This document is a detailed report of the implementation of 4 different Machine Learning Algorithms using 10 time 5-fold cross validation

Programming Project report

Group 2

Members:
Christabel tay (3739680),
Khadijat Abdulmalik (3740872)
Oluwasola Adesola Ayodeji (3726597)

Contents

1 Introduction

In this experimental study, our goal was to evaluate the performance of four machine learning algorithms: Adaboost on Tree Stumps, Random Forest, Artificial Neural Network (ANN) with backpropagation, and K-nearest neighbors (kNN) with two different distance functions (Manhattan and Euclidean). We implemented these algorithms using Python and evaluated them on several datasets obtained from the UCI Machine Learning Repository. The datasets include breast cancer, car evaluation, ecoli, letter recognition, and mushroom classification problems. Our evaluation was based on 10 times 5-fold cross-validation, where we report the average accuracy and standard deviation for each algorithm. Lastly, we compare and discuss the effectiveness of the various algorithms in different classification tasks, providing insights into their strengths and weaknesses.

2 Description of platform

Programming language:

Python programming language was used for this project.

Platform:

Google Collab platform was used for this project. Google Colab is a cloud-based platform that provides free access to Jupyter notebooks along with GPU and TPU resources. It enables us to write and execute Python code collaboratively, store and share notebooks, and leverage Google's computing power for data analysis, machine learning, and research projects.

Libraries

i. Pandas - Pandas is a Python library for data manipulation and analysis. It provides data structures like DataFrames and Series, along with tools for reading, writing, and manipulating structured data. Python Pandas was used for data preprocessing, cleaning, data exploration, and transformation tasks.
ii. NumPy - NumPy is a Python library for numerical computing. It supports multidimensional arrays, mathematical functions, random number generation, linear algebra operations, and many more.
iii. Matplotlib - was used to plot the graph for visual representation comparing the algorithms with the different dataset.

iv. Sklearn - Scikit-learn (sklearn) is a comprehensive machine learning library in Python. It offers tools for various machine learning tasks, including classification, regression, clustering, dimensionality reduction, and model selection. The following sub modules were used under sklearn:
   - Metrics - Scikit-learn's metrics module includes various evaluation metrics for assessing the performance of machine learning models, the function *accuracy_score* was used to get the accuracies of each model.
   - Preprocessing-Scikit-learn's preprocessing module offers functions for data preprocessing tasks like scaling, label encoder, OneHotEncoder. This was majorly used in ANN to handle categorical attributes.
   - *KFold*-cross-validation technique helped to partition the dataset into k equally sized folds.

3 Data description

The datasets used in this study are from the UCI Machine Learning Repository, including Breast Cancer Wisconsin (Diagnostic), Car Evaluation, Ecoli, Letter Recognition, and Mushroom datasets. Each dataset contains a target variable to be predicted by the model, varying in type from categorical to continuous, with a diverse range of attributes and classes.

The tables describe the various datasets:

Datasets	Number of examples	Number of attributes	Number of classes
Breast cancer	689	11	2(2,4)
Car	1727	7	3(unacc, good, vgood)
Ecoli	335	9	8(cp, im, imS, imL, imU,om, omL, pp)
Letter-recognition	19999	17	26(I, D, N, G, S, B, A, J, M, X, O, R, F, C, T, H, W, L,P, E, V, Y, Q, U, K, Z)
Mushroom	8123	23	2(e, p)

4 Algorithms

4.1 Adaboost on tree stumps

Adaboost on decision stumps, is a variant of the Adaboost ensemble learning algorithm. It employs shallow trees with only one split, instead of using full decision trees. Adaboost on tree stumps iteratively trains these weak learners (decision stumps) and focuses on the most challenging instances in the dataset. By assigning higher weights to misclassified samples in each iteration, it creates a strong classifier by combining multiple weak classifiers.

4.2 Random forest

Random Forest is an ensemble learning method for classification, regression, and other tasks that operate by constructing a multitude of decision trees at training time. For classification tasks, the output of the Random Forest is the class selected by most trees. It works on the principle of bagging, which helps in reducing the variance of the model without increasing the bias. This means that Random Forest can reduce overfitting by averaging several deep decision trees, trained on different parts of the same training set, with the goal of improving the predictive accuracy and control over-fitting. The Random Forest algorithm introduces extra randomness when growing trees; instead of searching for the most important feature while splitting a node, it searches for the best feature among a random subset of features. This results in a wide diversity that generally results in a better model.

4.3 ANN with back propagation

Artificial Neural Networks (ANN) with backpropagation is a type of machine learning algorithm inspired by the human brain's neural networks. It consists of interconnected nodes organized in layers, including input, hidden, and output layers. The algorithm uses backpropagation to adjust the weights of connections between neurons during training to minimize the difference between predicted and actual outputs. In backward propagation, the errors are propagated backward from the output layer to the hidden layers, updating the weights based on the calculated gradients using gradient descent optimization.

4.4 kNN

K-nearest neighbors (kNN) is a machine learning algorithm used for classification and regression tasks. It operates by storing all available instances of training data and classifying new instances based on their similarity to existing data points. The algorithm calculates the distance between the new instance and each training data point, typically using Euclidean or Manhattan distance metrics. It then selects the k-nearest neighbors and assigns the majority class for classification as the predicted outcome. KNN is simple, but its performance can be sensitive to the choice of k and the distance metric.

5 Pre-processing

We performed the following pre-processing steps on the various datasets.

5.1 Cleaning

We checked each dataset for missing or anomalous values and discovered that there were no missing values in the datasets. Then, we drop irrelevant columns (ID and sequence) from the Breast Cancer and Ecoli dataset respectively, the IDs were not contributing to the “target variable”. We found invalid data in car dataset specifically in column doors and persons, we corrected the invalid data by selecting the most common value in the class the example data belongs to. We then replaced the invalid data with the selected value.

5.2 Renaming features

From the dataset, we had to rename the columns headings to help distinguish between the attributes, to enable the models to understand each dataset.

5.3 Encoding Categorical Variables

Convert categorical variables into a form that could be provided to the ML algorithms to do a better job in prediction. For example, we applied encoding on some columns of the mushroom, Ecoli, car and letter-recognition dataset. In ANN we applied onehot encoding for the multiclass dataset.

5.4 Hyperparameter Selection

Hyperparameters are parameters that define the structure or behavior of a machine learning algorithm. We selected hyperparameters for the implementation of the various algorithms.

5.4.1 Adaboost on tree stumps

In Adaboost with tree stumps, we used five as the number of weak classifiers.

5.4.2 Random forest

Number of Trees (n_trees) = 20 (this figure was equal for all the dataset except Letter Recognition dataset, where we used 50 to help improve the accuracy)
Maximum Depth (max_dept) = 5
Minimum Samples Split (min_samples_split) = 2

5.4.3 ANN

Number of hidden layers: 2
Learning rate: 0.01
Number Iterations: 100 (Breast Cancer, Mushroom, Car) 1000(Letter of recognition, Ecoli)
Loss function: Cross Entropy Loss
No of output Neuron for hidden layers: 40, 40
Hidden Layer activation Function: Relu

5.4.4 kNN

Number of Nearest Neighbor: 3

6 Implementation

6.1 Adaboost on tree stumps

Step1: Shuffle Dataset

To begin with, the dataset was randomly shuffled to remove any bias in the data. This allowed each fold obtained in the cross-validation to be diverse.

Step 2: Split into Training and Test Sets

Each shuffled dataset was divided into 5 equal-sized folds and iterated over 5 times. Each iteration had one-fold used as test and the maintaining folds used as training set. This process ensured that each sample in the dataset is used for both training and testing.

Step 3: Training with Adaboost model

For each fold in the cross-validation process:

i. We selected the current fold as the test set and combined the remaining folds to create the training set.
ii. We initialized the AdaBoost model with a specified number of weak learners (decision stumps).
iii. We initialized the weights of all samples in the training set to be equal.
iv. For each weak learner:
    - We trained it on the current training set.
    - Adjusted the sample weights to emphasize the misclassified samples from the previous weak learner.
    - Computed the alpha weight, which represents the contribution of the current weak learner to the final prediction.
    - Updated the weights of the training samples based on the performance of the current weak learner.
v. We then repeated the above process for the specified number of weak learners or until a stopping criterion was met.

Evaluation on Test Set

After training the AdaBoost model on the training set:

- We used the trained model to predict the labels of the samples in the test set.
- Compared the predicted labels to the true labels of the test set to compute the accuracy.
- Stored the accuracy obtained for this fold.

Repeat for Each Fold

We repeated the process for each fold, storing all accuracy values. At the end of the iteration, we computed the average accuracy and standard deviation.

