MACHINE LEARNING AND NEURAL NETWORKS

MACHINE LEARNING CASE STUDY

USING LOGISTIC REGRESSION, RANDOM FOREST CLASSIFIER AND DECISION TREE CLASSIFIER

Google Colab Notebook Link:
https://colab.research.google.com/drive/11PQZUbRgNSWnXt2xF-n9D-QmctTXid-3#scrollTo=AlG26Hn5QWRr
Data Link: https://archive.ics.uci.edu/static/public/73/data.csv

Student ID: 22073997

INTRODUCTION

The UCI Machine Learning Repository provided the Mushroom dataset, which was employed in this investigation. The Audubon Society Field Guide donated this dataset on April 26, 1987, and it provides descriptions of the many physical attributes of mushrooms. This dataset's main goal is to categorise mushrooms as edible or poisonous according to their characteristics.

The dataset is widely utilised in machine learning and data science applications for classification tasks, and it is especially important for comprehending the edible nature of mushrooms. The dataset is an invaluable tool for investigating pattern recognition algorithms and developing predictive models to differentiate between edible and poisonous mushrooms due to its extensive collection of mushroom properties and their accompanying classifications.

PREPROCESSING TECHNIQUES

- An essential step in getting the dataset ready for machine learning models is preprocessing.
- It entails scaling the data, encoding categorical variables, and addressing missing values.

Handling Missing Values:

- Use the isnull() function to find missing values.
- Use the drop() function to remove columns with a high number of missing values.

```
# Drop the column with large null values
df.drop(columns=['stalk-root'], inplace=True)
```

Encoding Categorical Variables:

- Utilised label encoding to convert categorical variables into numerical format.
- Numerical values can be obtained by converting category values using the Scikit-learn LabelEncoder.

```
from sklearn.preprocessing import LabelEncoder

# Create a copy of your DataFrame to preserve the original data
df = df.copy()

# Initialize a LabelEncoder object
label_encoder = LabelEncoder()

# Iterate over each column in your DataFrame
for column in df.columns:
    # Check if the column data type is object (categorical)
    if df[column].dtype == 'object':
        # Use LabelEncoder to encode the categorical values into numerical values
        df[column] = label_encoder.fit_transform(df[column])

# Now, encoded_df contains numerical values for categorical columns
print(df.head())
```

Data spliting:

To assess how well machine learning models performed, the dataset was divided into training and testing sets. To make sure that the model's performance can be extrapolated to untested data, this step is essential. The procedure that was followed was as follows:

- The necessary libraries were imported, including train_test_split from Scikit-learn and StandardScaler.
- To divide the dataset into training and testing sets, the train_test_split function was utilised.
- With the test_size option set to 0.3, 30% of the data would be kept for testing and the remaining 70% for training.
- In order to guarantee the preservation of the class distribution in the training and testing sets, the stratify parameter was assigned to the target variable y. For unbalanced datasets, this is especially crucial to avoid biassed model performance.

Scaling the Features:

- With StandardScaler, the features in the training and testing sets were scaled.
- For models that are sensitive to feature sizes, such as logistic regression and support vector machines, scaling guarantees that all features have the same scale.
- For the training set, the fit_transform method was used, and for the testing set, the transform method. By doing this, data leakage is avoided by ensuring that the scaling parameters acquired from the training set are consistently applied to the testing set.

```
# Create a train and test dataset
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

# split the dataset with test size = 0.3

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state = 45, stratify = y)
#Scale the dependent variable
scale= StandardScaler()
X_train = scale.fit_transform(X_train)
X_test = scale.transform(X_test)
```

Benefits:

- Making sure the data is in an appropriate format for training machine learning models is the goal of preprocessing.
- By managing inconsistent data and getting it ready for analysis, it helps to increase the models' accuracy and performance.

ML ARCHITECTURE AND PARAMETERS

Model Selection:
Three different classification algorithms were experimented

- Logistic Regression
- Random Forest Classifier
- Decision Tree Classifier

Logistic Regression:
Parameters: To control overfitting, we employed a regularisation parameter (C). To get the optimal C value, We performed hyperparameter tuning using cross-validation.
Hyperparameter tuning: To get the optimal C value, GridSearchCV with 5-fold cross-validation was used.
Random Forest Classifier:
Three hyperparameters were fine-tuned: max_depth, min_samples_leaf, and n_estimators.
Hyperparameter tuning: To find the ideal set of hyperparameters, GridSearchCV with 3-fold cross-validation was employed.

```
from sklearn.model_selection import KFold, GridSearchCV
from sklearn.linear_model import LogisticRegression

lr = LogisticRegression()
parameter = {'C': [0.001, 0.01, 0.1, 1.0, 10.0]}
#Kfold for evaluating the performance of my model
#gridsearch technique used for hyperparameter tuning
kf = KFold(n_splits=5, shuffle=True, random_state=30)
grid_cv = GridSearchCV(lr, parameter, cv=kf)
grid_cv.fit(X_train, y_train.values.ravel())  # Convert y_train DataFrame to a 1D array using values.ravel()

print('The Best Parameter: {}'.format(grid_cv.best_params_))
print('The Best Score: {}'.format(grid_cv.best_score_))
```

```
The Best Parameter: {'C': 1.0}
The Best Score: 0.9511083494473322
```

```
# Try to get the best hyperparameter for random forest classifier
# import the random forest classifier
from sklearn.ensemble import RandomForestClassifier
rf = RandomForestClassifier ()
parameter ={'max_depth':[3,4,5],'min_samples_leaf':[0.1,0.2,0.3],'n_estimators':[300, 400, 450]}
grid_rf = GridSearchCV(rf, param_grid =parameter, cv=3, n_jobs=-1)
grid_rf.fit(X_train,y_train_1d)
print('Best score: {}'.format (grid_rf.best_score_))
print ('Best params: {}'.format (grid_rf.best_params_))
print ('Best model: {}'. format(grid_rf.best_estimator_))
rf_model = grid_rf.best_estimator_
```

```
Best score: 0.9198032055635341
Best params: {'max_depth': 4, 'min_samples_leaf': 0.1, 'n_estimators': 400}
Best model: RandomForestClassifier(max_depth=4, min_samples_leaf=0.1, n_estimators=400)
```

