

```

import argparse
import json
import os
import sys
import time
import warnings
from dataclasses import dataclass
from typing import Dict, List, Tuple, Optional

import numpy as np
import pandas as pd

from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.pipeline import Pipeline
from sklearn.metrics import (
    accuracy_score, f1_score, roc_auc_score, classification_report,
    mean_squared_error, mean_absolute_error, r2_score
)
from sklearn.model_selection import train_test_split, GridSearchCV, StratifiedKFold,
KFold

# Models
from sklearn.linear_model import LogisticRegression, Ridge, Lasso
from sklearn.ensemble import RandomForestClassifier, RandomForestRegressor,
GradientBoostingClassifier, GradientBoostingRegressor

# Feature selection
from sklearn.feature_selection import SelectKBest, mutual_info_classif,
mutual_info_regression
from sklearn.feature_selection import SelectFromModel

warnings.filterwarnings("ignore", category=UserWarning)

# ----- Utilities -----

def timestamp() -> str:
    return time.strftime("%Y%m%d_%H%M%S")

def ensure_dir(path: str) -> None:
    os.makedirs(path, exist_ok=True)

def is_classification_target(y: pd.Series, explicit_task: Optional[str] = None) -> bool:
    if explicit_task in {"classification", "regression"}:
        return explicit_task == "classification"
    # Auto-detect: treat as classification if dtype is object/category/bool OR
    # number of unique values is relatively small.
    if pd.api.types.is_bool_dtype(y) or pd.api.types.is_categorical_dtype(y) or y.dtype
    == "object":
        return True
    unique = y.nunique(dropna=True)
    # Heuristic: if <= 20 unique or unique/len <= 0.05, assume classification
    if unique <= 20 or (unique / max(len(y), 1)) <= 0.05:
        return True
    return False

def get_feature_names(preprocessor: ColumnTransformer, X: pd.DataFrame) -> List[str]:
    names = []
    for name, trans, cols in preprocessor.transformers_:
        if name == "remainder":
            continue

```

```

    if hasattr(trans, "named_steps"):
        # Drill down to OneHotEncoder to get feature names
        last = list(trans.named_steps.values())[-1]
        if isinstance(last, OneHotEncoder):
            try:
                ohe = last
                ohe_features = ohe.get_feature_names_out(cols).tolist()
            except Exception:
                # Fallback
                ohe_features = [f"{c}__OHE" for c in cols]
            names.extend(ohe_features)
        elif hasattr(last, "get_feature_names_out"):
            names.extend(last.get_feature_names_out(cols).tolist())
        else:
            # For StandardScaler or SimpleImputer etc.
            if isinstance(cols, list):
                names.extend(cols)
            else:
                names.append(str(cols))
    else:
        # If it's a plain transformer like 'drop' or string
        if isinstance(cols, list):
            names.extend(cols)
        else:
            names.append(str(cols))
return names

def model_feature_importances(model, feature_names: List[str]) -> pd.DataFrame:
    """
    Extract feature importances or coefficients for a fitted model if available.
    Returns a dataframe sorted by importance desc.
    """
    coefs = None
    importances = None
    name = model.__class__.__name__
    if hasattr(model, "feature_importances_"):
        importances = model.feature_importances_
        df = pd.DataFrame({"feature": feature_names, "importance": importances})
        return df.sort_values("importance", ascending=False).reset_index(drop=True)
    if hasattr(model, "coef_"):
        coef = model.coef_
        if coef.ndim == 1:
            coefs = coef
        else:
            # Multiclass -> average absolute coefs across classes
            coefs = np.mean(np.abs(coef), axis=0)
        df = pd.DataFrame({"feature": feature_names, "importance": np.abs(coefs)})
        return df.sort_values("importance", ascending=False).reset_index(drop=True)
    # Fallback
    return pd.DataFrame({"feature": feature_names, "importance": np.nan})

def save_json(obj, path: str):
    with open(path, "w", encoding="utf-8") as f:
        json.dump(obj, f, indent=2, ensure_ascii=False)

# ----- Core Runner -----

@dataclass
class RunConfig:
    data_path: str
    target: str
    task: str = "auto" # "auto", "classification", "regression"

