CS5014

P2: Classification of Seal Images

REPORT

220026989

1. Introduction

This report summarizes the approaches taken for completing the practical - 2 of machine learning module. The goal of this project is to create a classification model that, by selecting the best model from the available options, can accurately predict the outcome of a given collection of data. The classification models were constructed in accordance with the dataset's binary and multi-class data, which was employed in this research. The practical's goal is to categorize Seal images. For the binary class of data, this project's classification models include Logistic Regression, KNN, Decision Trees, and SVC; for the multiclass dataset, they include KNN, Decision Tree, and SVC. The project requires a number of processes, including loading the datasets, preparing the data, choosing the features, and training and assessing the models. The performance of the models is evaluated using evaluation metrics such as accuracy, precision, recall, F1 score, and balanced accuracy.

1.1 Structure

- **ML2.py**
- **PCA_plot**
- **visualPlot**
- **images**
- **requirements.txt**
- **predictions**

- ML2.py
  - The main python script to be run
- PCA_plot
  - A folder that contains the PCA (Principal Component Analysis) plot for the binary and multi classes provided.
- visualPlot

  - A folder that contains a plot of the visual representation of the distribution of data categories for both binary and multi classes provided.
- Images
  - A folder that contains the visual representation (images) of all the models used for both binary and multi classes. The images are categorized into two different folders within named BINARY and MULTI
- requirements.txt
  - All the dependencies to be installed for the python script to run smoothly.
- Predictions
  - Contains the csv files for the predictions of X_test provided for both binary and multi classes separated by folders of the category name.

1.2 Compilation and running instructions

- Install the dependencies
  >> pip install requirements.txt

- Running the script
  >> python ML2.py

- **CAUTION**

  >> A basic path CS5014/{binary/multi}/{dataset.csv} from its source file is a requirement for the program.

2. Loading Data

When the script starts running it loads six different datasets provided for the coursework – an X_train ,X_test, y_train for the binary class and similarly another three sets for the multi class. The pd.read_csv() function is used to read the CSV file and returns a DataFrame object. The function takes several arguments, including the file path of the CSV file, which is passed as a string to the r (raw string) prefix to prevent backslashes in the path from being interpreted as escape characters.
The header=None argument specifies that the CSV file does not have a header row, and therefore, the function should not treat the first row as column names. From the data exploration part of the script we get to know the type and shape of the dataset provided. Note that we only do data exploration for the training data set and not the test data set inorder to avoid data leakage.

3. Exploratory Data analysis and preprocessing

3.1 Checking the null values

The python function called check_null takes in two arguments: the dataframe – df and the name of the dataset – df_name (is only used for clarity). The df argument is expected to be a Pandas DataFrame object and df_name is a string containing the name of the DataFrame

The function first uses the isnull() method of the DataFrame to create a boolean mask identifying all the null values in the DataFrame. The any(axis=0) method is then used to check if there are any null values present along the columns of the DataFrame. This is done from the inference of the given specifications and data exploration, as each column is a data to be classified. The sum() method is applied to the resulting boolean Series to get the total number of columns that have at least one null value, which is the easier way to find whether there are any null values in any of the columns.

3.2 Duplicate values

The function first uses the duplicated() method of the DataFrame to create a boolean mask identifying all the rows that are duplicates of previous rows. The sum() method is then used to count the number of rows that are duplicates. The resulting count is stored in the total_duplicates_rows variable.

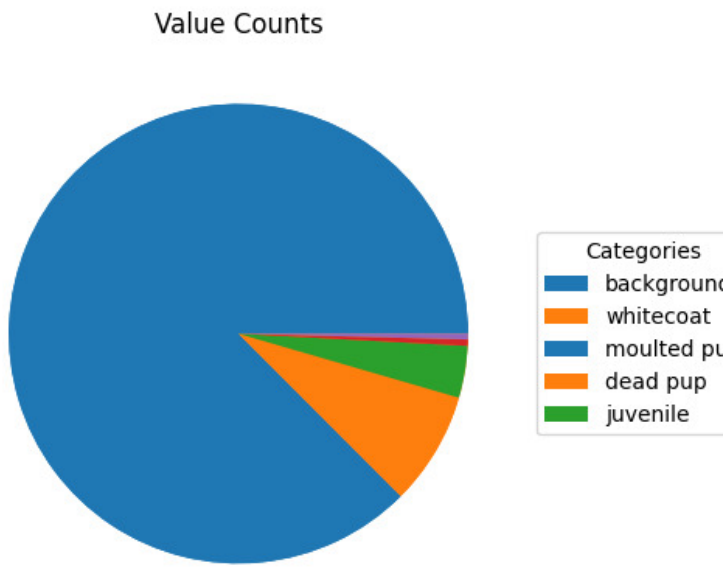
The function then uses the T attribute of the DataFrame to transpose the DataFrame, and then applies the duplicated() method to check for duplicated columns.

