```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
import zipfile

zip_path = 'solar_data.zip'  # Update the name if different
extract_path = '/content/solar_data'

with zipfile.ZipFile(zip_path, 'r') as zip_ref:
    zip_ref.extractall(extract_path)

print("✅ Zip file extracted to:", extract_path)
```

```
✅ Zip file extracted to: /content/solar_data
```

```
!pip uninstall tensorflow -y
!pip install tensorflow
!pip install catboost lightgbm xgboost optuna
!pip install scikit-learn --upgrade
```

```
Found existing installation: tensorflow 2.18.0
Uninstalling tensorflow-2.18.0:
  Successfully uninstalled tensorflow-2.18.0
Collecting tensorflow
  Downloading tensorflow-2.19.0-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (4.1 kB)
Requirement already satisfied: absl-py>=1.0.0 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (1.4.0)
Requirement already satisfied: astunparse>=1.6.0 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (1.6.3)
Requirement already satisfied: flatbuffers>=24.3.25 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (25.2.10)
Requirement already satisfied: gast!=0.5.0,!=0.5.1,!=0.5.2,>=0.2.1 in /usr/local/lib/python3.11/dist-packages (from tensorflow
Requirement already satisfied: google-pasta>=0.1.1 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (0.2.0)
Requirement already satisfied: libclang>=13.0.0 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (18.1.1)
Requirement already satisfied: opt-einsum>=2.3.2 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (3.4.0)
Requirement already satisfied: packaging in /usr/local/lib/python3.11/dist-packages (from tensorflow) (24.2)
Requirement already satisfied: protobuf!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<6.0.0dev,>=3.20.3 in /usr/local/
Requirement already satisfied: requests<3,>=2.21.0 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (2.32.3)
Requirement already satisfied: setuptools in /usr/local/lib/python3.11/dist-packages (from tensorflow) (75.2.0)
Requirement already satisfied: six>=1.12.0 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (1.17.0)
Requirement already satisfied: termcolor>=1.1.0 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (3.1.0)
Requirement already satisfied: typing-extensions>=3.6.6 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (4.14.0)
Requirement already satisfied: wrapt>=1.11.0 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (1.17.2)
Requirement already satisfied: grpcio<2.0,>=1.24.3 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (1.72.1)
Collecting tensorboard~=2.19.0 (from tensorflow)
  Downloading tensorboard-2.19.0-py3-none-any.whl.metadata (1.8 kB)
Requirement already satisfied: keras>=3.5.0 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (3.8.0)
Requirement already satisfied: numpy<2.2.0,>=1.26.0 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (2.0.2)
Requirement already satisfied: h5py>=3.11.0 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (3.13.0)
Collecting ml-dtypes<1.0.0,>=0.5.1 (from tensorflow)
  Downloading ml_dtypes-0.5.1-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (21 kB)
Requirement already satisfied: tensorflow-io-gcs-filesystem>=0.23.1 in /usr/local/lib/python3.11/dist-packages (from tensorflo
Requirement already satisfied: wheel<1.0,>=0.23.0 in /usr/local/lib/python3.11/dist-packages (from astunparse>=1.6.0->tensorfl
Requirement already satisfied: rich in /usr/local/lib/python3.11/dist-packages (from keras>=3.5.0->tensorflow) (13.9.4)
Requirement already satisfied: namex in /usr/local/lib/python3.11/dist-packages (from keras>=3.5.0->tensorflow) (0.1.0)
Requirement already satisfied: optree in /usr/local/lib/python3.11/dist-packages (from keras>=3.5.0->tensorflow) (0.16.0)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.11/dist-packages (from requests<3,>=2.21.0->
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.11/dist-packages (from requests<3,>=2.21.0->tensorflow)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.11/dist-packages (from requests<3,>=2.21.0->tensor
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.11/dist-packages (from requests<3,>=2.21.0->tensor
Requirement already satisfied: markdown>=2.6.8 in /usr/local/lib/python3.11/dist-packages (from tensorboard~=2.19.0->tensorflo
Requirement already satisfied: tensorboard-data-server<0.8.0,>=0.7.0 in /usr/local/lib/python3.11/dist-packages (from tensorboa
Requirement already satisfied: werkzeug>=1.0.1 in /usr/local/lib/python3.11/dist-packages (from tensorboard~=2.19.0->tensorflo
Requirement already satisfied: MarkupSafe>=2.1.1 in /usr/local/lib/python3.11/dist-packages (from werkzeug>=1.0.1->tensorboard
Requirement already satisfied: markdown-it-py>=2.2.0 in /usr/local/lib/python3.11/dist-packages (from rich->keras>=3.5.0->tens
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in /usr/local/lib/python3.11/dist-packages (from rich->keras>=3.5.0->te
Requirement already satisfied: mdurl~=0.1 in /usr/local/lib/python3.11/dist-packages (from markdown-it-py>=2.2.0->rich->keras>=