6.2 Random forest

Below is an overview of how we implemented Random Forest:

Step 1: Decision Tree Algorithm

First, we needed to implement a basic decision tree algorithm. A decision tree splits the dataset based on the feature that results in the highest information gain (or

lowest impurity). This process is recursively repeated in a divide-and-conquer manner until a stopping criterion is met (e.g., maximum depth).

- **Entropy or Gini Impurity**: Calculated the impurity of data to decide on the best split.
- **Best Criteria**: For each feature, we used different thresholds to find the split that results in the highest information gain or lowest impurity.
- **Recursive Splitting**: Once the best split is found, split the dataset and recursively apply the same process to each child node.
- **Stopping Criteria**: The recursion stops if the maximum depth is reached, the node has fewer samples than the minimum node size, or the node is pure (all samples belong to the same class).

Step 2: Introduce Randomness

Random Forest introduces randomness in two main ways to ensure that each decision tree in the forest is different:

- **Bootstrap Aggregating (Bagging):** For each tree, we used a different bootstrap sample of the data. A bootstrap sample is a random sample of the dataset with replacement, usually of the same size as the original dataset.
- **Feature Sampling:** At each split in the construction of a tree, randomly select a subset of the features to consider.

Step 3: Build the Forest

Construct multiple decision trees using the above processes described. Each tree is built on a different bootstrap sample of the dataset and uses random subsets of features for splitting.

- **Number of Trees**: Decided on the number of trees to include in random forest. Although, more trees generally increase prediction accuracy but also affects the computational cost should the dataset be large.

Step 4: Aggregate Predictions

Once all trees are constructed, we use them to make predictions. Then, averaging the predictions of all trees.

i. **Cross-Validation Setup**: once the algorithm was all set, we applied 10 times 5-fold cross-validation to each dataset, ensuring a comprehensive evaluation.

ii. **Model Training and Testing**: For each fold, we trained the Random Forest model on the training set and evaluated its performance on the testing set. Storing each accuracy in a list.
iii. **Performance Aggregation**: Calculated the average accuracy and standard deviation of the model across all folds and iterations to assess the overall performance.

6.3 ANN with back propagation

We implemented neural network class that can accept the input shape, number of hidden layers and output shape as an input and use this information to initiate an object of the class with the attribute passed to it. Find below details of implementation:

Step 1: Building the Neural Network

- **Initialization:** We initialized network parameters (weights and biases) for all layers except the input layer. The initialization methods used for weights are *He* initialization technique and *biases* (zeros initialization). Stored the initialized network parameters in the ***networkParameters*** dictionary, set the best loss to infinity, and initialized the *bestNetParameter* dictionary to store the best network parameters.
- **Activation Functions:** Defined different activation functions such as sigmoid, tanh, relu, and their derivatives.
- **Forward Propagation:** Implemented the *forward_propagation* method to compute the forward pass of the neural network. Iterate through each layer, compute the linear and activation functions, and store intermediate values in a *forward_cache* dictionary.
- **Cost Function:** Implemented the *compute_cost* method to calculate the cost (loss) of the neural network predictions. We used cross-entropy loss for classification.
- **Backward Propagation:** Implemented the *backward_propagation* method to compute the gradients of the network parameters with respect to the loss. We used the chain rule to propagate gradients backward through the network layers.
- **Parameter Updates:** Implemented the *update_parameters* method to update the network parameters (weights and biases) using gradient descent. Updated the parameters based on the gradients and the learning rate.
- **Prediction:** Implemented the *predict* method to make predictions using the trained neural network. Perform forward propagation and convert the output probabilities to class labels.

Step 2: Training

Implemented the train method to train the neural network using the given training data. Performed forward and backward propagation iteratively for a specified number of iterations. Then, Print the training progress, including the iteration number, cost, and training/test accuracy. Keep track of the best test accuracy and corresponding network parameters.

Step 3: Hyperparameter Tuning

Experimented with different hyperparameters such as learning rate, number iteration, number of hidden layer neurons, and network architecture. We tuned the hyperparameters to improve the performance of the neural network on the train set.

Step 4: Testing and Evaluation

We evaluated the trained neural network on the test set using the best network parameters. Calculated the final test accuracy and compared it with the training accuracy. Analyzed the model performance and adjust hyperparameters when necessary.

6.4 KNN

We implemented a K-Nearest Neighbor Class with two distance metrics: Manhattan distance (*predictLManhattan*) and Euclidean distance (*predictLEuclidean*). Find below the methods created:

- **Train Method:** The train method is used to train the KNN model. It takes input features X and corresponding labels Y and stores them as instance variables *xtr* and *ytr,* respectively.
- **Prediction with Manhattan Distance (*predictLManhattan*):** For each test data point the Manhattan distance from that point to all training data points is calculated. The indices of the top three nearest neighbors are obtained and the labels corresponding to these three nearest neighbors are retrieved. The counts of each label are calculated and the label with the maximum count (majority vote) is assigned as the predicted label for the test point. The accuracy of the predictions is computed as the ratio of correct predictions to the total number of test samples.
- **Prediction with Euclidean Distance (*predictLEuclidean*):** The implementation is similar to Manhattan distance, but the difference is the distance calculation. The rest of the process remains the same as in the Manhattan distance prediction. The accuracy of the predictions is calculated by comparing the predicted labels with

the true labels y. The number of correct predictions is divided by the total number of test samples.

7 Results

7.1 Adaboost on tree stumps

	Average accuracy	Standard deviation
Breast cancer data	0.9312353545734839	0.017977832292101512
Car data	0.6903816704364582	0.021568617694366403
Ecoli data	0.059701492537313425	0.02636346527262044
Letter Recognition data	0.03875199424856215	0.002432720689367345
Mushroom data	0.8476291474043198	0.019922603593645602

7.2 Random Forest

	Average accuracy	Standard deviation
Breast cancer data	0.9524501541623843	0.017429082932062908
Car data	0.8529757895618664	0.02001279782267196
Ecoli data	0.8074626865671639	0.05050425168862277
Letter Recognition data	0.3834890435108777	0.014409692385959254
Mushroom data	0.9834914437286851	0.00621899095711920

7.3 ANN with back propagation

	Average accuracy	Standard deviation
Breast cancer data	0.9046	0.0788
Car data	0.8512	0.0685
Ecoli data	0.7968	0.1034

Letter Recognition data	0.7344	0.1281
Mushroom data	0.9875	0.0332

7.4 KNN with Euclidean

	Average accuracy	**Standard deviation**
Breast cancer data	0.96	0.0075
Car data	0.84	0.0314
Ecoli data	0.81	0.0504
Letter Recognition data	0.89	0.0098
Mushroom data	1.0	0.00

7.5 KNN with Manhattan

	Average accuracy	**Standard deviation**
Breast cancer data	0.97	0.0133
Car data	0.85	0.0204
Ecoli data	0.79	0.0398
Letter Recognition data	0.88	0.008
Mushroom data	1.0	0.00

8 Analysis of results

8.1 Adaboost on tree stumps

Breast Cancer Data

- **Average Accuracy**: High (93.12%), indicating that AdaBoost is effective at boosting the performance of simple models like tree stumps in binary classification tasks.

- **Standard Deviation**: Low (1.80%), suggesting consistent performance across different validation folds, which speaks to the robustness of AdaBoost for this dataset.

Car Data

- **Average Accuracy**: Moderate (69.04%), which is significantly lower than the Breast Cancer dataset. This indicates that the decision boundaries for the Car data might be too complex for a tree stump to capture effectively, even after boosting.
- **Standard Deviation**: Low (2.16%), implying stable performance despite the moderate accuracy, indicating that while AdaBoost maintains consistency, the base learner's simplicity limits performance.

Ecoli Data

- **Average Accuracy**: Very Low (5.97%), which is significantly lower than the other datasets. This suggests that a single decision rule is far from sufficient to capture the patterns in the Ecoli data, and the AdaBoost algorithm cannot compensate for the weak performance of the base learner.
- **Standard Deviation**: Moderate (2.64%), reflecting some variability in performance, which could be due to the model occasionally catching some patterns by chance due to data variability.

Letter Recognition Data

- **Average Accuracy**: Very Low (3.88%), indicating that the model is essentially not learning the task at hand. Given the complexity of the task and the high number of classes, a tree stump is too simple a model to handle this dataset, even when combined in an ensemble with AdaBoost.
- **Standard Deviation**: Very Low (0.24%), suggesting that the model consistently fails to predict the correct class, further supporting the inadequacy of a tree stump for this problem.

Mushroom Data

- **Average Accuracy**: Good (84.76%), suggesting that AdaBoost can leverage even simple models like tree stumps to a certain extent on datasets with features that provide strong indicators of the target class.
- **Standard Deviation**: Low (1.99%), indicating that the ensemble's performance is fairly consistent across different subsets of the data.

