Universal Code Writer for Machine Learning

Course: Artificial Intelligence & Machine Learning (8th Semester)

Supervisor: Dr. Saiaf Bin Rayhan, Assistant Professor, Department of Aerospace Engineering, Aviation and Aerospace University, Bangladesh (AAUB)

Contributor: Md. Hemal Ishak, Teaching Assistant | Undergrad Student, Department of Aerospace Engineering

Overview: A Python-based unified code writer developed as a comprehensive framework for building, training, and evaluating machine learning models. It streamlines the entire workflow- from data preprocessing to model evaluation, enabling students to experiment efficiently across tasks.

ML Algorithm Code Writer

Scope

- Designed as an educational template for beginners to understand the workflow of classical machine learning models.
- Demonstrates data splitting, model training, prediction, and evaluation in a simplified manner.
- Applicable to most classical ML algorithms (e.g., Linear Regression, Logistic Regression, Decision Trees, Random Forest, SVM, KNN).
- Includes **basic hyperparameter tuning** examples (e.g., GridSearchCV, RandomizedSearchCV, or Optuna) for model improvement.

Limitations

- Tree visualization is not included in the current version. But for ensemble learning this can be a good tool to adopt.
- Restricted to classical machine learning algorithms only.
- Deep learning and neural network models are not supported.

Step 1. Importing the libraries

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, OneHotEncoder, StandardScaler
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix # For classification task
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error, mean_absolute_percentage_error # For regression task
```

For classification

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.naive_bayes import BernoulliNB
from sklearn.naive_bayes import MultinomialNB
from sklearn.linear_model import LogisticRegression
```

```
import xgboost as xgb
import lightgbm as lgb
from catboost import CatBoostClassifier
from sklearn.svm import SVC
from sklearn.ensemble import AdaBoostClassifier
from sklearn.linear_model import RidgeClassifier
from sklearn.neighbors import RadiusNeighborsClassifier
from sklearn.ensemble import GradientBoostingClassifier
```

Cautions:

- You should know how the algorithm works.
- You should know the hyperparameters that control overfitting and underfitting.

For regression

```
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Ridge
from sklearn.linear_model import Lasso
from sklearn.linear_model import ElasticNet
from sklearn.linear_model import SGDRegressor
```

```
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.ensemble import AdaBoostRegressor
from sklearn.ensemble import ExtraTreesRegressor
```

```
from sklearn.svm import SVR
from sklearn.neighbors import KNeighborsRegressor
from sklearn.neighbors import RadiusNeighborsRegressor
from sklearn.linear_model import BayesianRidge
from sklearn.gaussian_process import GaussianProcessRegressor
import xgboost as xgb
import lightgbm as lgb
from catboost import CatBoostRegressor
```

Cautions:

- You should know how the algorithm works.
- You should know the hyperparameters that control overfitting and underfitting.

Step 2. Reading the files

```
# Read a CSV file
df = pd.read_csv(r"Paste the copied path from the file")

# Read an Excel file
df = pd.read_excel(r"Paste the copied path from the file")

# 'df' is an arbitrary variable name for the dataframe; you can choose any name you like.

df.head(2)
# This shows the first 2 rows, not columns.
# You can use this to check your data, identify categorical features, input variables, and target variables.
```

Step 3. Identify input variables (input features) and output variables (output features/target variables)

```
output = df.pop("target_column_name")

# The pop() command removes the target column from the dataframe
# and assigns it to 'output'.
# Now, 'df' contains only the input features, and 'output' is the target variable.

input = df
# Remaining columns are input features after pop() command.
```

Step 4. Converting Categorical Data

```
input_x= pd.get_dummies(input, drop_first=True)

# Numerical columns are left unchanged.

# Categorical columns are converted into dummy variables.

# drop_first=True avoids the dummy variable trap (removes one category per column)


label_encoder = LabelEncoder()

output_encoded = label_encoder.fit_transform(output)
```

Step 5. Converting dataset to StandardScaler*

```
numerical_features = ['feature_1', 'feature_2', 'feature_3']

scaler = StandardScaler()

input_x[numerical_features] = scaler.fit_transform(input_x[numerical_features])

# By separating numerical_features we are not disturbing the hot encoded values. This is very important not to covert them.

# Only the numerical features will be converted between 0 and 1.
```

Step 6. Split the dataset into Test Data and Train Data

```
x_train, x_test, y_train, y_test = train_test_split(input_x, output_y, test_size = 0.2, random_state=42)
```

test_size = 0.2 suggests that the algorithm will test 20% data which are unseen and train 80% data from where it will try to recognize the pattern.