Downloading tensorflow-2.19.0-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (644.9 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 644.9/644.9 MB 724.3 kB/s eta 0:00:00
Downloading ml_dtypes-0.5.1-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (4.7 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 4.7/4.7 MB 41.6 MB/s eta 0:00:00
Downloading tensorboard-2.19.0-py3-none-any.whl (5.5 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 5.5/5.5 MB 40.5 MB/s eta 0:00:00
Installing collected packages: ml-dtypes, tensorboard, tensorflow
  Attempting uninstall: ml-dtypes
    Found existing installation: ml-dtypes 0.4.1
    Uninstalling ml-dtypes-0.4.1:
      Successfully uninstalled ml-dtypes-0.4.1
  Attempting uninstall: tensorboard
```

```
    Found existing installation: tensorboard 2.18.0
```

```
# Ensure tensorflow is installed correctly
!pip install tensorflow

# Import necessary libraries
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import KFold, train_test_split, cross_val_score
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import StandardScaler, RobustScaler, QuantileTransformer
from sklearn.ensemble import RandomForestRegressor
from catboost import CatBoostRegressor
from lightgbm import LGBMRegressor
import lightgbm as lgb
from xgboost import XGBRegressor
# Import TensorFlow and Keras components
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Input, BatchNormalization, LeakyReLU
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
from tensorflow.keras.optimizers import Adam

import warnings
warnings.filterwarnings('ignore')

# Verify TensorFlow installation and version (optional, but good for debugging)
print(f"TensorFlow version: {tf.__version__}")

# The rest of your code follows...
```

Show hidden output

```
import pandas as pd

# ✅ Set correct path to the dataset folder
data_path = '/content/solar_data/dataset'

# ✅ Load CSV files using f-strings
train = pd.read_csv(f'{data_path}/train.csv')
test = pd.read_csv(f'{data_path}/test.csv')
sample_submission = pd.read_csv(f'{data_path}/sample_submission.csv')  # Make sure this file exists

# ✅ Print shapes
print(f"Train shape: {train.shape}")
print(f"Test shape: {test.shape}")
```

```
Train shape: (20000, 17)
Test shape: (12000, 16)
```

```
# Advanced data cleaning and preprocessing
def advanced_preprocessing(train, test):
    # Convert potential string columns in numeric_cols to numeric, coercing errors
    numeric_cols = ['temperature', 'irradiance', 'humidity', 'panel_age', 'maintenance_count',
                    'soiling_ratio', 'voltage', 'current', 'module_temperature', 'cloud_coverage',
                    'wind_speed', 'pressure']

    for col in numeric_cols:
        if col in train.columns:
            # Convert to numeric, forcing errors to NaN
            train[col] = pd.to_numeric(train[col], errors='coerce')
        if col in test.columns:
             # Convert to numeric, forcing errors to NaN
            test[col] = pd.to_numeric(test[col], errors='coerce')

    # Handle missing values more sophisticatedly
    train['error_code'] = train['error_code'].fillna('No_Error')
    test['error_code'] = test['error_code'].fillna('No_Error')
    train['installation_type'] = train['installation_type'].fillna('Unknown')
    test['installation_type'] = test['installation_type'].fillna('Unknown')

    # More nuanced efficiency cleaning
    train['efficiency'] = train['efficiency'].clip(lower=0.01, upper=0.99)
```

```
    # Advanced missing value imputation for test set
    # numeric_cols is already defined above

    for col in numeric_cols:
        if col in test.columns:
            # Use more sophisticated imputation
            if col == 'panel_age':
                test[col] = test[col].fillna(test[col].median())
            elif col == 'module_temperature':
                # Impute based on temperature correlation - ensure 'temperature' is also numeric
                # (handled by the conversion loop above)
                mask = test[col].isna()
                # Check if 'temperature' column exists and is numeric before using it
                if 'temperature' in test.columns and pd.api.types.is_numeric_dtype(test['temperature']):
                     test.loc[mask, col] = test.loc[mask, 'temperature'] * 1.2 + 2
                else: # Fallback imputation if temperature is not usable
                     test[col] = test[col].fillna(train[col].median())
            else:
                # Ensure the column in train is numeric before calculating median
                if col in train.columns and pd.api.types.is_numeric_dtype(train[col]):
                    test[col] = test[col].fillna(train[col].median())
                else:
                    # Fallback imputation if train median cannot be calculated (should not happen with coerce)
                    print(f"Warning: Cannot compute median for train['{col}']. Using 0 for fillna in test['{col}'].")