```

```

id_cols: Optional[List[str]] = None
test_size: float = 0.2
random_state: int = 42
cv_folds: int = 5
n_jobs: int = -1
k_values: Tuple[int, ...] = (5, 10, 20, "all") # for SelectKBest

class AssignmentRunner:
    def __init__(self, cfg: RunConfig):
        self.cfg = cfg
        self.outputs_dir = os.path.join("outputs", timestamp())
        ensure_dir(self.outputs_dir)

    def load_data(self) -> Tuple[pd.DataFrame, pd.Series]:
        df = pd.read_csv(self.cfg.data_path)
        if self.cfg.id_cols:
            for c in self.cfg.id_cols:
                if c in df.columns:
                    df = df.drop(columns=[c])
        if self.cfg.target not in df.columns:
            raise ValueError(f"Target column '{self.cfg.target}' not in dataset! Columns:
{list(df.columns)}")
        y = df[self.cfg.target]
        X = df.drop(columns=[self.cfg.target])
        return X, y

    def build_preprocessor(self, X: pd.DataFrame) -> ColumnTransformer:
        numeric_features = X.select_dtypes(include=["int64", "float64", "int32",
"float32", "bool"]).columns.tolist()
        categorical_features = [c for c in X.columns if c not in numeric_features]

        numeric_transformer = Pipeline(steps=[
            ("imputer", SimpleImputer(strategy="median")),
            ("scaler", StandardScaler())
        ])

        # Using sparse=False for broad sklearn compatibility; adjust if needed.
        categorical_transformer = Pipeline(steps=[
            ("imputer", SimpleImputer(strategy="most_frequent")),
            ("onehot", OneHotEncoder(handle_unknown="ignore", sparse=False))
        ])

        preprocessor = ColumnTransformer(
            transformers=[
                ("num", numeric_transformer, numeric_features),
                ("cat", categorical_transformer, categorical_features),
            ]
        )
        return preprocessor

    def base_models_and_grids(self, is_classification: bool):
        if is_classification:
            models = {
                "LogisticRegression": LogisticRegression(max_iter=200,
solver="liblinear"),
                "RandomForest":
RandomForestClassifier(random_state=self.cfg.random_state),
                "GradientBoosting":
GradientBoostingClassifier(random_state=self.cfg.random_state)
            }
            grids = {
                "LogisticRegression": {"model__penalty": ["l1", "l2"], "model__C": [0.1,

```

```

1, 10]],

    "RandomForest": {
        "model__n_estimators": [200, 500],
        "model__max_depth": [None, 10, 20],
        "model__min_samples_split": [2, 5],
        "model__min_samples_leaf": [1, 2]
    },
    "GradientBoosting": {
        "model__n_estimators": [100, 300],
        "model__learning_rate": [0.05, 0.1],
        "model__max_depth": [2, 3]
    }
}
else:
    models = {
        "Ridge": Ridge(random_state=self.cfg.random_state) if "random_state" in
Ridge().get_params() else Ridge(),
        "RandomForest":
RandomForestRegressor(random_state=self.cfg.random_state),
        "GradientBoosting":
GradientBoostingRegressor(random_state=self.cfg.random_state)
    }
    grids = {
        "Ridge": {"model__alpha": [0.1, 1.0, 10.0]},
        "RandomForest": {
            "model__n_estimators": [200, 500],
            "model__max_depth": [None, 10, 20],
            "model__min_samples_split": [2, 5],
            "model__min_samples_leaf": [1, 2]
        },
        "GradientBoosting": {
            "model__n_estimators": [100, 300],
            "model__learning_rate": [0.05, 0.1],
            "model__max_depth": [2, 3]
        }
    }
    return models, grids

def build_cv(self, is_classification: bool):
    if is_classification:
        return StratifiedKFold(n_splits=self.cfg.cv_folds, shuffle=True,
random_state=self.cfg.random_state)
    return KFold(n_splits=self.cfg.cv_folds, shuffle=True,
random_state=self.cfg.random_state)

def evaluate_classification(self, y_true, y_pred, y_proba=None) -> Dict:
    metrics = {
        "accuracy": float(accuracy_score(y_true, y_pred)),
        "f1_macro": float(f1_score(y_true, y_pred, average="macro"))
    }
    # ROC AUC only for binary classifications
    if y_proba is not None and len(np.unique(y_true)) == 2:
        try:
            auc = roc_auc_score(y_true, y_proba[:, 1] if y_proba.ndim == 2 else
y_proba)
            metrics["roc_auc"] = float(auc)
        except Exception:
            pass
    return metrics

def evaluate_regression(self, y_true, y_pred) -> Dict:
    return {