Step 7. Train the model with the default hyperparameters

Classifiers

```
model = DecisionTreeClassifier()
model.fit(x_train, y_train)

model = RandomForestClassifier()
model.fit(x_train, y_train)

model = ExtraTreesClassifier()
model.fit(x_train, y_train)

model = KNeighborsClassifier()
model.fit(x_train, y_train)

model = GaussianNB()
model.fit(x_train, y_train)

model = BernoulliNB()
model.fit(x_train, y_train)

model = MultinomialNB()
model.fit(x_train, y_train)

model = LogisticRegression(max_iter=1000)
model.fit(x_train, y_train)

model = xgb.XGBClassifier()
model.fit(x_train, y_train)

model = lgb.LGBMClassifier()
model.fit(x_train, y_train)

model = CatBoostClassifier(verbose=0)
model.fit(x_train, y_train)

model = SVC()
model.fit(x_train, y_train)

model = AdaBoostClassifier()
model.fit(x_train, y_train)

model = RidgeClassifier()
model.fit(x_train, y_train)

model = RadiusNeighborsClassifier()
model.fit(x_train, y_train)

model = GradientBoostingClassifier()
model.fit(x_train, y_train)
```

Regressors

```
model = LinearRegression()
model.fit(x_train, y_train)

model = Ridge()
model.fit(x_train, y_train)

model = Lasso()
model.fit(x_train, y_train)

model = ElasticNet()
model.fit(x_train, y_train)

model = SGDRegressor()
model.fit(x_train, y_train)

model = DecisionTreeRegressor()
model.fit(x_train, y_train)

model = RandomForestRegressor()
model.fit(x_train, y_train)

model = GradientBoostingRegressor()
model.fit(x_train, y_train)
```

```
model = AdaBoostRegressor()
model.fit(x_train, y_train)

model = ExtraTreesRegressor()
model.fit(x_train, y_train)

model = SVR()
model.fit(x_train, y_train)

model = KNeighborsRegressor()
model.fit(x_train, y_train)

model = RadiusNeighborsRegressor()
model.fit(x_train, y_train)

model = BayesianRidge()
model.fit(x_train, y_train)
```

```
model = GaussianProcessRegressor()
model.fit(x_train, y_train)

model = xgb.XGBRegressor()
model.fit(x_train, y_train)

model = lgb.LGBMRegressor()
model.fit(x_train, y_train)

model = CatBoostRegressor()
model.fit(x_train, y_train)
```

Classifiers

Step 7. Checking the Metrics

```
y_train_pred = model.predict(x_train)
y_test_pred = model.predict(x_test)

# For checking the performance for both training data and test data.

# Training metrics
print("Training Metrics:")
print("Accuracy:", accuracy_score(y_train, y_train_pred))
print("Classification Report (Train):\n", classification_report(y_train,y_train_pred))
print("Confusion Matrix:\n", confusion_matrix(y_train, y_train_pred))

# Testing metrics
print("Testing Metrics:")
print("Accuracy:", accuracy_score(y_test, y_test_pred))
print("Classfication Report (Test):\n", classification_report(y_test, y_test_pred))
print("Confusion Matrix:\n", confusion_matrix(y_test, y_test_pred))
```

Regressors

```
y_train_pred = model.predict(x_train)
y_test_pred = model.predict(x_test)

# For checking the performance for both training data and test data.

# Training metrics
print("Training Metrics:")
print("MAE:", mean_absolute_error(y_train, y_train_pred))
print("MSE:", mean_squared_error(y_train, y_train_pred))
print("RMSE:", np.sqrt(mean_squared_error(y_train, y_train_pred)))
print("R²:", r2_score(y_train, y_train_pred))

# Testing metrics
print("Testing Metrics:")
print("MAE:", mean_absolute_error(y_test, y_test_pred))
print("MSE:", mean_squared_error(y_test, y_test_pred))
print("RMSE:", root_mean_squared_error(y_test, y_test_pred))
print("R²:", r2_score(y_test, y_test_pred))
```

```
import matplotlib.pyplot as plt
plt.figure(figsize=(7,7))
plt.scatter(y_test, y_test_pred, color='blue', label='Predicted vs Actual')
plt.plot([y_test.min(), y_test.max()],
         [y_test.min(), y_test.max()],
         color='red', linewidth=2, label='Perfect Prediction')
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.title('Actual vs Predicted Values')
plt.legend()
plt.grid(True)
plt.show()
# Provides the curve for Actual Vs Predicted Values

comparison = pd.DataFrame({
   "Actual": y_test,
   "Predicted": y_test_pred
})

comparison["Error"] = np.abs(comparison["Actual"] - comparison["Predicted"])
comparison_sorted = comparison.sort_values(by="Error", ascending=False)

print("Top 20 Largest Errors:")
print(comparison_sorted.head(20))
# Provides the actual list of errors
```