                    test[col] = test[col].fillna(0) # Or some other default value


    # Handle zero values more intelligently
    for col in ['voltage', 'current']:
        # Ensure columns are numeric before handling zeros
        if col in train.columns and pd.api.types.is_numeric_dtype(train[col]):
             zero_mask_train = train[col] == 0
             # Ensure median is calculated on non-zero numeric values
             median_nonzero_train = train[train[col] > 0][col].median()
             if not pd.isna(median_nonzero_train):
                train.loc[zero_mask_train, col] = median_nonzero_train
             else:
                 # If no non-zero values, use overall median or other value
                 train.loc[zero_mask_train, col] = train[col].median() # Fallback to overall median

        if col in test.columns and pd.api.types.is_numeric_dtype(test[col]):
             zero_mask_test = test[col] == 0
             # Use the median from train's non-zero numeric values
             if not pd.isna(median_nonzero_train): # Use the value calculated for train
                 test.loc[zero_mask_test, col] = median_nonzero_train
             else:
                 # Fallback imputation for test if train median is NaN
                 test.loc[zero_mask_test, col] = test[col].median() # Fallback to test overall median


    return train, test

train, test = advanced_preprocessing(train, test)
```

```
# Enhanced feature engineering
def create_advanced_features(df, is_train=True):
    # Basic power calculation
    df['power'] = df['voltage'] * df['current']

    # Enhanced effective irradiance calculation
    df['effective_irradiance'] = (df['irradiance'] *
                                  (1 - df['soiling_ratio']) *
                                  (1 - df['cloud_coverage'] / 100) *
                                  (1 - np.maximum(0, df['module_temperature'] - 25) * 0.004))

    # Temperature differentials
    df['temp_diff'] = df['module_temperature'] - df['temperature']
    df['temp_ratio'] = df['module_temperature'] / (df['temperature'] + 1e-6)

    # Advanced age binning
    df['age_group'] = pd.cut(df['panel_age'],
                             bins=[0, 2, 7, 15, 25, 50],
                             labels=['very_new', 'new', 'mid', 'old', 'very_old'])

    # String-based features
```

```
        df['panels_per_string'] = df.groupby('string_id')['id'].transform('count')
        df['string_avg_age'] = df.groupby('string_id')['panel_age'].transform('mean')

        # Physics-based features
        df['theoretical_power'] = df['irradiance'] * 0.2 * (1 - (df['module_temperature'] - 25) * 0.004)
        df['power_efficiency_ratio'] = df['power'] / (df['theoretical_power'] + 1e-6)

        # Environmental interaction features
        df['irradiance_temp_interaction'] = df['irradiance'] * np.exp(-df['module_temperature'] / 100)
        df['wind_cooling_effect'] = df['wind_speed'] * (df['module_temperature'] - df['temperature'])
        df['humidity_temp_stress'] = df['humidity'] * df['module_temperature'] / 100

        # Maintenance and degradation features
        df['maintenance_efficiency'] = df['maintenance_count'] / (df['panel_age'] + 1)
        df['age_soiling_interaction'] = df['panel_age'] * df['soiling_ratio']
        df['degradation_factor'] = 1 - (df['panel_age'] * 0.007)  # Typical 0.7% per year

        # Advanced polynomial and interaction terms
        df['irradiance_squared'] = df['irradiance'] ** 2
        df['irradiance_log'] = np.log1p(df['irradiance'])
        df['voltage_current_ratio'] = df['voltage'] / (df['current'] + 1e-6)
        df['current_per_irradiance'] = df['current'] / (df['irradiance'] + 1e-6)

        # Atmospheric pressure effects
        df['pressure_normalized'] = (df['pressure'] - 1013.25) / 1013.25  # Normalized to sea level

        # Cloud and weather interactions
        df['clear_sky_factor'] = (100 - df['cloud_coverage']) / 100
        df['weather_performance'] = df['clear_sky_factor'] * (1 - df['humidity'] / 200)

        # Panel stress indicators
        df['thermal_stress'] = np.maximum(0, df['module_temperature'] - 85)  # High temp stress
        df['electrical_stress'] = df['voltage'] * df['current'] / (df['irradiance'] + 1e-6)

        # Time-based degradation (assuming sequential IDs relate to time)