Analysis Summary

- **Effectiveness of Boosting Simple Models**: AdaBoost demonstrates the ability to enhance the performance of simple models on some datasets, particularly those where the decision boundaries can be well-approximated by a series of rules (Breast Cancer and Mushroom data).
- The very low accuracies on Ecoli and Letter Recognition data indicate that when the underlying relationships in the data are complex and cannot be effectively captured by a simple tree stump, boosting methods like AdaBoost might not be sufficient to achieve good performance.
- **Variability**: The standard deviations generally remain low, indicating that the performance of AdaBoost is consistent despite the varying accuracy. This is typical for boosting algorithms, as they tend to produce stable ensembles by focusing on difficult-to-predict instances.
- **Dataset Complexity**: The results also highlight the importance of choosing a suitable model for the dataset's complexity. While tree stumps can be effective when boosted for simpler problems, more complex datasets require base learners that can capture more nuanced patterns.

In conclusion, while AdaBoost is a powerful algorithm that can turn weak learners into strong ones, its performance is heavily dependent on the complexity of the data and the adequacy of the base learner.

8.2 Random forest

Random Forest performs best on the Breast Cancer and Mushroom datasets. The high accuracy and low standard deviation indicate that the model has not only learned these datasets well but also generalizes consistently across different subsets of the data.

Breast Cancer Data

- **Average Accuracy**: High (95.25%), indicating that the Random Forest model predicts correctly with great reliability.
- **Standard Deviation**: Low (1.74%), suggesting that the model's performance is consistent across different folds or iterations of cross-validation.

Car Data

- **Average Accuracy**: Moderately High (85.30%), showing that the model performs well but with some room for improvement.

- **Standard Deviation**: Relatively Low (2.00%), indicating that the model's performance is quite stable across different evaluations.

Ecoli Data

- **Average Accuracy**: Good (80.75%), which is lower than Breast Cancer and Mushroom datasets but still indicates a decent level of predictive power.
- **Standard Deviation**: Moderate (5.05%), the highest among the datasets, implying that there is variability in the model's performance, which could be due to the nature of the data or fewer instances to learn from.

Letter Recognition Data

- **Average Accuracy**: Low (38.35%), which is significantly lower than the other datasets, indicating that Random Forest struggles to predict the correct category.
- **Standard Deviation**: Low (1.44%), suggesting that despite the overall low performance, the consistency of the model's prediction is relatively stable.

Mushroom Data

- **Average Accuracy**: Very High (98.35%), showing that Random Forest is extremely effective at classifying this dataset.
- **Standard Deviation**: Very Low (0.62%), indicating that the model's performance is highly consistent across different folds or iterations.

Analysis Summary

- **Consistency**: The relatively low standard deviations across all datasets (except Ecoli) suggest that Random Forest provides stable performance across different training and validation splits.
- **Challenging Dataset**: The Letter Recognition dataset presents a challenge for the Random Forest model, indicated by significantly lower accuracy. This could be due to high dimensionality, many classes, or complex patterns that are difficult to capture.
- **Possible Overfitting**: While the standard deviation for the Ecoli dataset is higher, suggesting some variability, it's still not indicative of overfitting, as overfitting would typically be accompanied by a much higher average accuracy on the training data.
- **Suitability and Complexity**: The results might suggest that Random Forest is more suited to certain types of problems, such as binary classification (e.g., Breast Cancer and Mushroom) or datasets with clear distinctions between classes. The model may struggle with more complex tasks like multi-class classification with a large number of categories (e.g., Letter Recognition).

- **Tuning and Improvements**: For datasets with lower accuracy (Letter Recognition dataset), further hyperparameter tuning, feature engineering, or data preprocessing might improve Random Forest's performance.

Overall, Random Forest shows excellent capability as a classification algorithm with robustness across varied datasets, though it may require careful consideration and adjustment for complex problems with many classes or intricate patterns.

8.3 ANN with back propagation

Breast Cancer Data

- **Average Accuracy**: High (90.46%), which suggests that the ANN model has a strong predictive capability for this dataset.
- **Standard Deviation**: Moderately High (7.88%), indicating there's some variability in the model's performance. This could be due to the complexity of the dataset or overfitting to the training data.

Car Data

- **Average Accuracy**: Moderately High (85.12%), showing the model performs well but may benefit from further optimization or tuning.
- **Standard Deviation**: Moderate (6.85%), which points to variability in the performance across different iterations, but not excessively so.

Ecoli Data

- **Average Accuracy**: Good (79.68%), but lower compared to Breast Cancer and Car data, suggesting that the model might be finding it more challenging to capture the patterns in this dataset.
- **Standard Deviation**: High (10.34%), the second highest among the datasets, indicating considerable inconsistency in the model's performance which may suggest the presence of complex or noisy data that is not being captured well by the model.

Letter Recognition Data

- **Average Accuracy**: Moderate (73.44%) suggests that this model may be finding it difficult to recognize letters because of their complexity and high dimensionality of this particular dataset.
- **Standard Deviation**: High (12.81%), the highest among the datasets, indicating the model's performance is quite variable, perhaps due to the complexity of the task and the possibility of different characters being harder to distinguish.

Mushroom Data

- **Average Accuracy**: Very High (98.75%), showing that the ANN model is exceptionally effective at classifying this dataset.

- **Standard Deviation**: Low to Moderate (3.32%), indicating relatively consistent performance across different iterations.

Analysis Summary

- **Strong Performer**: ANN with backpropagation is quite effective for most datasets, particularly excelling in the Mushroom and Breast Cancer datasets.
- **Complex Pattern Handling**: The relatively strong performance on the Letter Recognition dataset suggests that the ANN model, with its ability to capture complex, non-linear relationships, is well-suited for tasks involving complex patterns or high-dimensional data.
- **Consistency Issues**: The high standard deviations observed for the Ecoli and Letter Recognition datasets might point to the model overfitting to certain features or parts of the dataset. It could also reflect the challenges of optimizing neural networks, which can be sensitive to the choice of hyperparameters and training procedures.
- **Comparison with Random Forest**: When compared to Random Forest, ANN shows a significantly better performance on the Letter Recognition dataset, which could be due to the ANN's capacity to learn more complex, hierarchical representations of data.
- **Potential Overfitting**: The higher standard deviations across the datasets when compared to the Random Forest results might suggest that ANNs are more prone to overfitting, especially with more complex or smaller datasets.

The ANN with backpropagation shows promise and effectiveness in handling various types of classification tasks. However, its performance might be improved with careful tuning of the network architecture and training process, particularly regularization techniques to manage overfitting and ensure the model's generalizability.

8.4 (KNN) algorithm with Euclidean distance

Breast Cancer Data

- **Average Accuracy**: Very High (96%), indicating excellent predictive performance for this dataset, suggesting that the feature space allows for clear margins between classes.
- **Standard Deviation**: Extremely Low (0.75%), showing that the model performs with remarkable consistency across different cross-validation folds.

Car Data

- **Average Accuracy**: Good (84%), which is a solid result, although there's room for improvement compared to other algorithms or datasets.

- **Standard Deviation**: Moderate (3.14%), suggesting some variability in model performance which could potentially be improved with feature engineering or normalization.

Ecoli Data

- **Average Accuracy**: Good (81%), demonstrating a decent level of accuracy, though not as high as some other datasets or algorithms.
- **Standard Deviation**: Moderate to High (5.04%), indicating a fair amount of variability in the model's predictions, possibly due to complex relationships in the data that are harder to capture with Euclidean distance.

Letter Recognition Data

- **Average Accuracy**: High (89%), which is a strong performance, especially given the potential complexity of the dataset with many classes.
- **Standard Deviation**: Very Low (0.98%), suggesting consistent performance and that KNN with Euclidean distance is capturing the essential patterns effectively in this dataset.

Mushroom Data

- **Average Accuracy**: Perfect (100%), which is the optimal result and indicates that the dataset likely has features that allow for a clear separation of classes using Euclidean distance.
- **Standard Deviation**: Zero (0%), showing that the model's predictions are completely stable across all evaluations.

Analysis Summary

- **High Accuracy**: KNN with Euclidean distance shows very high accuracy, particularly on the Breast Cancer and Mushroom datasets. This suggests that for these datasets, the class boundaries are well-defined in the Euclidean space.
- **Consistency**: The standard deviations are generally low, indicating consistent model performance. The Breast Cancer and Letter Recognition datasets, in particular, show that KNN can reliably capture the underlying structure of the data.
- **Stability in Complexity**: The Letter Recognition data results are noteworthy as they indicate that KNN with Euclidean distance is effective even in multi-class classification scenarios, given low-dimensional, well-structured feature space.
- **Room for Improvement**: While the results are generally good, the Ecoli and Car data show higher standard deviations, which means there's more variability in the model performance, and these datasets may benefit from additional preprocessing or more careful selection of the 'k' parameter.