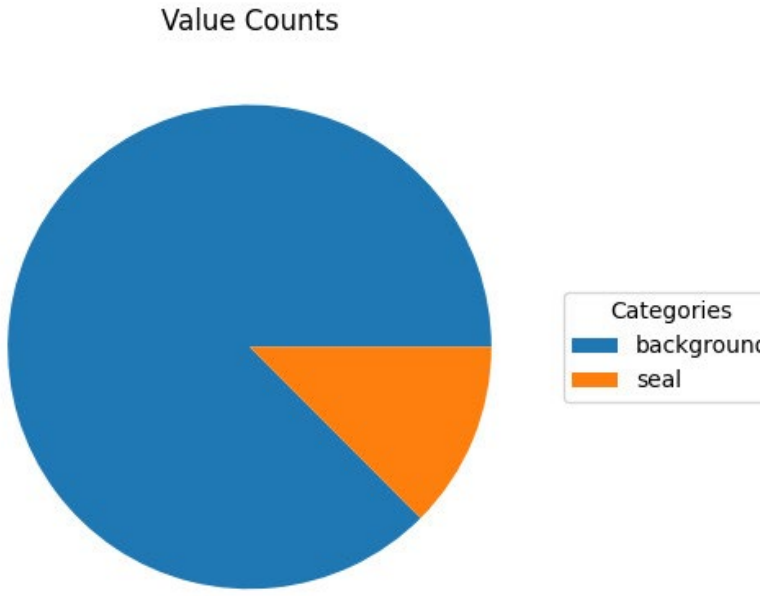
3.3 Visualization

plot_data_Y takes in the Y_train data inorder to visualize the categories of classification in the data. The function first selects the first column of the DataFrame using the columns attribute of the DataFrame and index 0 (only a single column exists). The value_counts() method is then used to count the frequency of each value in the selected column. The resulting counts are stored in the value_counts variable.

The function then uses the pie() method of the matplotlib library to create a pie chart of the value counts. The savefig() method is then used to save the resulting plot in a file named after the DataFrame in a directory named visualPlot.

Fig : Visualization for a) Multi class
b)Binary class





When one class has significantly more samples than the others, the model becomes biased towards the majority class and underfit or completely ignore the minority class. This means that the model may perform well on the majority class but may fail to recognize or predict the minority class accurately, which can be especially problematic in cases where the minority class represents a critical or rare outcome.

Furthermore, since most machine learning algorithms aim to minimize overall error, they may prioritize correctly classifying the majority class at the expense of the minority class. This can lead to poor predictive accuracy, low precision, and high false-negative rates for the minority class.

To address class imbalance, it is often necessary to use techniques such as oversampling the minority class, undersampling the majority class, or using specialized algorithms such as cost-sensitive learning or ensemble methods.
For the same reason we implement class-weighted techniques in most of the hyperparameter optimization giving equal or balanced importance to both the classes inorder to avoid this underfitting of minority classes.

3.4 Train-validation split

The train data loaded is then split into train and validation with the help of train_test_split function. In this case we allocate 20% of the data for validation, and 80% for training. This means that X_train_B and y_train_B will contain 80% of the data, and X_val_B and y_val_B will contain 20% of the data.
stratify=Y_train_binary: ensures that the validation set is created to have the same proportion of classes as the training set. This is important to ensure that the validation set is representative of the overall distribution of classes in the data, which can help to avoid bias in model evaluation.
random_state=50: is used to set the random seed for reproducibility. By setting a specific random seed, we can ensure that the same split is obtained each time the code is run.