        df['operational_index'] = df['id'] / df['id'].max()  # Proxy for operational time

        return df

    train = create_advanced_features(train, True)
    test = create_advanced_features(test, False)
```

```
    # Enhanced encoding
    def advanced_encoding(train, test):
        # Target encoding with regularization
        target_cols = ['string_id', 'error_code']

        for col in target_cols:
            # Global mean for regularization
            global_mean = train['efficiency'].mean()

            # Calculate mean and count for each category
            agg = train.groupby(col)['efficiency'].agg(['mean', 'count']).reset_index()

            # Bayesian target encoding with regularization
            alpha = 10  # Regularization strength
            agg[f'{col}_enc'] = (agg['count'] * agg['mean'] + alpha * global_mean) / (agg['count'] + alpha)

            # Map to train and test
            train[f'{col}_enc'] = train[col].map(agg.set_index(col)[f'{col}_enc'])
            test[f'{col}_enc'] = test[col].map(agg.set_index(col)[f'{col}_enc']).fillna(global_mean)

        # One-hot encoding for installation_type and age_group
        categorical_cols = ['installation_type', 'age_group']
        train_encoded = pd.get_dummies(train, columns=categorical_cols, prefix=categorical_cols)
        test_encoded = pd.get_dummies(test, columns=categorical_cols, prefix=categorical_cols)

        # Ensure test has all columns from train
        missing_cols = set(train_encoded.columns) - set(test_encoded.columns)
        for col in missing_cols:
            if col != 'efficiency':
                test_encoded[col] = 0

        # Reorder columns
        feature_cols = [col for col in train_encoded.columns if col not in ['id', 'efficiency', 'string_id', 'error_code']]
```

```
    test_encoded = test_encoded.reindex(columns=feature_cols, fill_value=0)

    return train_encoded, test_encoded

train_encoded, test_encoded = advanced_encoding(train, test)
```

```
# Feature selection and scaling
def prepare_features(train_df, test_df):
    # Define feature columns
    feature_cols = [col for col in train_df.columns if col not in ['id', 'efficiency', 'string_id', 'error_code']]

    X = train_df[feature_cols]
    y = train_df['efficiency']
    X_test = test_df[feature_cols]

    # Remove features with low variance or high correlation
    from sklearn.feature_selection import VarianceThreshold
    from sklearn.feature_selection import SelectKBest, f_regression
    from sklearn.impute import SimpleImputer # Import SimpleImputer

    # Remove low variance features
    var_threshold = VarianceThreshold(threshold=0.01)
    X_var = var_threshold.fit_transform(X)
    X_test_var = var_threshold.transform(X_test)

    # Impute remaining NaNs after VarianceThreshold
    # Use median strategy as it's robust to outliers
    imputer = SimpleImputer(strategy='median')
    X_var_imputed = imputer.fit_transform(X_var)
    X_test_var_imputed = imputer.transform(X_test_var)


    # Select top features
    # Apply SelectKBest to the imputed data
    selector = SelectKBest(score_func=f_regression, k=min(50, X_var_imputed.shape[1]))
    X_selected = selector.fit_transform(X_var_imputed, y)
    X_test_selected = selector.transform(X_test_var_imputed)

    # Get selected feature names - Need to map back to original columns before variance threshold
    # and then get the indices of the selected features from the thresholded list
    features_after_var = np.array(feature_cols)[var_threshold.get_support()]
    selected_features = features_after_var[selector.get_support()]


    # Create DataFrames with selected features
    # The index alignment is lost during the numpy transformation; recreate with original index
    X_final = pd.DataFrame(X_selected, columns=selected_features, index=X.index)
    X_test_final = pd.DataFrame(X_test_selected, columns=selected_features, index=X_test.index)

    return X_final, y, X_test_final, selected_features

X, y, X_test, selected_features = prepare_features(train_encoded, test_encoded)
print(f"Selected {len(selected_features)} features")
```

```
Selected 41 features
```

```
# Advanced scaling
scaler = QuantileTransformer(output_distribution='normal', random_state=42)
X_scaled = pd.DataFrame(scaler.fit_transform(X), columns=X.columns, index=X.index)