- **Algorithm Robustness**: KNN with Euclidean distance seems robust in handling different types of datasets, from binary classification to multi-class scenarios, provided the features are discriminative enough in Euclidean space.

The KNN algorithm with Euclidean distance proves to be a strong classifier, especially when the data is well-separated. However, its performance could be sensitive to the scale of the data, and hence proper scaling or normalization could further improve its effectiveness. Additionally, the choice of 'k', the number of neighbors to consider, plays a crucial role in the model's ability to generalize and should be tuned accordingly.

8.5 (KNN) algorithm with Euclidean distance

Analysis Summary

- **Consistently High Accuracy**: KNN with Manhattan distance showcases very high accuracy, especially on the Breast Cancer and Mushroom datasets. The perfect scores on the Mushroom dataset reaffirm the alignment of features with the Manhattan metric.
- **Comparative Performance**: When comparing the results with the Euclidean distance, we see a similar pattern of performance with minor differences in accuracy and consistency. The Manhattan distance metric seems to offer a slight edge for the Breast Cancer data but a minor decrease in performance for the Ecoli and Letter Recognition datasets.
- **Performance Stability**: The standard deviations are generally low for all datasets, highlighting stable model performance with the Manhattan distance. It is particularly impressive in how consistent it is for the Letter Recognition data.
- **Potential Data Sensitivity**: The slightly higher standard deviation for the Ecoli data might suggest sensitivity to the feature scale or the presence of outliers.

Overall, KNN with Manhattan distance performs very well across a range of datasets. It achieves particularly outstanding results with the Breast Cancer and Mushroom datasets and offers robust and stable predictions for complex multi-class classification problems like Letter Recognition. The choice between Manhattan and Euclidean distance should be based on the specific structure of the dataset, and both may require tuning the number of neighbors 'k' to achieve optimal results.

8.6 Comparing all the Algorithms implemented

To compare the analyses of the different algorithms, we must look at their performance on the five datasets: Breast Cancer, Car, Ecoli, Letter Recognition, and Mushroom (Figure 1).

Breast Cancer Data

- **Random Forest** and **KNN with Manhattan** distance show the best performance, suggesting these methods are well-suited to binary classification tasks with clear decision boundaries.
- **AdaBoost** also performs well but slightly less than the other algorithms, which could be due to the simplicity of the tree stump base learner.
- **ANN** shows good performance, though slightly lower, indicating its capturing complex patterns effectively but might be slightly prone to overfitting.
- **KNN with Euclidean** distance has a performance similar to ANN, suggesting that in this case, the choice of distance metric may not be as critical.

Car Data

- **Random Forest** and **ANN** perform similarly well, indicating that the task benefits from more complex models capable of capturing intricate patterns.
- **AdaBoost** shows moderate performance, hinting at the limitations of a simple base learner for a multi-class classification problem with possibly non-linear decision boundaries.
- **KNN with both distances** performs fairly, yet not as well as the more complex models, which could be due to the high-dimensional feature space typical for this dataset.

Ecoli Data

- **Random Forest** provides reasonable performance, although with some variability, suggesting that it can handle the complexity to a certain extent.
- **KNN with Euclidean** distance performs slightly better than with Manhattan, which may reflect the structure of the data aligning more closely with Euclidean geometry.
- **ANN** also achieves a good accuracy level but with considerable variability, pointing to potential overfitting.
- **AdaBoost** shows very low performance, indicating that the simplicity of a tree stump is inadequate for the dataset's complexity.

Letter Recognition Data

- **ANN** demonstrates significant superiority in this dataset, likely due to its ability to model complex relationships in high-dimensional spaces.
- **KNN with Euclidean** distance also does well, possibly because the spatial distribution of features aligns with the Euclidean geometry.
- **KNN with Manhattan** has slightly lower performance, suggesting the feature relationships might be more diagonal than axis-aligned.

- **Random Forest** has the lowest accuracy among the more complex models, which could be due to the model being unable to capture all the nuances of the data.
- **AdaBoost** performs poorly, highlighting that boosting a tree stump is not sufficient for such a complex multi-class problem.

Mushroom Data

- **Random Forest**, **KNN with both distances**, and **ANN** show excellent to perfect accuracies, implying that the decision boundaries in this dataset are well-defined and can be easily modeled by various algorithms.
- **AdaBoost**'s good performance suggests that even simple rules can be powerful when correctly combined, though it falls short of the perfect scores achieved by the other methods.

Overall Comparison

- **Random Forest** is consistently robust across all datasets, especially excelling in the Mushroom dataset. Its ensemble approach appears to generalize well.
- **ANNs** show great potential in datasets with complex, non-linear patterns, particularly in the Letter Recognition dataset.
- **KNN with Euclidean** distance generally performs well but may struggle with more complex or high-dimensional data unless well-tuned.
- **KNN with Manhattan** distance performs comparably to Euclidean in most cases but may not always capture the intricacies of the data's structure.
- **AdaBoost with a tree stump** is effective on simpler problems but falls short on more complex datasets.

Each algorithm has its strengths and weaknesses, and the choice of algorithm often depends on the dataset's characteristics and the problem at hand. Random Forest and ANN seem to offer more complexity and flexibility for various tasks, while KNN can be effective with well-structured data. AdaBoost's performance heavily relies on the base learner's suitability to the data.