3.5 Encoding the target variable data

In this code snippet, the LabelEncoder class from the scikit-learn library is used to transform categorical target variables into numerical labels for both binary and multi-class classification problems.First, the fit_transform() method of the LabelEncoder class is used to fit the encoder to the target data and transform the data into numerical labels. The values.ravel() method is used to convert the target data from a pandas DataFrame to a numpy array before applying the encoder.

Then, the transformed target data is converted back to a pandas DataFrame using the pd.DataFrame() method. This is done so that the transformed data can be easily integrated into the training process of machine learning models.
For multi-class classification problems, the same process is repeated as for binary classification, but the encoder is fitted and applied to the target data with multiple classes.

3.6 Standardising the data

Initially, a StandardScaler object is created, which will be used to standardize the data. Then, the fit() method of the StandardScaler object is called on the training data - X_train_B which calculates the mean and standard deviation of each feature in the training data.After fitting the scaler to the training data, the transform() method of the StandardScaler object is called on the training data X_train_B, testing data X_test_B, and validation data X_val_B to standardize the data based on the mean and standard deviation calculated during the fit() step.

The standardized data is then converted back to pandas DataFrames using the pd.DataFrame() method, and stored in the X_Binary_Train, X_Binary_Test, and X_Binary_Val variables.
The similar is done for the multi class dataset.

3.7 Dimensionality Reduction through Correlation Matrix thresholding

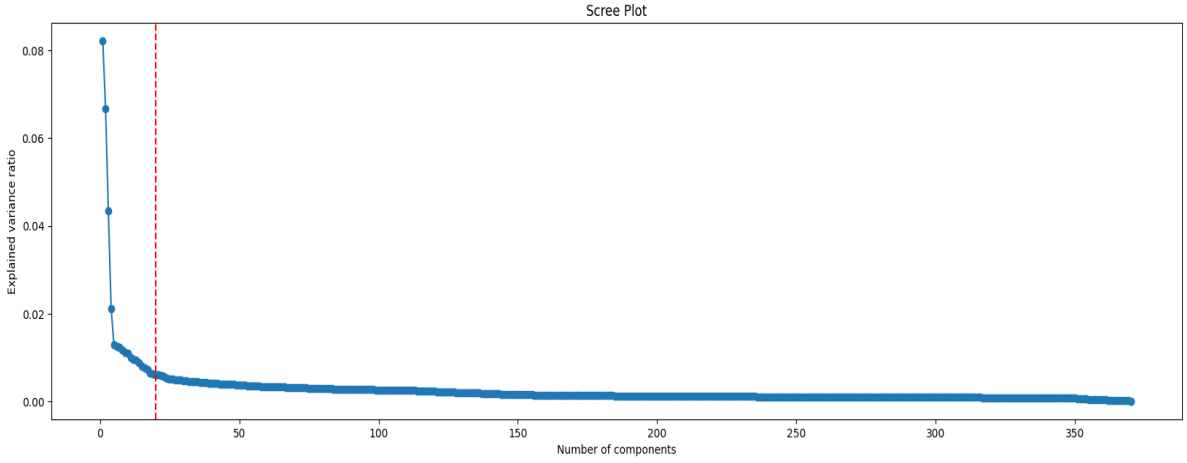
Initially, the correlation matrix is computed using the corr() function applied to the binary training data X_Binary_Train. This correlation matrix is a square matrix where each entry represents the correlation coefficient between a pair of features.
Next, a correlation threshold is defined, which in this case is set to 0.8. This threshold determines the minimum correlation between two features that is considered "high". (I could've decreased the threshold to 0.6 which would rather decrease the features by a large amount. Since we are using PCA for dimensionality reduction again, I took 0.8 to be the threshold).
Using the threshold, we find pairs of columns that have correlations above the threshold using the np.where() function. The resulting pairs of columns are stored in the high_correlations list as tuples of column names.Finally, the code iterates through each pair of columns in high_correlations, and drops it using the drop() function along the column axis (i.e., axis=1). This is done for the training, validation, and testing datasets, which are X_Binary_Train, X_Binary_Val, and X_Binary_Test, respectively.
By dropping columns with high correlation, the dimensionality of the dataset is

reduced, and the remaining columns are less likely to contain redundant information.

