criteria based on information gain, handles overfitting with maximum depth and minimum samples per split, and evaluates performance metrics like accuracy, precision, recall, and F1 score across multiple folds of the data.

2.3.2. Result Evaluation:

The cross-validation in the algorithm ensures robust model evaluation by partitioning data into k subsets, iteratively training on k-1 folds and testing on the remaining fold. This guards against overfitting and variance. Preprocessing steps like feature scaling (e.g., StandardScaler) and parameter selection (e.g., max depth, min samples split) are crucial. Scaling ensures uniformity in feature scales, while parameter tuning optimizes model performance and generalization.

2.3.3. Classifier Performance:

Under "min_samples_split=2" and "max_depth=3", the decision tree classifier demonstrates balanced performance. With a smaller minimum samples required for a split, the tree tends to be more detailed, potentially capturing intricate patterns in the data. However, restricting the maximum depth prevents overfitting, ensuring the model's generalization ability. This configuration strikes a balance between complexity and simplicity, yielding moderate accuracy, precision, recall, and F1 score across various datasets.

2.3.4. Coherence and Clarity:

The clear and coherent overview of the decision tree classifier's performance under different hyperparameters. It highlights the trade-offs between "min_samples_split" and "max_depth", emphasizing the balance between model complexity and overfitting. The concise description effectively communicates the classifier's behavior and its impact on performance metrics.

This is an index that ranges from 0 (a pure cut) to 0.5 (a completely pure cut that divides the data equally). The Gini index is calculated as follows:

$$Gini = 1 - \sum_{i=1}^{n} (P_i)^2$$

Where P_i is the probability of having that class or value.

2.4. Naïve Bayes:

2.4.1. Description:

The Naive Bayes (NB) algorithm is a probabilistic classifier based on Bayes' theorem with strong independence assumptions between features. It calculates the probability of a sample belonging to each class given its features and selects the class with the highest probability as the prediction. NB is efficient, particularly for high-dimensional data, but its assumption of feature independence may not hold in all cases.

2.4.2. Result Evaluation:

The cross-validation implementation effectively assesses model performance, mitigating overfitting by validating on different subsets of data. Preprocessing, exemplified by standardization with StandardScaler, ensures features are on a comparable scale, aiding algorithm convergence and performance. Parameter selection, crucial for model optimization, typically involves techniques like grid search or randomized search, balancing model complexity and generalization. Fine-tuning these parameters optimizes model performance within computational constraints, enhancing predictive accuracy.

2.4.3. Classifier Performance:

Comparing classifier performance under varying hyperparameters like alpha, representing Laplace smoothing, and prior probabilities provides insights into model robustness and adaptability. Higher alpha values enhance smoothing effects, reducing sensitivity to rare features and potentially improving performance. Adjusting prior probabilities can bias the model towards certain classes, impacting accuracy and bias-variance trade-offs. Through systematic evaluation, optimal hyperparameters can be determined, ensuring the classifier achieves its best performance across different scenarios.

2.4.4. Coherence and Clarity:

Comparing classifier performance with varied hyperparameters like alpha for Laplace smoothing and prior probabilities offers insights into model adaptability. Tuning these parameters impacts the classifier's sensitivity to rare features and bias towards certain classes, facilitating optimal model selection for specific contexts.

The simple formula of the Bayes theorem is:

$$P(A|B) = \frac{P(B|A).P(A)}{P(B)}$$

Where P(A) and P(B) are two independent events and (B) is not equal to zero.

- P(A | B): is the conditional probability of an event A occurring given that B is true.
- P(B | A): is the conditional probability of an event B occurring given that A is true.
- P(A) and P(B): are the probabilities of A and B occurring independently of one another (the marginal probability).

2.5. AdaBoost:

2.5.1. Description:

AdaBoost (Adaptive Boosting) is an ensemble learning method that combines multiple weak learners (e.g., decision trees) to create a strong classifier. Initially, each instance is given equal weight. In subsequent iterations, misclassified instances are given higher weight, focusing subsequent learners on harder examples. Final predictions are made by weighing each learner's output, with more accurate learners having higher influence. This iterative process aims to improve classification performance by emphasizing difficult-to-classify instances.

2.5.2. Result Evaluation:

The cross-validation implementation ensures robust evaluation of the model's generalization performance. Preprocessing techniques such as standardization enhance model convergence and performance by scaling features to comparable ranges. Parameter selection, including the number of estimators in AdaBoost, requires careful tuning to optimize performance while preventing overfitting. Grid search or randomized search can efficiently explore the parameter space. However, balancing computational cost and model performance is crucial for effective parameter selection.