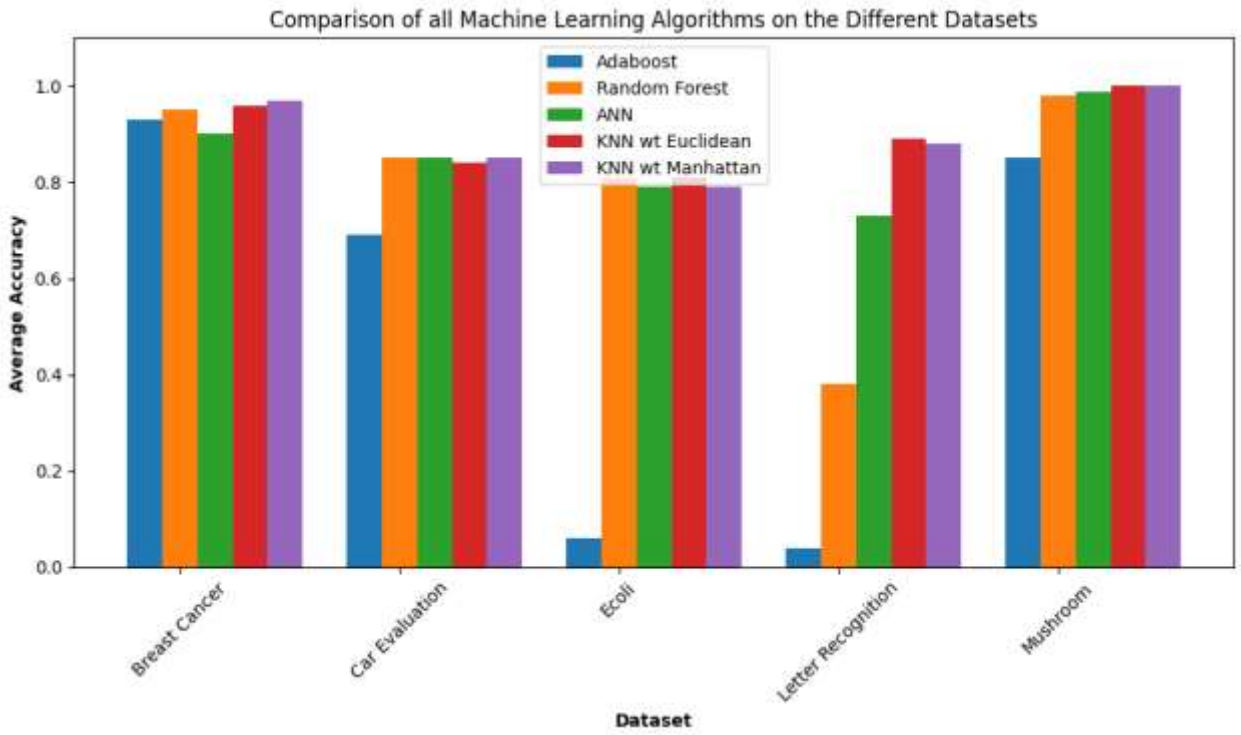

Figure 1: *Comparison of the 4 Algorithms accuracy with the 5 datasets.*

9 Appendix

9.1 Adaboost with tree stump

```
from sklearn.model_selection import KFold
from sklearn.utils import shuffle
import pandas as pd
import numpy as np
from sklearn.metrics import accuracy_score
# Load the CSV file into a DataFrame (replace 'file.csv' with the actual path)
df = pd.read_csv('breast-cancer-wisconsin.data')
print(f"{df}")
#Trim ID from dataset
df = df.iloc[:, 1:]

print(f"headers:{df.head()}")
# Remove the 'feature6' column from the DataFrame
df = df.drop('1.3', axis=1)

#trim the last column
data=df.iloc[:, :-1]
```

```
target= df.iloc[:,-1 ]

# Convert the DataFrame into a dictionary with 'data' key containing values as
lists
breastcancer_data = data.values
breastcancer_target = target.values

 #print(breastcancer_data)
# print(breastcancer_target)
X = breastcancer_data
# print(f"X :{X}")
y = breastcancer_target
y[y==2]=-1
y[y==4]=1

# print(f"X:{X}")
# print(f"y:{y}")

class DecisionStump:
    def __init__(self):
        self.polarity = 1
        self.threshold = None
        self.feature_idx = None
        self.alpha = None

    def predict(self,X):
        n_samples = X.shape[0]
        X_c = X[:,self.feature_idx]
        preds = np.ones(n_samples)

        if self.polarity ==1:
            preds[X_c < self.threshold] = -1
        else:
            preds[X_c > self.threshold] = -1

        return preds
class myAdaBoost:
    def __init__(self,n_clf=5):
        self.n_clf = n_clf

    def fit(self,X,y):
        n_samples,n_features = X.shape
        w = np.full(n_samples, (1/n_samples))

        self.clfs=[]
```

```
        for _ in range(self.n_clf):
            clf = DecisionStump()
            min_error = float('inf')
            for feat in range(n_features):
                X_c = X[:,feat]
                thresholds=np.unique(X_c)
                for threshold in thresholds:
                    p=1
                    preds=np.ones(n_samples)
                    preds[X_c<threshold]=-1

                    misclassified = w[y!=preds]
                    error=sum(misclassified)
def accuracy(y_true, y_pred):
    accuracy = np.sum(y_true == y_pred) / len(y_true)
    return accuracy

                    if error >0.5:
                        p=-1
                        error=1-error

                    if error<min_error:
                        min_error=error
                        clf.threshold=threshold
                        clf.feature_idx=feat
                        clf.polarity=p

            EPS=1e-10
            clf.alpha=0.5*np.log((1.0-min_error+EPS)/(min_error+EPS))
            preds = clf.predict(X)
            w *= np.exp(-clf.alpha*y*preds)
            w/=np.sum(w)
            self.clfs.append(clf)

    def predict(self,X):
        clf_preds = [clf.alpha*clf.predict(X) for clf in self.clfs]
        y_pred = np.sum(clf_preds,axis=0)
        y_pred = np.sign(y_pred)
        return y_pred
# Assuming X, y are your features and labels
accuracies = []

for iteration in range(10):
    X, y = shuffle(X, y, random_state=iteration)  # Shuffle the dataset
    kf = KFold(n_splits=5, shuffle=False)
```

```
    for train_index, test_index in kf.split(X):
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]

        clf = myAdaBoost()
        clf.fit(X_train, y_train)

        # Predict on the test part of the dataset
        predictions = clf.predict(X_test)

        # Compute accuracy and add it to the accuracies list
        accuracy=accuracy_score(y_test, predictions)
        accuracies.append(accuracy)

        # y_pred = clf.predict(X_test)
        # acc = accuracy(y_test, y_pred)
        print(f"accuracy:{accuracy}")

    print(f"----------------------------------------------------------------------")
# Calculate the average accuracy and standard deviation
average_accuracy = np.mean(accuracies)
std_dev_accuracy = np.std(accuracies)
print(f"Average Accuracy: {average_accuracy}")
print(f"Standard Deviation: {std_dev_accuracy}")
```

9.2 Random forest

"""Import Libraries"""
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import KFold
from sklearn.metrics import accuracy_score
from sklearn.utils import shuffle

"""Import Data"""
dataset_path = "/content/breast-cancer.data"
dataset_path1 = "/content/car.data"

```
dataset_path2 = "/content/ecoli.data"
dataset_path3 = "/content/letter-recognition.data"
dataset_path4 = "/content/mushroom.data"

df1 = pd.read_csv(dataset_path) #Breast Cancer Dataset
df2 = pd.read_csv(dataset_path1) #Car Dataset
df3 = pd.read_csv(dataset_path2, sep = '\s+') #Ecoli Dataset
df4 = pd.read_csv(dataset_path3) #Letter Recognition Dataset
df5 = pd.read_csv(dataset_path4) #Mushroom Dataset
df1.head()
df1.dtypes
print(df1['1.1'].dtype)

# Get the data types of all attributes
dtypes = df1.dtypes

# Check if each attribute is numeric or not
for attribute, dtype in dtypes.items():
  if dtype == 'object':
    print(f'{attribute}: Discrete')
  elif pd.api.types.is_numeric_dtype(dtype):
    print(f'{attribute}: Continuous')

print(df4)

"""# Exploratory Data Analysis

Perform Dataset Preprocessing on each datasets. Check for missing values in the datasets

Breast Cancer Dataset
"""

df1.columns = ['ID', 'Radius', 'Texture', 'Perimeter', 'Smoothness', 'Compactness', 'Concavity',
'Concave Points', 'Symmetry', 'Fractal Dimension', 'Class']
df1.drop(['ID'], axis = 1, inplace = True)
df1.head()

print(df1.info())
print(df2.info())
print(df3.info())
print(df4.info())
print(df5.info())

#Car
```

```
df2.columns = ['Buying', 'Maint', 'Doors', 'Persons', 'Lug_Boots', 'Safety', 'Class']
df2.head()
print(df2['Class'])

#E Coli
df3.columns = ['Sequence', 'Mcg', 'Gvh', 'Lip', 'Chg', 'Aac', 'Alm1', 'Alm2', 'Class']
df3.drop(["Sequence"],axis = 1 ,inplace=True)
df3.head()

#Letter Recognition
df4.columns =['Class','x-box', 'y-box', 'width', 'high',  'onpix', 'x-bar', 'y-bar', 'x2bar', 'y2bar',
'xybar', 'x2ybr', 'xy2br', 'x-ege', 'xegvy' , 'y-ege', 'yegvx']
df4.head()
print(df4['Class'])

#Mushroom
df5.columns =['Class','cap-shape', 'cap-surface', 'cap-color', 'bruises', 'odor', 'gill-
attachment', 'gill-spacing', 'gill-size', 'gill-color', 'stalk-shape', 'stalk-root', 'stalk-surface-
above-ring', 'stalk-surface-below-ring', 'stalk-color-above-ring', 'stalk-color-below-ring',
'veil-type', 'veil-color', 'ring-number', 'ring-type', 'spore-print-color', 'population', 'habitat']
df5.head()
print(df5['Class'])

"""Random Forest Implementation"""

class TreeNode:
  def __init__(self, feature=None, threshold=None, left=None, right=None, *, value=None):
    self.feature = feature
    self.threshold = threshold
    self.left = left
    self.right = right
    self.value = value

  def is_leaf_node(self):
    return self.value is not None

# Implement DecisionTree as basic learner for Random Forest
class DecisionTree:
  def __init__(self, max_depth=10, min_samples_split=2, n_feats=None):
    self.max_depth = max_depth
    self.min_samples_split = min_samples_split
    self.n_feats = n_feats
    self.root = None
```

```
  def fit(self, X, y):
    self.n_feats = X.shape[1] if not self.n_feats else min(self.n_feats, X.shape[1])
    self.root = self._grow_tree(X, y)

  def predict(self, X):
    return np.array([self._traverse_tree(x, self.root) for x in X])

  def _grow_tree(self, X, y, depth=0):
    n_samples, n_features = X.shape
    n_labels = len(np.unique(y))

    # Stopping criteria
    if (depth >= self.max_depth or n_labels == 1 or n_samples < self.min_samples_split):
      leaf_value = self._most_common_label(y)
      return TreeNode(value=leaf_value)

    feat_idxs = np.random.choice(n_features, self.n_feats, replace=False) #feature
randomness: randomly select a subset of features to consider.

    # Greedily select the best split according to Gini impurity
    best_feat, best_thresh = self._best_criteria(X, y, feat_idxs)

    # Grow the children recursively
    left_idxs, right_idxs = self._split(X[:, best_feat], best_thresh)
    left = self._grow_tree(X[left_idxs, :], y[left_idxs], depth+1)
    right = self._grow_tree(X[right_idxs, :], y[right_idxs], depth+1)
    return TreeNode(best_feat, best_thresh, left, right)

  def _best_criteria(self, X, y, feat_idxs):
    best_gain = -1
    split_idx, split_thresh = None, None
    for feat_idx in feat_idxs:
      X_column = X[:, feat_idx]
      thresholds = np.unique(X_column)
      for threshold in thresholds:
        gain = self._information_gain(y, X_column, threshold)
        if gain > best_gain:
          best_gain = gain
          split_idx = feat_idx
          split_thresh = threshold
    return split_idx, split_thresh

  def _information_gain(self, y, X_column, split_thresh):
    # Parent loss
```

```
        parent_loss = self._gini(y)

        # Generate split
        left_idxs, right_idxs = self._split(X_column, split_thresh)
        if len(left_idxs) == 0 or len(right_idxs) == 0:
            return 0

        # Weighted average of the loss for the children
        n = len(y)
        n_l, n_r = len(left_idxs), len(right_idxs)
        e_l, e_r = self._gini(y[left_idxs]), self._gini(y[right_idxs])
        child_loss = (n_l / n) * e_l + (n_r / n) * e_r

        # Information gain is difference in loss before vs. after split
        ig = parent_loss - child_loss
        return ig

    def _split(self, X_column, split_thresh):
        left_idxs = np.argwhere(X_column < split_thresh).flatten()
        right_idxs = np.argwhere(X_column >= split_thresh).flatten()
        return left_idxs, right_idxs

    def _gini(self, y):
        _, counts = np.unique(y, return_counts=True)
        p = counts / counts.sum()
        gini = 1 - sum(p**2)
        return gini

    def _most_common_label(self, y):
        counter = Counter(y)
        most_common = counter.most_common(1)[0][0]
        return most_common

    def _traverse_tree(self, x, node):
        if node.is_leaf_node():
            return node.value
        if x[node.feature] < node.threshold:
            return self._traverse_tree(x, node.left)
        return self._traverse_tree(x, node.right)

from collections import Counter
class RandomForest:
    def __init__(self, n_trees=100, max_depth=10, min_samples_split=2, n_feats=None):
        self.n_trees = n_trees
```

```
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.n_feats = n_feats
        self.trees = []

    def fit(self, X, y):
        self.trees = []
        for _ in range(self.n_trees):
            tree = DecisionTree(max_depth=self.max_depth,
min_samples_split=self.min_samples_split, n_feats=self.n_feats)
            X_sample, y_sample = self._bootstrap_sample(X, y)
            # selected_features = self._select_features(X)
            tree.fit(X_sample, y_sample)
            self.trees.append(tree)

    def predict(self, X):
        tree_preds = np.array([tree.predict(X) for tree in self.trees])
        tree_preds = np.swapaxes(tree_preds, 0, 1)
        y_pred = [self._most_common_label(tree_pred) for tree_pred in tree_preds]
        return np.array(y_pred)

    def _bootstrap_sample(self, X, y):  #bagging: create random sample with replacement of
the dataset.
        n_samples = X.shape[0]
        idxs = np.random.choice(n_samples, size=n_samples, replace=True)
        return X[idxs], y[idxs]

    def _most_common_label(self, y):
        counter = Counter(y)
        most_common = counter.most_common(1)[0][0]
        return most_common

    def _select_features(self, X_train):
        # num_features = X_train.shape[0]
        num_features = X_train.shape[1]
        selected_features = np.random.choice(num_features, size=int(np.sqrt(num_features)),
replace=False)
        return selected_features

"""Breast Cancer Dataset"""

X = df1.drop('Class', axis=1) #features
y = df1['Class']          #target variable
```

```
X_np = X.values  # Convert DataFrame to numpy array if X is a DataFrame
y_np = y.values

accuracies = []

for iteration in range(10):
    X, y = shuffle(X_np, y_np, random_state=iteration)  # Shuffle the dataset
    kf = KFold(n_splits=5, shuffle=True)  # We already shuffled then perform KFold

    for train_index, test_index in kf.split(X):
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]

        # Convert y to binary if needed, e.g., benign (2) -> 0, malignant (4) -> 1
        y_train_binary = (y_train == 4).astype(int)
        y_test_binary = (y_test == 4).astype(int)

        # print(y_train_binary)
        # print(y_test_binary)

        # Train RandomForest using the training part of the dataset
        rf = RandomForest(n_trees=20, max_depth=5, min_samples_split=2, n_feats=None)
        rf.fit(X_train, y_train_binary)

        # Predict on the test part of the dataset
        predictions = rf.predict(X_test)

        # Compute accuracy and add it to the accuracies list
        accuracies.append(accuracy_score(y_test_binary, predictions))

# Calculate the average accuracy and standard deviation
average_accuracy = np.mean(accuracies)
std_dev_accuracy = np.std(accuracies)

print(accuracies)
print(np.array(accuracies).shape)
print(f"Average Accuracy: {average_accuracy}")
print(f"Standard Deviation: {std_dev_accuracy}")
```