```

```

        "rmse": float(np.sqrt(mean_squared_error(y_true, y_pred))),
        "mae": float(mean_absolute_error(y_true, y_pred)),
        "r2": float(r2_score(y_true, y_pred))
    }

    def run_grid_search(self, X_train, y_train, preprocessor, is_classification: bool) ->
    Tuple[Dict, Dict]:
        results = {}
        best_estimators = {}

        models, grids = self.base_models_and_grids(is_classification)
        cv = self.build_cv(is_classification)
        scoring = "accuracy" if is_classification else "neg_mean_squared_error"

        for name, model in models.items():
            pipe = Pipeline(steps=[("preprocessor", preprocessor), ("model", model)])
            grid = grids[name]
            gs = GridSearchCV(pipe, grid, cv=cv, n_jobs=self.cfg.n_jobs, scoring=scoring,
refit=True)
            gs.fit(X_train, y_train)
            results[name] = {
                "best_params": gs.best_params_,
                "best_score_cv": float(gs.best_score_)
            }
            best_estimators[name] = gs.best_estimator_
        return results, best_estimators

    def run_feature_selection_and_modeling(
        self, X_train, y_train, X_test, y_test, preprocessor, is_classification: bool
    ) -> Dict:
        """Two methods: (A) SelectKBest mutual info, (B) SelectFromModel (tree-based)."""
        fs_reports = {}
        cv = self.build_cv(is_classification)
        scoring = "accuracy" if is_classification else "neg_mean_squared_error"

        # Choose proper mutual info function
        mi_func = mutual_info_classif if is_classification else mutual_info_regression

        # A) SelectKBest
        models, grids = self.base_models_and_grids(is_classification)
        sk_results = {}
        for model_name, model in models.items():
            pipe = Pipeline(steps=[
                ("preprocessor", preprocessor),
                ("feature_selection", SelectKBest(score_func=mi_func, k=10)),
                ("model", model)
            ])
            grid = dict(grids[model_name]) # copy base grid
            grid.update({"feature_selection__k": list(self.cfg.k_values)})
            gs = GridSearchCV(pipe, grid, cv=cv, n_jobs=self.cfg.n_jobs, scoring=scoring,
refit=True)
            gs.fit(X_train, y_train)
            best_model = gs.best_estimator_

            # Evaluate
            y_pred = best_model.predict(X_test)
            if is_classification:
                y_proba = None
                if hasattr(best_model, "predict_proba"):
                    try:
                        y_proba = best_model.predict_proba(X_test)
                    except Exception:

```

```

        y_proba = None
        metrics = self.evaluate_classification(y_test, y_pred, y_proba)
    else:
        metrics = self.evaluate_regression(y_test, y_pred)

    # Get selected feature names
    # Fit a clone to learn selected mask easier (or extract from pipeline)
    fs_step = best_model.named_steps["feature_selection"]
    preproc_step = best_model.named_steps["preprocessor"]
    feature_names = get_feature_names(preproc_step, X_train)
    if hasattr(fs_step, "get_support"):
        mask = fs_step.get_support()
        selected_features = [f for f, m in zip(feature_names, mask) if m]
    else:
        selected_features = feature_names

    sk_results[model_name] = {
        "best_params": gs.best_params_,
        "best_score_cv": float(gs.best_score_),
        "test_metrics": metrics,
        "selected_features": selected_features
    }
    fs_reports["SelectKBest"] = sk_results

    # B) SelectFromModel using RandomForest as selector (works for both tasks)
    selector_est = RandomForestClassifier(random_state=self.cfg.random_state) if
is_classification else RandomForestRegressor(random_state=self.cfg.random_state)

    sfm_results = {}
    for model_name, model in models.items():
        pipe = Pipeline(steps=[
            ("preprocessor", preprocessor),
            ("feature_selection", SelectFromModel(estimator=selector_est,
threshold="median")),
            ("model", model)
        ])
        grid = dict(grid)
        # Add selector parameters if desired (e.g., n_estimators), but keep simple
for runtime
        gs = GridSearchCV(pipe, grid, cv=cv, n_jobs=self.cfg.n_jobs, scoring=scoring,
refit=True)
        gs.fit(X_train, y_train)
        best_model = gs.best_estimator_

        y_pred = best_model.predict(X_test)
        if is_classification:
            y_proba = None
            if hasattr(best_model, "predict_proba"):
                try:
                    y_proba = best_model.predict_proba(X_test)
                except Exception:
                    y_proba = None
            metrics = self.evaluate_classification(y_test, y_pred, y_proba)
        else:
            metrics = self.evaluate_regression(y_test, y_pred)

        fs_step = best_model.named_steps["feature_selection"]
        preproc_step = best_model.named_steps["preprocessor"]
        feature_names = get_feature_names(preproc_step, X_train)

        if hasattr(sfm := fs_step, "get_support"):
            mask = sfm.get_support()

