```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import missingno
import seaborn as sns
import matplotlib.pyplot as plt
from scipy.stats import skew, norm, probplot
from sklearn.preprocessing import PowerTransformer, RobustScaler, StandardScaler
from sklearn.model_selection import train_test_split, cross_val_score, KFold, TimeSeriesSplit
from sklearn.linear_model import Ridge, Lasso, LassoCV, LinearRegression
from sklearn.ensemble import GradientBoostingRegressor, StackingRegressor,
ExtraTreesRegressor, VotingRegressor
from sklearn.feature_selection import mutual_info_regression, SelectKBest
from sklearn.pipeline import make_pipeline
from sklearn.metrics import mean_squared_error
from sklearn.feature_selection import SelectKBest, mutual_info_regression

import optuna
import xgboost as xgb
import lightgbm as lgb
import catboost
from lazypredict.Supervised import LazyRegressor

pd.set_option('display.max_columns', None)
plt.style.use('ggplot')

## Data Loading with Proper Separation
train = pd.read_csv('/kaggle/input/home-data-for-ml-course/train.csv')
test = pd.read_csv('/kaggle/input/home-data-for-ml-course/test.csv')

# Save IDs for submission
test_id = test['Id']

# Create target and features
y_train = train['SalePrice']
X_train = train.drop('SalePrice', axis=1)
X_test = test.copy()

def eda_report(df):
    # Missing values
    missing = df.isnull().sum()
    missing = missing[missing > 0].sort_values(ascending=False)
    missing_pct = (missing / len(df)) * 100

    # Numeric features analysis
    numeric = df.select_dtypes(include=[np.number])
    skewness = numeric.apply(lambda x: skew(x.dropna())).sort_values(ascending=False)
    kurtosis = numeric.kurtosis().sort_values(ascending=False)

    # Categorical features analysis
    categorical = df.select_dtypes(include=['object'])
    cat_summary = {}
    for col in categorical:
```

```
        cat_summary[col] = df[col].value_counts().shape[0]

    return {
        'missing_values': missing,
        'missing_percentage': missing_pct,
        'skewness': skewness,
        'kurtosis': kurtosis,
        'categorical_cardinality': cat_summary
    }

train_report = eda_report(X_train)
test_report = eda_report(X_test)

print(train_report)

print(test_report)

def create_features(df):
    df = df.copy()

    # Age features
    df['HouseAge'] = df['YrSold'] - df['YearBuilt']
    df['RemodAge'] = df['YrSold'] - df['YearRemodAdd']
    df['GarageAge'] = df['YrSold'] - df['GarageYrBlt']

    # Total area features
    df['TotalSF'] = df['TotalBsmtSF'] + df['1stFlrSF'] + df['2ndFlrSF']
    df['TotalPorchSF'] = df['OpenPorchSF'] + df['EnclosedPorch'] + df['3SsnPorch'] +
df['ScreenPorch']

    # Bathroom features
    df['TotalBath'] = df['FullBath'] + 0.5*df['HalfBath'] + df['BsmtFullBath'] +
0.5*df['BsmtHalfBath']

    # Quality interactions
    df['OverallQual_TotalSF'] = df['OverallQual'] * df['TotalSF']
    df['OverallQual_GrLivArea'] = df['OverallQual'] * df['GrLivArea']

    # Seasonality
    df['SoldInSummer'] = df['MoSold'].isin([6,7,8]).astype(int)

    # Simplify rare categories
    df['MSZoning'] = df['MSZoning'].replace({'C (all)': 'C', 'FV': 'RM', 'RH': 'RM'})
    df['Exterior1st'] = df['Exterior1st'].replace({'Brk Cmn': 'BrkFace', 'Stone': 'Other', 'ImStucc':
'Other'})

    return df

X_train = create_features(X_train)
X_test = create_features(X_test)

X_train.shape
```

```
## Handle Missing Values
def handle_missing(df, train_df=None, is_train=True):
    df = df.copy()

    # Columns to drop (high missing or low variance)
    drop_cols = ['PoolQC', 'MiscFeature', 'Alley', 'Fence', 'FireplaceQu',
                 'Utilities', 'Street', 'LandSlope', 'Condition2', 'RoofMatl']
    df = df.drop(drop_cols, axis=1, errors='ignore')

    # Numeric columns - fill with median from training data
    numeric_cols = df.select_dtypes(include=[np.number]).columns
    if is_train:
        numeric_medians = df[numeric_cols].median()
    else:
        numeric_medians = train_df[numeric_cols].median()

    for col in numeric_cols:
        df[col] = df[col].fillna(numeric_medians[col])