9.3 ANN with back propagation

```
from google.colab import files
uploaded = files.upload()
```

```
import time
import random
import numpy as np
#import matplotlib.pyplot as plt
#from tensorflow.keras.datasets import mnist
#from keras.utils import to_categorical
import pandas as pd
from sklearn.model_selection import train_test_split,KFold
from sklearn.preprocessing import LabelEncoder,OneHotEncoder
```

Neural Network Class

```
class NueralNetwork():
  def __init__(self, layer_param):
    self.layer_param = layer_param
    #dictionary for weight and biases
    self.networkParameters={}
    self.bestNetParameter={}
    self.bestLoss = float('inf')
    #get number of layers in Neural network
    self.networkLength = len(layer_param)
    #intilialize network parameters for all layers excluding the input layer
    for l in range(1, self.networkLength):
      self.networkParameters['W' + str(l)] = np.random.randn(layer_param[l],layer_param[l-
1]) / np.sqrt(layer_param[l-1]) #*0.01
      self.networkParameters['b' + str(l)] = np.zeros((layer_param[l],1))

  def softmax(self,z):
   expZ = np.exp(z)
   return expZ/(np.sum(expZ, 0))
  def sigmoid(self,Z):
    A = 1/(1+np.exp(-Z))
    return A
  def relu(self,Z):
    A = np.maximum(0,Z)
    return A
  def derivative_relu(self,Z):
    return np.array(Z > 0, dtype = 'float')
```

```
  def forward_propagation(self,X, activation):
    forward_cache = {}
    L = len(self.networkParameters) // 2

    forward_cache['A0'] = X

    for l in range(1, L):
      forward_cache['Z' + str(l)] = self.networkParameters['W' + str(l)].dot(forward_cache['A'
+ str(l-1)]) + self.networkParameters['b' + str(l)]
      #forward_cache['Z' + str(l)] = parameters['W' + str(l)].dot(forward_cache['A' + str(l-1)])
+ parameters['b' + str(l)]

      if activation == 'tanh':
        forward_cache['A' + str(l)] = self.tanh(forward_cache['Z' + str(l)])
      else:
        forward_cache['A' + str(l)] = self.relu(forward_cache['Z' + str(l)])


    forward_cache['Z' + str(L)] = self.networkParameters['W' + str(L)].dot(forward_cache['A'
+ str(L-1)]) + self.networkParameters['b' + str(L)]

    if forward_cache['Z' + str(L)].shape[0] == 1:
      forward_cache['A' + str(L)] = self.sigmoid(forward_cache['Z' + str(L)])
    else :
      forward_cache['A' + str(L)] = self.softmax(forward_cache['Z' + str(L)])
    return forward_cache['A' + str(L)], forward_cache

  def compute_cost(self,AL, Y):
    m = Y.shape[1]
    if Y.shape[0] == 1:
      cost = (1./m) * (-np.dot(Y,np.log(AL).T) - np.dot(1-Y, np.log(1-AL).T))
    else:
      cost = -(1./m) * np.sum(Y * np.log(AL))

    cost = np.squeeze(cost)     # To make sure  cost's shape is what we expect.

    return cost
```

```
  def backward_propagation(self, AL, Y, forward_cache, activation):
    grads = {}
    L = len(self.networkParameters)//2
    m = AL.shape[1]
    grads["dZ" + str(L)] = AL - Y
    grads["dW" + str(L)] = 1./m * np.dot(grads["dZ" + str(L)],forward_cache['A' + str(L-1)].T)
    grads["db" + str(L)] = 1./m * np.sum(grads["dZ" + str(L)], axis = 1, keepdims = True)
    for l in reversed(range(1, L)):
      if activation == 'tanh':
        grads["dZ" + str(l)] = np.dot(self.networkParameters['W' + str(l+1)].T,grads["dZ" +
str(l+1)])*self.derivative_tanh(forward_cache['A' + str(l)])
      else:
        grads["dZ" + str(l)] = np.dot(self.networkParameters['W' + str(l+1)].T,grads["dZ" +
str(l+1)])*self.derivative_relu(forward_cache['A' + str(l)])

      grads["dW" + str(l)] = 1./m * np.dot(grads["dZ" + str(l)],forward_cache['A' + str(l-1)].T)
      grads["db" + str(l)] = 1./m * np.sum(grads["dZ" + str(l)], axis = 1, keepdims = True)

    return grads

  def update_parameters(self, grads, learning_rate):

    L = len(self.networkParameters) // 2

    for l in range(L):
      self.networkParameters["W" + str(l+1)] = self.networkParameters["W" + str(l+1)] -
learning_rate * grads["dW" + str(l+1)]
      self.networkParameters["b" + str(l+1)] = self.networkParameters["b" + str(l+1)] -
learning_rate * grads["db" + str(l+1)]

  def predict(self, X, y, activation):

    m = X.shape[1]
    y_pred, caches = self.forward_propagation(X, activation)
    if y.shape[0] == 1:
      y_pred = np.array(y_pred > 0.5, dtype = 'float')
    else:
```

```
        y = np.argmax(y, 0)
        y_pred = np.argmax(y_pred, 0)
      return np.round(np.sum((y_pred == y)/m), 2)

    def train(self,X,Y,X_test,y_test, learning_rate, num_iteration, activation = 'relu'):
      np.random.seed(1)
      bestTestAcc = 0

      #parameters = initialize_parameters(layers_dims)

      for i in range(0, num_iteration):
        AL, forward_cache = self.forward_propagation(X, activation)
        cost = self.compute_cost(AL, Y)
        grads = self.backward_propagation(AL, Y, forward_cache, activation)
        #print(f"First {grads}")
        self.update_parameters(grads, learning_rate)
        if i % (num_iteration/10) == 0:
          testAcc = self.predict(X_test, y_test, activation)

          #get best Accuracy on test data
          if bestTestAcc < testAcc :
            bestTestAcc = testAcc
          print("\niter:{} \t cost: {} \t train_acc:{} \t test_acc:{}".format(i, np.round(cost, 2),
self.predict(X, Y, activation), testAcc))
        if i % 10 == 0:
          print("==", end = '')

      return bestTestAcc,forward_cache


1) Breast Cancer data Set

#Covert Concavity from object value to float
df1['Concavity'] = pd.to_numeric(df1['Concavity'], errors='coerce')

# Convert NaN values to zero (optional, depends on your preference)
df1['Concavity'].fillna(0, inplace=True)
```

```
# Convert the column to integer type
df1['Concavity'] = df1['Concavity'].astype(int)

# Select Features and Target value
X = df1.iloc[:, :-1] #features
y = df1.iloc[:, -1] #target

y = y.values.reshape(X.shape[0],1)

 # Convert numerical labels to categorical values
y = np.array(y > 2, dtype = 'float')
```