Step 8. Hyperparameter Tuning

Model training with hyperparameters

- Example based on Decision Tree model
- For other model the hyperparameters will be changed
- The shaded box will be changed

Grid Search CV

```
# Import GridSearchCV library
from sklearn.model_selection import GridSearchCV

# Make a dictionary with considerable hyperparameters
param_grid= {
   'max_depth': [2, 3, 5, 10, 20],
   'min_samples_leaf': [5, 10, 20, 50, 100],
   'criterion': ["gini", "entropy"]
}

grid_search= GridSearchCV(
   estimator=model,
   param_grid=param_grid,
   scoring="accuracy",
   cv=5,
   n_jobs=-1
)
grid_search.fit(x_train, y_train)

# print best hyperparameters and CV score
print("Best parameters are: ", grid_search.best_params_)
print("Best Cross validation score is: ", grid_search.best_score_)
```

Randomized Search CV

```
# Import RandomizedSearchCV library
from sklearn.model_selection import RandomizedSearchCV

# make a dictionary with considerable hyperparameters
from scipy.stats import randint

param_dist = {
   'max_depth': randint(1, 20),
   'min_samples_split': randint(2, 20),
   'min_samples_leaf': randint(1, 20),
   "criterion": ["gini", "entropy"],
}

# Apply RandomSearchCV
random_search = RandomizedSearchCV(
   estimator=model,
   param_distributions=param_dist,
   n_iter=50,
   scoring="accuracy",
   cv=5,
   n_jobs=-1,
   random_state=42,
)
random_search.fit(x_train, y_train)

# print best hyperparameters and CV score
print("Best Parameters:", random_search.best_params_)
print("Best CV Score:", random_search.best_score_)
```

Step 8. Hyperparameter Tuning

Bayesian Optimization

Model training with hyperparameters

```
# Defining optuna
def objective(trial):
    max_depth = trial.suggest_int("max_depth", 1, 10)
    min_samples_split = trial.suggest_int("min_samples_split", 2, 10)
    min_samples_leaf = trial.suggest_int("min_samples_leaf", 1, 5)

    model = DecisionTreeClassifier(
        max_depth=max_depth,
        min_samples_split=min_samples_split,
        min_samples_leaf=min_samples_leaf,
        random_state=42
    )
    model.fit(x_train, y_train)
    y_pred = model.predict(x_test)
    accuracy = accuracy_score(y_test, y_pred)
    return accuracy

# Training with optuna

study=optuna.create_study(direction="maximize")
study.optimize(objective,n_trials=20, show_progress_bar=False)
print("Best Hyperparameters:", study.best_params)
optuna_model=DecisionTreeClassifier(**study.best_params,random_state=42)
optuna_model.fit(x_train, y_train)

y_train_pred_HT=optuna_model.predict(x_train)
y_test_pred_HT=optuna_model.predict(x_test)
```

```
# plotting tree after hyperparameter tuning

plt.figure(figsize=(16, 10))
plot_tree(optuna_model, filled=True,
feature_names=x_train.columns, class_names=True)
plt.title("Decision Tree of Bayesian optimization Tunned
Model", fontsize= 18, fontweight='bold')
plt.show()
```

Step 8. Hyperparameter Tuning

Print the metrics and visualization

Classifiers

Grid Search CV

```
# Taking best estimator
best_model = grid_search.best_estimator_
```