3.8 PCA Scree Plot for Dimensionality Reduction

PCA is a technique used for dimensionality reduction, which is a process of reducing the number of input features of a dataset while retaining the most important information in the data. How I understand is, PCA works by creating new features, called principal components, which are linear combinations of the original features. These new features are chosen in such a way that the first principal component captures the most variation in the data, the second component captures the most variation that is not captured by the first component, and so on.
The code first creates a PCA object using PCA(). Then, it fits the PCA object to the training data (first binary and then multi) using the fit() method. The scree plot is then generated. The scree plot shows the explained variance ratio for each principal component. The explained variance ratio indicates the proportion of the total variance in the data that is explained by each principal component. The x-axis of the scree plot shows the number of components, while the y-axis shows the explained variance ratio.

The plt.axvline() function is used to add a vertical line to the plot at x = 20, which can be used as a reference point to identify the "elbow" in the scree plot. The elbow is the point where adding additional components provides only a marginal increase in explained variance. This point is often used as a heuristic to determine the number of components to retain for further analysis.



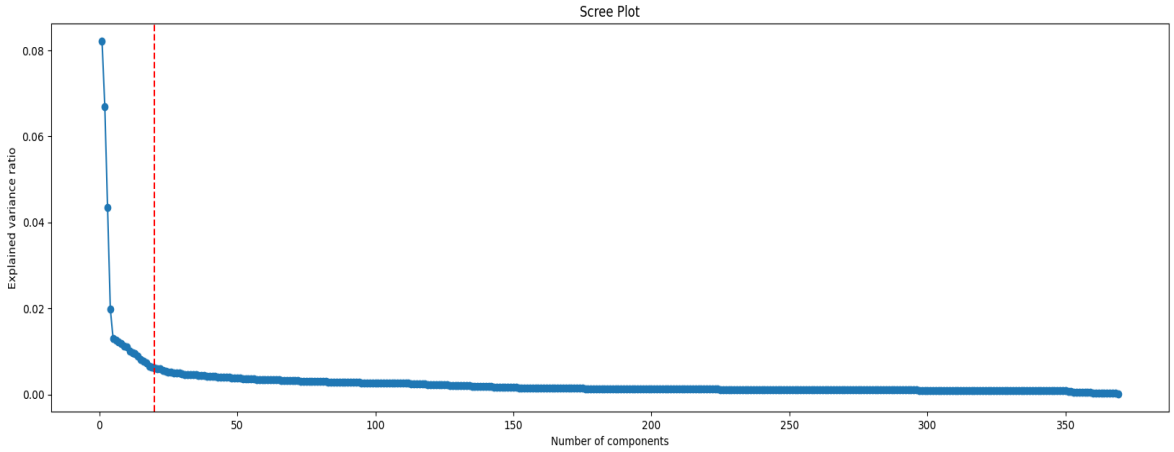


Fig: Scree plot for PCA for a) Binary class.
b) Multi class

4. Data training and validation

4.1 Hyperparameter optimization techniques used:

GridSearchCV:

GridSearchCV is a technique used to search for the optimal hyperparameters of a model by exhaustively trying all possible combinations of hyperparameters from a given search space. It is a brute-force approach where every possible combination of hyperparameters is tried, and the model is trained and evaluated using cross-validation on each combination. The final result is the combination of hyperparameters that produces the best performance on the validation set.

The main advantage of GridSearchCV is that it ensures that the hyperparameters are tuned in an exhaustive and systematic way, making it more likely to find the optimal combination of hyperparameters. However, the downside is that it can be computationally expensive and time-consuming, especially when the search space is large.

RandomizedSearchCV:

RandomizedSearchCV - a hyperparameter tuning method used to find the optimal hyperparameters of a machine learning model. It is an improvement over GridSearchCV because it searches the hyperparameters randomly and does

not evaluate every possible combination of hyperparameters, which can be computationally expensive and time-consuming.
The main advantage of RandomizedSearchCV is that it saves computational resources and time by sampling only a subset of the hyperparameter space. This is particularly useful when the hyperparameter space is large and there are many possible combinations of hyperparameters. RandomizedSearchCV also allows for a more thorough exploration of the hyperparameter space than manual tuning, which can lead to improved model performance. RandomizedSearchCV also provides the ability to specify the number of iterations or search points, which can be adjusted based on the available computational resources and the size of the hyperparameter space.