    # Categorical columns - fill with mode from training data
    categorical_cols = df.select_dtypes(include=['object']).columns
    if is_train:
        categorical_modes = df[categorical_cols].mode().iloc[0]
    else:
        categorical_modes = train_df[categorical_cols].mode().iloc[0]

    for col in categorical_cols:
        df[col] = df[col].fillna(categorical_modes[col])

    return df

X_train = handle_missing(X_train, is_train=True)
X_test = handle_missing(X_test, train_df=X_train, is_train=False)

# Log transform the target
y_train = np.log1p(y_train)

# Identify skewed features
numeric_features = X_train.select_dtypes(include=[np.number]).columns
skewness = X_train[numeric_features].apply(lambda x: skew(x.dropna()))
high_skew = skewness[abs(skewness) > 0.5].index

# Apply Yeo-Johnson transformation to skewed features
pt = PowerTransformer(method='yeo-johnson')
X_train[high_skew] = pt.fit_transform(X_train[high_skew])
X_test[high_skew] = pt.transform(X_test[high_skew])

# Cap outliers using robust statistics
def cap_outliers(df, cols):
    df = df.copy()
    for col in cols:
```

```
        q1 = df[col].quantile(0.05)
        q3 = df[col].quantile(0.95)
        iqr = q3 - q1
        lower = q1 - 3 * iqr
        upper = q3 + 3 * iqr
        df[col] = np.clip(df[col], lower, upper)
    return df

X_train = cap_outliers(X_train, numeric_features)
X_test = cap_outliers(X_test, numeric_features)

# Select top features using mutual information
def select_features(X, y, k=50):
    # First encode categoricals
    X_encoded = pd.get_dummies(X, drop_first=True)

    # Calculate mutual information
    mi = mutual_info_regression(X_encoded, y, random_state=42)
    mi = pd.Series(mi, index=X_encoded.columns)
    mi = mi.sort_values(ascending=False)

    # Select top k features
    selected = mi.head(k).index

    # Return both encoded data and selected features
    return X_encoded, selected

# Get encoded data and selected features
X_train_encoded, selected_features = select_features(X_train, y_train, k=50)

# Apply same encoding to test data
X_test_encoded = pd.get_dummies(X_test, drop_first=True)

# Ensure test has same columns as train
missing_cols = set(X_train_encoded.columns) - set(X_test_encoded.columns)
for col in missing_cols:
    X_test_encoded[col] = 0
X_test_encoded = X_test_encoded[X_train_encoded.columns]

# Now filter both datasets
X_train = X_train_encoded[selected_features]
X_test = X_test_encoded[selected_features]

# Feature selection first (reduce to top 25 features)
selector = SelectKBest(mutual_info_regression, k=25)
X_train_selected = selector.fit_transform(X_train, y_train)
X_test_selected = selector.transform(X_test)
selected_features = X_train.columns[selector.get_support()]
X_train = pd.DataFrame(X_train_selected, columns=selected_features)
X_test = pd.DataFrame(X_test_selected, columns=selected_features)

# Robust scaling for most features
```

```
# Scale numeric features
numeric_cols = X_train.select_dtypes(include=['int64','float64']).columns
scaler = RobustScaler()
X_train[numeric_cols] = scaler.fit_transform(X_train[numeric_cols])
X_test[numeric_cols] = scaler.transform(X_test[numeric_cols])

# One-hot encoding for categoricals
X_train = pd.get_dummies(X_train, drop_first=True)
X_test = pd.get_dummies(X_test, drop_first=True)

# Ensure test has same columns as train
missing_cols = set(X_train.columns) - set(X_test.columns)
for col in missing_cols:
    X_test[col] = 0
X_test = X_test[X_train.columns]

def xgb_objective(trial):
    params = {
        'n_estimators': trial.suggest_int('n_estimators', 100, 3000),
        'learning_rate': trial.suggest_float('learning_rate', 0.001, 0.3, log=True),
        'max_depth': trial.suggest_int('max_depth', 2, 12),
        'subsample': trial.suggest_float('subsample', 0.6, 1.0),
        'colsample_bytree': trial.suggest_float('colsample_bytree', 0.6, 1.0),
        'gamma': trial.suggest_float('gamma', 0, 1),
        'reg_alpha': trial.suggest_float('reg_alpha', 0, 10),
        'reg_lambda': trial.suggest_float('reg_lambda', 0, 10),