2) Car Dataset

```
#Ecoli data Set
#df1 = pd.read_csv("ecoli_new.csv") #Car Dataset



# Select Features and Target value
X = df2.iloc[:, :-1] #features
y = df2["Class"] #target

# Step 1: Label encode the categorical labels
label_encoder = LabelEncoder()
integer_encoded = label_encoder.fit_transform(y)
df_label_encoded = X.copy()
for col in X.columns:
   #print(f"{col}:  {df1[col].dtype}")
   if df2[col].dtype == 'object':  # Only encode categorical columns
      df_label_encoded[col] = label_encoder.fit_transform(df2[col])
X =  df_label_encoded

# Step 2: One-hot encode the integer-encoded labels
onehot_encoder = OneHotEncoder(sparse=False)
integer_encoded = integer_encoded.reshape(len(integer_encoded), 1)
y = onehot_encoder.fit_transform(integer_encoded)
print(y.shape)
```

3) Ecoli data Set

```
#Ecoli data Set
#df1 = pd.read_csv("ecoli_new.csv") #Car Dataset



# Select Features and Target value
X = df3.iloc[:, :-1] #features
y = df3.iloc[:, -1] #target
#print(X.shape)
#y=np.squeeze(y)

# Step 1: Label encode the categorical labels
label_encoder = LabelEncoder()
integer_encoded = label_encoder.fit_transform(y)
df_label_encoded = X.copy()
for col in X.columns:
   #print(f"{col}:  {df1[col].dtype}")
   if df3[col].dtype == 'object':  # Only encode categorical columns
      df_label_encoded[col] = label_encoder.fit_transform(df1[col])
X =  df_label_encoded

# Step 2: One-hot encode the integer-encoded labels
onehot_encoder = OneHotEncoder(sparse=False)
integer_encoded = integer_encoded.reshape(len(integer_encoded), 1)
y = onehot_encoder.fit_transform(integer_encoded)
print(y.shape)
```

4) Letter Recognition Data Set

```
#df4.head()

# Select Features and Target value
#print(f"first {df1.head()}")
X = df4.drop("Class", axis=1) #features
y = df4['Class'] #target
#print(y.head())
#print(X.head())
```

```
# Get unique values in the 'Column_Name' column
unique_values = y.unique()

# Step 1: Label encode the categorical labels
label_encoder = LabelEncoder()
integer_encoded = label_encoder.fit_transform(y)

# Step 2: One-hot encode the integer-encoded labels
onehot_encoder = OneHotEncoder(sparse=False)
integer_encoded = integer_encoded.reshape(len(integer_encoded), 1)
y = onehot_encoder.fit_transform(integer_encoded)
print(y.shape)
```

5) Mushroom Data

```
# Select Features and Target value
X = df5.iloc[:, :-1] #features
y = df5["Class"] #target

#Step 1: Label encode the categorical labels
label_encoder = LabelEncoder()
integer_encoded = label_encoder.fit_transform(y)
y = integer_encoded
df_label_encoded = X.copy()
for col in X.columns:
  #print(f"{col}:  {df1[col].dtype}")
  if df5[col].dtype == 'object':  # Only encode categorical columns
    df_label_encoded[col] = label_encoder.fit_transform(df5[col])
X =  df_label_encoded

#Reshape y value
y = y.reshape(X.shape[0],1)
#print(y)
# Convert numerical labels to categorical values
#y = np.array(y > 2, dtype = 'float')

print(y.shape)
```

NN-Declaration

```
#Declare Model parameters
layers_dims = [X.shape[1], 40, 40, y.shape[1]] #  4-layer model
lr = 0.01
iters = 100
myNN = NueralNetwork(layers_dims)
#Intialize K-fold
kf = KFold(n_splits=5, shuffle=True, random_state=0)
#Declare Accuracy List
testAccList=[]
for i in range(10):
 for train_index, test_index in kf.split(X):
    # Split data into training and test sets for the current fold
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y[train_index], y[test_index]

    # # Randomly select 32 samples from the training set
    # X_train_batch = X_train.sample(n=32, random_state=42)
    # y_train_batch = y_train[X_train_batch.index]



    # # Randomly select 32 samples from the training set
    # X_test_batch = X_test.sample(n=32, random_state=42)
    # y_test_batch = y_test[X_test_batch.index]



    X_train_conv = X_train.values
    X_test_conv = X_test.values
    X_train_conv =X_train_conv.T
    X_test_conv = X_test_conv.T
    y_train = y_train.T
    y_test = y_test.T
    #print(f"XTRAIN: {X_train_conv.shape}")
    #print(f"YTRAIN: {y_train.shape}")
    # Train the model
    bestTestAcc,fp = myNN.train(X_train_conv,y_train,X_test_conv,y_test,0.001,1000)
```

```
    testAccList.append(bestTestAcc)
    #break;

print(testAccList)
print(f"Average Accuracy is: {np.mean(testAccList)}")
print(f"Standard Deviation for Accuracy is: {np.std(testAccList)}")
```