4.2 Binary class category

RandomizedSearchCV applied:

```
#     random_search = RandomizedSearchCV(
#         estimator=model,
#         param_distributions=param_dist,
#         n_iter=100,
#         cv=5,
#         random_state=42
#     )
```

- estimator: This is the estimator object that implements the algorithm that you want to tune the hyperparameters for. In this case, model is used as the estimator object.

- param_distributions: This is a dictionary containing the parameters to be tuned for the estimator. The values for each parameter can be specified as a list of possible values or as a distribution. In this example, param_dist is used as the parameter dictionary.

- n_iter: This parameter controls the number of parameter settings that are sampled. A higher value for n_iter will increase the chances of finding the best combination of hyperparameters. In this case, n_iter is set to 100.

- cv: This parameter determines the cross-validation splitting strategy. In this example, cv is set to 5, meaning that a 5-fold cross-validation will be used.

- random_state: This parameter sets the seed for the random number generator, which is used to sample the parameter combinations. Setting this value will ensure that the results are reproducible. In this example, random_state is set to 42.

4.2.1 Logistic regression

The logistic_param_dist dictionary defines a set of hyperparameters to be used with logistic regression in a hyperparameter tuning algorithm. Here we use GridSearchCV as there are only few hyperparameters. Since the hyperparameters are already defined with a minimum iteration, it crashes the program if there are less iterations.

The hyperparameters and their values are:

- penalty: Regularization penalty, either L2 (ridge) or None. L2 regularization adds a penalty term to the loss function based on the sum of squared weights, which can help to reduce overfitting. The None option means no regularization is applied.
- class_weight: The way to handle class imbalance. Either None for equal class weight or balanced for automatic adjustment of weights based on the frequency of each class can be given. Since we know from the analysis that we are dealing with a highly imbalanced data, we only go for 'balanced' case.
- max_iter: Maximum number of iterations for the solver to converge, where the solver is the optimization algorithm used to minimize the loss function.

These hyperparameters have been chosen based on their relevance to logistic regression and their impact on the model's performance. Penalty and class_weight are two important hyperparameters that can help to improve the performance of the model, especially in imbalanced datasets. max_iter specifies the maximum number of iterations before the solver converges, which affects the convergence of the algorithm and the resulting model performance.

4.2.2 K Nearest Neighbours

- n_neighbors: This parameter determines the number of nearest neighbors to include in the majority voting process for making

predictions. The value range is selected from 10 to 20 using the randint function. A value of 10-20 is not too small and not too large, providing a good balance between bias and variance.

- weights: This parameter specifies the weight function to use in prediction. Here, the value is set to 'distance', which means that closer neighbors will have a greater influence on the prediction than farther neighbors. This again is done due to an imbalanced dataset. The other values are not provided.

- metric: This parameter specifies the distance metric used to compute the distance between points. Here, two metrics are chosen, namely 'euclidean' and 'manhattan'. The Euclidean distance is used when the features have continuous values and the Manhattan distance is used when the features have categorical or ordinal values.

4.2.3 Decision Tree

- max_depth: This parameter specifies the maximum depth of the decision tree. The values range from 2 to 10 and also include None. None indicates that the nodes are expanded until all the leaves contain less than min_samples_split samples or all the leaves are pure. The choice of values is made to prevent overfitting by controlling the depth of the tree.

- min_samples_split: This parameter specifies the minimum number of samples required to split a node. The values range from 2 to 10 using the randint function. A smaller value can lead to overfitting, and a larger value can lead to underfitting. The choice of values provides a good balance between bias and variance.

- min_samples_leaf: This parameter specifies the minimum number of samples required to be at a leaf node. The values range from 1 to 10 using the randint function. A smaller value can lead to overfitting, and a larger value can lead to underfitting. The choice of values provides a good balance between bias and variance.

- max_features: This parameter specifies the number of features to consider when looking for the best split. Three values are chosen, namely 'sqrt', 'log2', and None. 'sqrt' and 'log2' indicate the square

root and logarithm of the total number of features, respectively. None indicates that all the features will be considered.

- class_weight: This parameter specifies the weight of each class in the model. The value 'balanced' indicates that the weights will be inversely proportional to the class frequencies. This can help in handling imbalanced datasets.

- criterion: This parameter specifies the function to measure the quality of a split. Two values are chosen, namely 'gini' and 'entropy'. 'gini' measures the impurity of a node, while 'entropy' measures the information gain of a node.