Decision Tree Classifier:
Parameters: to control the maximum depth of the decision tree, i adjusted the max_depth parameter
Hyperparameter Tuning: i used the default criterion (gini) and (entropy) to compare the performance of the decision tree classifiers.

```
from sklearn.tree import DecisionTreeClassifier
from sklearn import tree
#Decision Tree Classifier with splitting criterion as Gini impurity, the maximum depth of the tree is 3.
dtc_gini = DecisionTreeClassifier(criterion='gini', max_depth=3, random_state=0)


# fit the model
dtc_gini.fit(X_train, y_train)
```

```
DecisionTreeClassifier
DecisionTreeClassifier(max_depth=3, random_state=0)
```

```
#Decision Tree Classifier with splitting criterion as Entropy impurity, the maximum depth of the tree is 3.
dtc_en = DecisionTreeClassifier(criterion='entropy', max_depth=3, random_state=0)


# fit the model
dtc_en.fit(X_train, y_train)
```

```
DecisionTreeClassifier
DecisionTreeClassifier(criterion='entropy', max_depth=3, random_state=0)
```

RESULTS AND CRITICAL EVALUATION OF MY MODEL

Features	Logistic Regression	Random Forest Classifier	Decision Tree using gini impurity	Decision Tree using entropy impurity
Accuracy	93.7%	90.7%	94.8%	94.0%
AUC Score	93.67%	90.48%	94.87%	94.1%
Precision	93% for edible class 95% for poisonous class	86% for edible class 96% for poisonous class	97% for edible class 92% for poisonous class	98% for edible class 91% for poisonous class
Recall	95% for edible class 92% for poisonous class	96% for edible class 84% for poisonous class	93% for edible class 97% for poisonous class	90% for edible class 98% for poisonous class
F1-score	94% for edible class 93% for poisonous class	91% for edible class 89% for poisonous class	95% for both class	94% for both class
Training set score	94%	89%	95%	94%

CRITICAL EVALUATION

While all models successfully classified mushrooms as edible or poisonous, the logistic regression and decision tree models outperformed the random forest classifier by a small margin.

Decision tree models performed quite well in terms of generalisation, with very little overfitting seen.

According to the AUC scores, all models did a good job of differentiating between the two classes; the decision tree that used the Gini impurity had the greatest AUC score.

A balanced performance in properly identifying both edible and poisonous mushrooms is demonstrated by precision, recall, and F1-scores.

Based on accuracy and AUC score, the best models are logistic regression and decision tree models using the Gini impurity.

LIMITATIONS AND POTENTIAL AREAS OF IMPROVEMENT

The study analyzed the Mushroom dataset using logistic regression, random forest classifier, and decision tree models to predict mushroom edibility. Despite high accuracy rates, the project has limitations such as limited feature set, potential class imbalance, data quality issues, and ethical considerations. Further research is needed to develop more robust models, including domain knowledge, alternative algorithms, and data quality. This is crucial for advancing mushroom classification and user safety.

Data quality: Like any real-world dataset, the Mushroom dataset could have mistakes, inconsistencies, or missing variables. These problems with the quality of the data can impair machine learning models' effectiveness and result in incorrect predictions.

Unbalanced Courses:

There could be an imbalance in the dataset between the classes, with edible mushrooms, for example, being more common than the other. This may result in the minority class being incorrectly classified and biassed model performance.

Engineering Features:

To better capture the properties of mushrooms, new features might be created or more significant information could be extracted from the current features in the dataset through feature engineering.

Hyperparameter tuning: Machine learning models may perform better if their hyperparameters are further optimised. To systematically look for the ideal set of hyperparameters, methods such as grid search or random search can be used.

CONCLUSION

I effectively investigated the use of machine learning approaches for the categorization of the edibility of mushrooms. By employing logistic regression, random forest classifier, and decision tree models, i was able to predict mushroom classes with a high degree of accuracy according to their physical characteristics.

The study did, however, also draw attention to certain drawbacks, such as the feature set of the dataset and possible class imbalance, in addition to difficulties with model interpretability and generalisation to real-world situations. Notwithstanding these drawbacks, our results highlight the promise of machine learning for classifying mushrooms and emphasise the need for more study to overcome these obstacles.

In order to progress the field of mushroom classification and guarantee the security of users depending on such models, it will be essential to make efforts to improve model robustness, address problems with data quality, and guarantee ethical concerns. All things considered, our effort lays the foundation for future developments in the field of mushroom classification by providing an insightful investigation into the use of machine learning in this area.