9.4 kNN

```
import numpy as np
import tensorflow as tf
from sklearn.model_selection import train_test_split,KFold
import pandas as pd
from sklearn.preprocessing import LabelEncoder,OneHotEncoder

# Set device to GPU if available
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

class KNearestNeighbor:
 def _init_(self):
  pass

 def train(self, X, Y):
  self.xtr=X
  self.ytr=Y

 def predictLManhattan(self, X,y):
  num_test = X.shape[0]

  Ypred = np.zeros(num_test, dtype=self.ytr.dtype)

  for i in range(num_test):
   distances= np.sum(np.abs(self.xtr - X[i,:]), axis=1)

   #min_index = np.argmin(distances)
   #Ypred[i] = self.ytr[min_index]
   #Get indices of the minimum 3
   indices = np.argsort(distances)[3]
```

```
    #Get label of the minimum 3
    k_nearest_labels = self.ytr[indices]
    #print(k_nearest_labels)
    # get count of each label
    k_nearest_labels = k_nearest_labels.astype(int)
    counts = np.bincount(k_nearest_labels)

    # Get the label with the maximum count (majority vote)
    Ypred[i] = np.argmax(counts)
  #return Ypred
  return np.round(np.sum((Ypred == y.squeeze())/num_test), 2)

 def predictLEuclidean(self, X,y):
  num_test = X.shape[0]

  Ypred = np.zeros(num_test, dtype=self.ytr.dtype)

  for i in range(num_test):

    #distances= np.sum(np.abs(self.xtr - X[i,:]), axis=1)
    distances = np.sqrt(np.sum((self.xtr - X[i, :]) ** 2, axis=1))
    #min_index = np.argmin(distances)
    #Ypred[i] = self.ytr[min_index]
    #Get indices of the minimum 3
    indices = np.argsort(distances)[3]
    #Get label of the minimum 3
    k_nearest_labels = self.ytr[indices]
    k_nearest_labels = k_nearest_labels.astype(int)

    # get count of each label
    counts = np.bincount(k_nearest_labels)

    # Get the label with the maximum count (majority vote)
    Ypred[i] = np.argmax(counts)
  return np.round(np.sum((Ypred == y.squeeze())/num_test), 2)
  #return Ypred
```

Import Datasets

```
dataset_path = "/content/breast-cancer-wisconsin.data"
dataset_path1 = "/content/car_new.csv"
dataset_path2 = "/content/ecoli.data"
dataset_path3 = "/content/letter-recognition.data"
dataset_path4 = "/content/mushroom.data"

df1 = pd.read_csv(dataset_path) #Breast Cancer Dataset
df2 = pd.read_csv(dataset_path1) #Car Dataset
df3 = pd.read_csv(dataset_path2, sep = '\s+') #Ecoli Dataset
df4 = pd.read_csv(dataset_path3) #Letter Recognition Dataset
df5 = pd.read_csv(dataset_path4) #Mushroom Dataset
df1.head()



#Breast Cancer data Set
df1.columns = ['ID', 'Radius', 'Texture', 'Perimeter', 'Smoothness', 'Compactness', 'Concavity',
'Concave Points', 'Symmetry', 'Fractal Dimension', 'Class']
df1.drop(['ID'], axis = 1, inplace = True)
#df1.head()

#Car
df2.columns = ['Buying', 'Maint', 'Doors', 'Persons', 'Lug_Boots', 'Safety', 'Class']
#df2.head()
#print(df2['Class'])

#E Coli
df3.columns = ['Sequence', 'Mcg', 'Gvh', 'Lip', 'Chg', 'Aac', 'Alm1', 'Alm2', 'Class']
df3.drop(["Sequence"],axis = 1 ,inplace=True)
#df3.head()

#Letter Recognition
df4.columns =['Class','x-box', 'y-box', 'width', 'high',  'onpix', 'x-bar', 'y-bar', 'x2bar', 'y2bar',
'xybar', 'x2ybr', 'xy2br', 'x-ege', 'xegvy' , 'y-ege', 'yegvx']
#df4.head()
#print(df4['Class'])

#Mushroom
```

```
df5.columns =['Class','cap-shape', 'cap-surface', 'cap-color', 'bruises', 'odor', 'gill-attachment', 'gill-spacing', 'gill-size', 'gill-color', 'stalk-shape', 'stalk-root', 'stalk-surface-above-ring', 'stalk-surface-below-ring', 'stalk-color-above-ring', 'stalk-color-below-ring', 'veil-type', 'veil-color', 'ring-number', 'ring-type', 'spore-print-color', 'population', 'habitat']
#df5.head()
#print(df5['Class'])
```

1) Breast Cancer Dataset

```
#Covert Concavity from object value to float
df1['Concavity'] = pd.to_numeric(df1['Concavity'], errors='coerce')

# Convert NaN values to zero (optional, depends on your preference)
df1['Concavity'].fillna(0, inplace=True)

# Convert the column to integer type
df1['Concavity'] = df1['Concavity'].astype(int)

# Select Features and Target value
X = df1.iloc[:, :-1] #features
y = df1.iloc[:, -1] #target

y = y.values.reshape(X.shape[0],1)

 # Convert numerical labels to categorical values
y = np.array(y > 2, dtype = 'float')
```

2) Car Dataset

```
# Select Features and Target value
X = df2.iloc[:, :-1] #features
y = df2["Class"] #target

# Step 1: Label encode the categorical labels
label_encoder = LabelEncoder()
integer_encoded = label_encoder.fit_transform(y)
df_label_encoded = X.copy()
for col in X.columns:
```

```
  #print(f"{col}:  {df1[col].dtype}")
  if df2[col].dtype == 'object':  # Only encode categorical columns
    df_label_encoded[col] = label_encoder.fit_transform(df2[col])
X =  df_label_encoded

# Step 2: One-hot encode the integer-encoded labels
onehot_encoder = OneHotEncoder(sparse=False)
integer_encoded = integer_encoded.reshape(len(integer_encoded), 1)
y=integer_encoded
#y = onehot_encoder.fit_transform(integer_encoded)
print(y.shape)
```

3) Ecoli data Set

```
# Select Features and Target value
X = df3.iloc[:, :-1] #features
y = df3.iloc[:, -1] #target
#print(X.shape)
#y=np.squeeze(y)

# Step 1: Label encode the categorical labels
label_encoder = LabelEncoder()
integer_encoded = label_encoder.fit_transform(y)
df_label_encoded = X.copy()
for col in X.columns:
  #print(f"{col}:  {df1[col].dtype}")
  if df3[col].dtype == 'object':  # Only encode categorical columns
    df_label_encoded[col] = label_encoder.fit_transform(df1[col])
X =  df_label_encoded

# Step 2: One-hot encode the integer-encoded labels
onehot_encoder = OneHotEncoder(sparse=False)
integer_encoded = integer_encoded.reshape(len(integer_encoded), 1)
y=integer_encoded
#y = onehot_encoder.fit_transform(integer_encoded)
print(y.shape)
```

4) Letter Recognition Data Set

```
X = df4.drop("Class", axis=1) #features
y = df4['Class'] #target
#print(y.head())
#print(X.head())

# Get unique values in the 'Column_Name' column
unique_values = y.unique()

# Step 1: Label encode the categorical labels
label_encoder = LabelEncoder()
integer_encoded = label_encoder.fit_transform(y)

# Step 2: One-hot encode the integer-encoded labels
onehot_encoder = OneHotEncoder(sparse=False)
integer_encoded = integer_encoded.reshape(len(integer_encoded), 1)
y=integer_encoded
#y = onehot_encoder.fit_transform(integer_encoded)
print(y.shape)
```

5) Mushroom Data

```
X = df5.iloc[:, :-1] #features
y = df5["Class"] #target

#Step 1: Label encode the categorical labels
label_encoder = LabelEncoder()
integer_encoded = label_encoder.fit_transform(y)
y = integer_encoded
df_label_encoded = X.copy()
for col in X.columns:
  #print(f"{col}:  {df1[col].dtype}")
  if df5[col].dtype == 'object':  # Only encode categorical columns
    df_label_encoded[col] = label_encoder.fit_transform(df5[col])
X =  df_label_encoded

#Reshape y value
y = y.reshape(X.shape[0],1)
```

```
#print(y)
# Convert numerical labels to categorical values
#y = np.array(y > 2, dtype = 'float')

print(y.shape)

knn = KNearestNeighbor()

#Intialize K-fold
kf = KFold(n_splits=5, shuffle=True, random_state=0)
#Declare Accuracy List
testAccListManH=[]
testAccListEcl=[]
for i in range(10):
  for train_index, test_index in kf.split(X):
    # Split data into training and test sets for the current fold
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y[train_index], y[test_index]

    X_train_conv = X_train.values
    X_test_conv = X_test.values
    #Train Knn
    knn.train(X_train_conv, y_train)

    #Predict Knn with manhattan Distance
    Yte_predict_manhattan = knn.predictLManhattan(X_test_conv,y_test)
    #Predict Knn with Euclidean Distance
    Yte_predict_euclidean = knn.predictLEuclidean(X_test_conv,y_test)

    #Append accuracy for Manhattan
    testAccListManH.append(Yte_predict_manhattan)
    #Append accuracy for Euclidean
    testAccListEcl.append(Yte_predict_euclidean)

print(testAccListManH)
print(f"Average Accuracy For Manhattan is: {np.round(np.mean(testAccListManH),2)}")
print(f"Standard Deviation for Manhattan Accuracy is:
{np.round(np.std(testAccListManH),4)}")
```

```
print(testAccListEcl)
print(f"Average Accuracy For Euclidean is: {np.round(np.mean(testAccListEcl),2)}")
print(f"Standard Deviation for Euclidean Accuracy is: {np.round(np.std(testAccListEcl),4)}")
```