        'min_child_weight': trial.suggest_int('min_child_weight', 1, 20)
    }

    model = xgb.XGBRegressor(**params, random_state=42, n_jobs=-1)
    score = cross_val_score(model, X_train, y_train,
                            scoring='neg_root_mean_squared_error',
                            cv=5, n_jobs=-1)
    return np.mean(score)

xgb_study = optuna.create_study(direction='maximize')
xgb_study.optimize(xgb_objective, n_trials=100, timeout=3600)

best_xgb = xgb.XGBRegressor(**xgb_study.best_params, random_state=42, n_jobs=-1)
best_xgb.fit(X_train, y_train)

def lgbm_objective(trial):
    params = {
        'objective': 'regression',
        'metric': 'rmse',
        'boosting_type': trial.suggest_categorical('boosting_type', ['gbdt', 'dart', 'goss']),
        'n_estimators': trial.suggest_int('n_estimators', 100, 2000),
        'num_leaves': trial.suggest_int('num_leaves', 31, 256),
        'max_depth': trial.suggest_int('max_depth', 3, 15),
        'learning_rate': trial.suggest_float('learning_rate', 0.005, 0.2, log=True),
        'min_child_samples': trial.suggest_int('min_child_samples', 5, 100),
        'subsample': trial.suggest_float('subsample', 0.7, 1.0),
```

```
        'colsample_bytree': trial.suggest_float('colsample_bytree', 0.7, 1.0),
        'reg_alpha': trial.suggest_float('reg_alpha', 0.0, 10.0),
        'reg_lambda': trial.suggest_float('reg_lambda', 0.0, 10.0),
        'min_split_gain': trial.suggest_float('min_split_gain', 0.0, 1.0),
        'min_child_weight': trial.suggest_float('min_child_weight', 1e-5, 1e2, log=True),
        'random_state': 42,
        'n_jobs': -1,
        'verbosity': -1
    }

    model = lgb.LGBMRegressor(**params)
    score = cross_val_score(
        model,
        X_train,
        y_train,
        scoring='neg_root_mean_squared_error',
        cv=5,
        n_jobs=-1
    )
    return np.mean(score)

# Run optimization
lgbm_study = optuna.create_study(direction='maximize')
lgbm_study.optimize(lgbm_objective, n_trials=50, timeout=1800)

# Train final model with early stopping
X_train_lgb, X_val_lgb, y_train_lgb, y_val_lgb = train_test_split(
    X_train, y_train, test_size=0.2, random_state=42
)
best_lgbm = lgb.LGBMRegressor(**lgbm_study.best_params, random_state=42, n_jobs=-1,
verbose=10)
best_lgbm.fit(
    X_train_lgb,
    y_train_lgb,
    eval_set=[(X_val_lgb, y_val_lgb)],
    eval_metric='rmse'
)

def catboost_objective(trial):
    params = {
        'iterations': trial.suggest_int('iterations', 500, 2000),
        'depth': trial.suggest_int('depth', 4, 10),
        'learning_rate': trial.suggest_float('learning_rate', 0.001, 0.3, log=True),
        'l2_leaf_reg': trial.suggest_float('l2_leaf_reg', 1e-5, 10),
        'random_strength': trial.suggest_float('random_strength', 1e-5, 10),
        'bagging_temperature': trial.suggest_float('bagging_temperature', 0, 10),
        'border_count': trial.suggest_int('border_count', 32, 255)
    }

    model = catboost.CatBoostRegressor(**params, random_state=42, verbose=0)
    score = cross_val_score(model, X_train, y_train,
                           scoring='neg_root_mean_squared_error',
```

```
                        cv=5, n_jobs=-1)
    return np.mean(score)

catboost_study = optuna.create_study(direction='maximize')
catboost_study.optimize(catboost_objective, n_trials=50, timeout=1800)

best_catboost = catboost.CatBoostRegressor(**catboost_study.best_params, random_state=42,
verbose=0)
best_catboost.fit(X_train, y_train)

# Create a robust stacking model
stack = StackingRegressor(
    estimators=[
        ('xgb', best_xgb),
        ('lgbm', best_lgbm),
        ('catboost', best_catboost)
    ],
    final_estimator=make_pipeline(
        RobustScaler(),
        LassoCV(cv=5, random_state=42)
    ),
    cv=5,
    n_jobs=-1
)

stack.fit(X_train, y_train)

## Make Predictions
# Predict on test set
y_pred = stack.predict(X_test)
y_pred = np.expm1(y_pred)  # Reverse log transform


result = pd.DataFrame()
result['Id'] = test_id
result['SalePrice'] = y_pred
result.to_csv('submission.csv', index=False)
```