X_test_scaled = pd.DataFrame(scaler.transform(X_test), columns=X_test.columns, index=X_test.index)
```

```
# Improved model configurations
models = {
    'catboost': CatBoostRegressor(
        depth=8,
        iterations=1000,
        learning_rate=0.03,
        l2_leaf_reg=3,
        bagging_temperature=0.8,
        random_strength=0.8,
        bootstrap_type='Bayesian',
        verbose=0,
        random_state=42
    ),
    'lightgbm': LGBMRegressor(
```

```
            n_estimators=1200,
            learning_rate=0.04,
            max_depth=7,
            num_leaves=64,
            min_child_samples=20,
            min_split_gain=0.001,
            subsample=0.8,
            colsample_bytree=0.8,
            reg_alpha=0.1,
            reg_lambda=0.1,
            force_col_wise=True,
            random_state=42
        ),
        'xgboost': XGBRegressor(
            n_estimators=800,
            learning_rate=0.035,
            max_depth=7,
            subsample=0.85,
            colsample_bytree=0.8,
            reg_alpha=0.1,
            reg_lambda=0.1,
            gamma=0.01,
            random_state=42
        ),
        'random_forest': RandomForestRegressor(
            n_estimators=300,
            max_depth=12,
            min_samples_split=5,
            min_samples_leaf=2,
            max_features='sqrt',
            random_state=42,
            n_jobs=-1
        )
    }
```

```
    # Enhanced neural network
    def create_enhanced_nn(input_shape):
        model = Sequential([
            Input(shape=(input_shape,)),
            Dense(512, activation='relu'),
            BatchNormalization(),
            Dropout(0.3),
            Dense(256, activation='relu'),
            BatchNormalization(),
            Dropout(0.3),
            Dense(128, activation='relu'),
            BatchNormalization(),
            Dropout(0.2),
            Dense(64, activation='relu'),
            Dropout(0.2),
            Dense(32, activation='relu'),
            Dense(1, activation='sigmoid')  # Sigmoid to constrain output between 0-1
        ])

        optimizer = Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999)
        model.compile(optimizer=optimizer, loss='mse', metrics=['mae'])
        return model
```

yaha se extra vala code hai

```
    # # Multi-seed ensemble training
    # def train_ensemble_with_seeds(X, y, X_test, base_models_dict, seeds=[42, 123, 456, 789, 2024]):
    #     n_splits = 8  # Increased folds
    #     kf = KFold(n_splits=n_splits, shuffle=True, random_state=42)

    #     # Determine the total number of models (tree-based + NN)
    #     n_models = len(base_models_dict) + 1
    #     test_preds_all = np.zeros((len(X_test), n_models))
    #     oof_preds_all = np.zeros((len(X), n_models))
    #     cv_scores = []

    #     # We will store the cross-validation scores for the first seed to report later
    #     first_seed_cv_scores = []
```

```
#     for seed_idx, seed in enumerate(seeds):
#         print(f"\nTraining with seed {seed}")

#         # Tree-based models
#         # Create fresh instances of models for each seed
#         current_models = {}
#         for name, model_class in base_models_dict.items():
#              # CatBoostRegressor and XGBoostRegressor require class instantiation
#              # LGBMRegressor and RandomForestRegressor work with set_params directly
#              # However, creating a new instance is safer and clearer
#              if name == 'catboost':
#                  current_models[name] = CatBoostRegressor(**model_class.get_params()) # Use existing params
#              elif name == 'xgboost':
#                   current_models[name] = XGBRegressor(**model_class.get_params()) # Use existing params
#              elif name == 'lightgbm':
#                   current_models[name] = LGBMRegressor(**model_class.get_params()) # Use existing params
#              elif name == 'random_forest':
#                   current_models[name] = RandomForestRegressor(**model_class.get_params()) # Use existing params


#         for model_idx, (name, model) in enumerate(current_models.items()):
#             # Set the random state for the current seed
#             model.set_params(random_state=seed)
#             oof_preds = np.zeros(len(X))
#             test_preds = np.zeros(len(X_test))