4.3 Multi class category

RandomizedSearchCV applied:

Here we add a few couple of parameters:

- n_jobs=-1: This parameter specifies the number of CPU cores to use for parallelizing the search process. Setting it to -1 means that all available cores will be used. This can significantly speed up the search process when dealing with large datasets and complex models. The parameter can also take a positive integer value to specify the exact number of cores to use.

- scoring='accuracy': This parameter specifies the scoring metric used to evaluate the performance of the model. In this case, the 'accuracy' metric is used, which measures the proportion of correctly classified instances.

When n_jobs is not used, the search process runs on a single CPU core, which can be slower for large datasets and complex models. On the other hand, when scoring is not used, the default scoring metric for the given model is used, which may not be suitable for the specific problem being solved.

In the multi class section we use KNN,decision tree models as seen before. Here we also introduce another model:

- Random Forest : Combines multiple decision trees to reduce overfitting and improve generalization. Also, effective in handling imbalanced datasets as it randomly samples the data and creates decision trees on different subsets of data, which helps to balance class distribution

Hyperparameters:

- n_estimators: This hyperparameter controls the number of decision trees in the random forest ensemble. The randint(50, 200) expression means that the number of trees will be randomly chosen from the integer range [50, 200] during the training process.

- max_depth: This hyperparameter controls the maximum depth of each decision tree in the random forest ensemble. The randint(2, 20) expression means that the maximum depth of each tree will be randomly chosen from the integer range [2, 20] during the training process.

- min_samples_split: This hyperparameter controls the minimum number of samples required to split an internal node in each decision tree. The randint(2, 20) expression means that the minimum number of samples required to split a node will be randomly chosen from the integer range [2, 20] during the training process.

- min_samples_leaf: This hyperparameter controls the minimum number of samples required to be in a leaf node in each decision tree. The randint(1, 10) expression means that the minimum number of samples required to be in a leaf node will be randomly chosen from the integer range [1, 10] during the training process.

- max_features: This hyperparameter controls the maximum number of features that can be used in each decision tree. The ['sqrt', 'log2'] expression means that the maximum number of features will be either the square root or the base-2 logarithm of the total number of features in the dataset.

- class_weight: This hyperparameter controls the weight assigned to each class in the training process to handle imbalanced data. The ['balanced', 'balanced_subsample'] expression means that the class weights will be either balanced or balanced subsample. 'balanced' option will set the class weights inversely proportional to their frequency in the training set, and 'balanced_subsample' option will set the class weights in the same way as 'balanced', but it will also subsample the training set for each tree to balance the class distribution in each subset.
  During the training process, the values of these hyperparameters will be randomly sampled from their respective ranges or options. This random sampling helps to explore different combinations of hyperparameters and

find the optimal combination that gives the best performance on the validation set.

5. Analysis:

Model Used - Class Category	Accuracy	Precision	Balanced Accuracy	Recall	F1_score
Logistic Regression – Binary	0.78	.85	.38	0.78	0.81
KNN, Binary	0.87	0.83	0.04	0.87	0.83
Decision Tree, Binary	0.85	0.84	0.26	0.85	0.84
KNN, Multi	0.87	0.83	0.015	0.88	0.83
DT, Multi	0.83	0.82	0.08	0.83	0.82
RF, Multi	0.88	0.84	0.06	0.88	0.85

Table 1: Performance table for all the models.

Table 1 shows the metrics (accuracy, Precision, Balanced Accuracy, Recall, F1_Score) for all the algorithms applied to both the binary and multi-class classification task. It can be seen from the table that for binary classification task, KNN performed better amongst the three algorithms implemented getting the highest accuracy of 87%.Logistic regression performance was worst among the three having only 78% accuracy. On the other hand, for the multi-class classification Random Forest algorithm performed better comparing to the other algorithms scoring an accuracy of 88%. In the case of multi-class classification all the algorithms performed almost similarly with only slight variance in percentages.

6. Conclusion

In conclusion, the goal of this study was to create classification models for datasets with binary and multiple classes. Logistic regression, K-Nearest Neighbours, Decision Trees, and Random Forest were only a few of the machine learning algorithms that were used to train and assess the categorization models. The metrics of each algorithm for specific category of classes were recorded. The details of the process has been captured and stored in various folders as images has been submitted. Finally, the metrics of each model was compared with respect to the class category.