Randomized Search CV

```
# Taking best estimator
best_model = random_search.best_estimator_
```

```
# Train and Test Accuracy scores
y_train_pred_HT=best_model.predict(x_train)
y_test_pred_HT=best_model.predict(x_test)
print("Train Accuracy:",accuracy_score(y_train, y_train_pred_HT))
print("Test Accuracy:",accuracy_score(y_test, y_test_pred_HT))

# Classification report
print("Classification report: Hyperparameter tunned model")
print(classification_report(y_test, y_test_pred_HT))

 # Confusion matrix
cm = confusion_matrix(y_test, y_test_pred_HT)
plt.figure(figsize=(6,5))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.title("Confusion Matrix")
plt.show()
```

```
# plotting tree after hyperparameter tuning
# For only Grid Search and Randomized Search CV

from sklearn.tree import plot_tree

plt.figure(figsize=(16, 10))
plot_tree(best_model, filled=True,
feature_names=x_train.columns, class_names=[str(c)
for c in y_train.unique()])
plt.title("Decision Tree of Hyperparameter Tunned
Model", fontsize= 18, fontweight='bold')
plt.show()
```

Step 8. Hyperparameter Tuning

Print the metrics and visualization

Regressors

Grid Search CV

```
# Taking best estimator
best_model = grid_search.best_estimator_
```

Randomized Search CV

```
# Taking best estimator
best_model = random_search.best_estimator_
```

```
# Train and Test Accuracy scores
y_train_pred_HT=best_model.predict(x_train)
y_test_pred_HT=best_model.predict(x_test)

# Train and Test R² scores
print("Train R² Score:", r2_score(y_train, y_train_pred_HT))
print("Test R² Score :", r2_score(y_test, y_test_pred_HT))

# Compute other regression metrics for test set
mse = mean_squared_error(y_test, y_test_pred_HT)
rmse = np.sqrt(mse)
mae = mean_absolute_error(y_test, y_test_pred_HT)

print("\nTest Set Evaluation Metrics:")
print("MSE :", mse)
print("RMSE:", rmse)
print("MAE :", mae)
```

```
# Actual vs Predicted Plot

plt.scatter(y_test, y_test_pred_HT, alpha=0.7)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--')
plt.xlabel("Actual")
plt.ylabel("Predicted")
plt.title("Actual vs Predicted")
plt.show()
```

Step 9. K-Fold Cross Validation

K-Fold Cross Validation is an alternative to the (80/20) train/test split approach.

- In K-Fold CV, the data is split into k folds. The model trains on (k–1) folds and tests on the remaining fold, repeating this k times.
- The final accuracy is the average score across all folds, giving a more reliable performance estimate.

```
# Import library
from sklearn.model_selection import KFold, cross_val_score, cross_val_predict
# Initialize K-Fold cross validation
kf = KFold(n_splits=5, shuffle=True, random_state=42)
```

Classifiers

```
# traning the model on each fold and calculating the accuracy score
scores = cross_val_score(model, x, y, cv=kf, scoring='accuracy')
print(f"Accuracy for each fold: {scores}")

# Average accuracy
average_accuracy = np.mean(scores)
print(f"Average Accuracy: {average_accuracy:.2f}")

# Get cross-validated predictions
y_pred = cross_val_predict(model, x, y, cv=kf)

# Print Global classification report
print("Classification Report of K-Fold CV:")
print(classification_report(y, y_pred))

 # Confusion matrix
cm = confusion_matrix(y, y_pred)
plt.figure(figsize=(6,5))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.title("Confusion Matrix")
plt.show()
```

Regressors

```
# Train the model on each fold and calculating R² score
scores = cross_val_score(model, x, y, cv=kf, scoring='r2')
print(f"R² Score for each fold: {scores}")

# Average R²
average_r2 = np.mean(scores)
print(f"Average R² Score: {average_r2:.3f}")

# Get cross-validated predictions
y_pred = cross_val_predict(model, x, y, cv=kf)

# print regression metrics
print("\nGlobal Evaluation Metrics:")
mse= mean_squared_error(y, y_pred)
print("MSE :" , mse )
print("RMSE:", np.sqrt(mse))
print("MAE:", mean_absolute_error(y, y_pred))
print("R²:", r2_score(y, y_pred))

# plot predicted vs actual
plt.figure(figsize=(4,4))
plt.scatter(y, y_pred, alpha=0.5, s=20)
plt.plot([y.min(), y.max()], [y.min(), y.max()], 'r--')
plt.xlabel("Actual"); plt.ylabel("Predicted");
plt.title("Predicted vs Actual")
plt.show()
```

What is left to study as per plan?

- Regularization (L1 and L2 – Ridge, Lasso and ElasticNet)
- Support Vector Machine (SVM)
- Missing data handling and scaler