```

```

        selected_features = [f for f, m in zip(feature_names, mask) if m]
    else:
        selected_features = feature_names

    sfm_results[model_name] = {
        "best_params": gs.best_params_,
        "best_score_cv": float(gs.best_score_),
        "test_metrics": metrics,
        "selected_features": selected_features
    }
    fs_reports["SelectFromModel_RF"] = sfm_results

    return fs_reports

def fit(self):
    # 1) Load
    X, y = self.load_data()
    is_cls = is_classification_target(y, explicit_task=self.cfg.task)

    # 2) Split
    stratify = y if is_cls and y.nunique() > 1 else None
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=self.cfg.test_size, random_state=self.cfg.random_state,
        stratify=stratify
    )

    # 3) Preprocessor (missing values handled here)
    preprocessor = self.build_preprocessor(X)

    # 4) Base modeling with grid search
    base_report, best_estimators = self.run_grid_search(X_train, y_train,
        preprocessor, is_cls)

    # Evaluate base on test
    base_metrics = {}
    for name, est in best_estimators.items():
        y_pred = est.predict(X_test)
        if is_cls:
            y_proba = None
            if hasattr(est, "predict_proba"):
                try:
                    y_proba = est.predict_proba(X_test)
                except Exception:
                    y_proba = None
            base_metrics[name] = self.evaluate_classification(y_test, y_pred,
                y_proba)
        else:
            base_metrics[name] = self.evaluate_regression(y_test, y_pred)

    # Also try to collect feature importances where available
    feat_names = get_feature_names(preprocessor, X_train)
    try:
        if hasattr(est.named_steps["model"], "feature_importances_") or
            hasattr(est.named_steps["model"], "coef_"):
            imp_df = model_feature_importances(est.named_steps["model"],
                feat_names)
            imp_path = os.path.join(self.outputs_dir,
                f"base_feature_importances_{name}.csv")
            imp_df.to_csv(imp_path, index=False)
        except Exception:
            pass

```

```

        # 5) Feature selection + modeling
        fs_reports = self.run_feature_selection_and_modeling(X_train, y_train, X_test,
y_test, preprocessor, is_cls)

        # 6) Save everything
        report = {
            "task": "classification" if is_cls else "regression",
            "base_cv_results": base_report,
            "base_test_metrics": base_metrics,
            "feature_selection_reports": fs_reports
        }
        ensure_dir(self.outputs_dir)
        save_json(report, os.path.join(self.outputs_dir, "report.json"))

        print("\n=== SUMMARY ===")
        print(json.dumps(report, indent=2))

def parse_args() -> RunConfig:
    p = argparse.ArgumentParser(description="Assignment 2 - ML Pipeline with Missing
Value Handling, Grid Search, and Feature Selection.")
    p.add_argument("--data", dest="data_path", required=True, help="Path to CSV
dataset.")
    p.add_argument("--target", dest="target", required=True, help="Name of the target
column.")
    p.add_argument("--task", choices=["auto", "classification", "regression"],
default="auto", help="Task type. Defaults to auto-detect.")
    p.add_argument("--id-cols", dest="id_cols", nargs="*", default=None, help="Optional
ID/index columns to drop before training.")
    p.add_argument("--test-size", type=float, default=0.2, help="Test size fraction.
Default 0.2")
    p.add_argument("--cv-folds", type=int, default=5, help="Cross-validation folds for
GridSearchCV. Default 5")
    p.add_argument("--random-state", type=int, default=42, help="Random seed. Default
42")
    p.add_argument("--n-jobs", type=int, default=-1, help="Parallel jobs for
GridSearchCV. Default -1 (use all cores)")
    args = p.parse_args()

    return RunConfig(
        data_path=args.data_path,
        target=args.target,
        task=args.task,
        id_cols=args.id_cols,
        test_size=args.test_size,
        cv_folds=args.cv_folds,
        random_state=args.random_state,
        n_jobs=args.n_jobs
    )

if __name__ == "__main__":
    cfg = parse_args()
    runner = AssignmentRunner(cfg)
    runner.fit()

```