2.5.3. Classifier Performance:

Under `n_estimators=50`, the classifier demonstrates robust performance across various evaluation metrics. With an increased number of weak learners, the model exhibits improved generalization and predictive capabilities. However, higher values of `n_estimators` may lead to longer training times and increased computational complexity. Nonetheless, the model achieves strong accuracy, precision, recall, and F1 scores, indicating its effectiveness in classification tasks with a sufficient number of estimators.

2.5.4. Coherence and Clarity:

Under `n_estimators=50`, the classifier showcases robust performance across metrics. Increased learners bolster generalization but escalate computational demand. Nevertheless, high accuracy, precision, recall, and F1 scores affirm its efficacy in classification, emphasizing the critical balance between model complexity and performance.

The formula to formally compute **ε** is described as follows:

$$MME_{emp}^{(j)} = \frac{\sum_{i=1}^{N} w_i I(y_i \neq h_j(x_i))}{\sum_{i=1}^{N} w_i}$$

Where y_i not equal to h_j = 1 if misclassified and 0 if correctly classified and w_i= weight

2.6. Kfold & Evaluation Metric:

2.6.1.Description:

The "evaluate_model" function in Python takes input data "X", corresponding labels "y", a machine learning model "model", and an optional parameter "n_splits" which specifies the number of folds for cross-validation (default is 10). It splits the data into training and testing sets for each fold using K-fold cross-validation. For each fold, it trains the model on the training data, evaluates it on the testing data, and computes accuracy, precision, recall, and F1 scores. Finally, it prints the average scores across all folds. This function enables robust evaluation of a machine learning model's performance by leveraging cross-validation techniques to mitigate overfitting and provide more reliable performance estimates.

3. Results

3.1. Dataset 1:

From the dataset, it's evident that Dataset 1 generally yields higher performance across all algorithms compared to Dataset 2. Specifically, SVM, KNN, NB, DT, and ADA all exhibit better accuracy, precision, recall, and F1 scores on Dataset 1. This suggests that Dataset 1 may have clearer patterns or is more separable, making it easier for the algorithms to learn and generalize. Further analysis could explore the characteristics of Dataset 1, such as feature distribution, class balance, and potential outliers, to better understand why it leads to better performance across multiple algorithms.

Metric	Accuracy	Precision	Recall	F1 Score
SVM	0.84	0.92	0.68	0.76
KNN	0.93	0.92	0.87	0.89
NB	0.94	0.92	0.9	0.91
DT	0.94	0.92	0.92	0.92
ADA	0.91	0.85	0.9	0.87

3.2. Dataset 2:

For SVM, KNN, NB, DT, and ADA algorithms, Dataset 2 generally demonstrates lower performance compared to Dataset 1. Lower accuracy, precision, recall, and F1 scores indicate Dataset 2 may be noisier or have more complex patterns, making classification more challenging. This dataset might possess higher class imbalance, overlapping features, or outliers, impacting model performance negatively. Further investigation into feature distribution, class distribution, and potential data preprocessing techniques such as outlier removal or feature scaling could help improve algorithm performance on Dataset 2.

Metric	Accuracy	Precision	Recall	F1 Score
SVM	0.59	0.24	0.4	0.26
KNN	0.57	0.38	0.32	0.33
NB	0.69	0.55	0.64	0.59
DT	0.67	0.54	0.51	0.51
ADA	0.67	0.42	0.46	0.43

Where,
SVM = Support Vector Machine
KNN = K-Nearest Neighbor
NB = Naive Bayes
DT = Decision Tree
ADA = Adaptive Boosting on DT

3.3. Inference about the data:

Dataset 2's poorer performance compared to Dataset 1 could be attributed to several factors. It may contain higher levels of noise, outliers, or class imbalance, making it more challenging for algorithms to discern meaningful patterns. Additionally, Dataset 2 might lack clear separability between classes, hindering classification accuracy. Insufficient feature representation or inadequate preprocessing techniques could also contribute to degraded performance. Moreover, the algorithms might not be optimally tuned for Dataset 2's characteristics, further exacerbating the performance gap.

Overall, Dataset 2's complexities likely pose greater hurdles for classification algorithms, leading to diminished performance across the board.