#             print(f"  Training {name} for seed {seed}...")

#             for fold, (train_idx, val_idx) in enumerate(kf.split(X)):
#                 X_train_fold = X.iloc[train_idx]
#                 X_val_fold = X.iloc[val_idx]
#                 y_train_fold = y.iloc[train_idx]
#                 y_val_fold = y.iloc[val_idx]

#                 if name == 'lightgbm':
#                     model.fit(
#                         X_train_fold, y_train_fold,
#                         eval_set=[(X_val_fold, y_val_fold)],
#                         callbacks=[lgb.early_stopping(50, verbose=False)]
#                     )
#                 elif name == 'xgboost':
#                     model.fit(
#                         X_train_fold, y_train_fold,
#                         eval_set=[(X_val_fold, y_val_fold)],
#                         verbose=False
#                     )
#                 elif name == 'catboost':
#                     model.fit(
#                         X_train_fold, y_train_fold,
#                         eval_set=[(X_val_fold, y_val_fold)],
#                         early_stopping_rounds=50,
#                         verbose=False
#                     )
#                 else:
#                     model.fit(X_train_fold, y_train_fold)

#                 oof_preds[val_idx] = model.predict(X_val_fold)
#                 test_preds += model.predict(X_test) / n_splits

#             # Clip predictions
#             oof_preds = np.clip(oof_preds, 0.01, 0.99)
#             test_preds = np.clip(test_preds, 0.01, 0.99)

#             cv_score = 100 * (1 - np.sqrt(mean_squared_error(y, oof_preds)))
#             print(f"    {name} OOF CV Score (Seed {seed}): {cv_score:.6f}")

#             # Accumulate predictions averaged across seeds
#             test_preds_all[:, model_idx] += test_preds / len(seeds)
#             oof_preds_all[:, model_idx] += oof_preds / len(seeds)

#             if seed_idx == 0:
#                 first_seed_cv_scores.append(cv_score)

#         # Neural Network
#         print(f"  Training Neural Network for seed {seed}...")
#         oof_preds_nn = np.zeros(len(X))
#         test_preds_nn = np.zeros(len(X_test))
```

```
#         # NN also needs a fresh instance per seed if its training is part of the seed loop
#         # Although the NN itself is trained per fold, the overall OOF and test predictions
#         # accumulate across seeds, requiring a separate NN process for each seed.
#         # Alternatively, you could train NNs outside the seed loop if their seeds are handled differently.
#         # For consistency with the tree models in this structure, we'll retrain/re-initialize per seed.
#         # Note: This means the *same* NN architecture is used, but initialized and trained independently for each seed.
#         # If you wanted a truly different NN per seed, you'd modify the NN creation itself based on the seed.
#         # Here, we assume the NN structure is fixed, but its training starts fresh per seed.

#         for fold, (train_idx, val_idx) in enumerate(kf.split(X)):
#             X_train_fold = X_scaled.iloc[train_idx].values
#             X_val_fold = X_scaled.iloc[val_idx].values
#             y_train_fold = y.iloc[train_idx].values
#             y_val_fold = y.iloc[val_idx].values

#             # Re-create and re-compile the NN model for each fold/seed combination
#             nn_model = create_enhanced_nn(X.shape[1])
#             early_stopping = EarlyStopping(monitor='val_loss', patience=15, restore_best_weights=True)
#             lr_scheduler = ReduceLROnPlateau(monitor='val_loss', factor=0.7, patience=8, min_lr=1e-6)

#             nn_model.fit(
#                 X_train_fold, y_train_fold,
#                 validation_data=(X_val_fold, y_val_fold),
#                 epochs=200,
#                 batch_size=64,
#                 callbacks=[early_stopping, lr_scheduler],
#                 verbose=0
#             )

#             val_pred = nn_model.predict(X_val_fold, verbose=0).flatten()
#             oof_preds_nn[val_idx] = np.clip(val_pred, 0.01, 0.99)

#             test_pred = nn_model.predict(X_test_scaled.values, verbose=0).flatten()
#             test_preds_nn += np.clip(test_pred, 0.01, 0.99) / n_splits # Average test predictions across folds for this seed

#         cv_score_nn = 100 * (1 - np.sqrt(mean_squared_error(y, oof_preds_nn)))
#         print(f"    Neural Network OOF CV Score (Seed {seed}): {cv_score_nn:.6f}")

#         # Accumulate NN predictions averaged across seeds
#         test_preds_all[:, -1] += test_preds_nn / len(seeds) # Average the per-seed, per-fold-averaged test preds
#         oof_preds_all[:, -1] += oof_preds_nn / len(seeds) # Average the per-seed OOF preds

#         if seed_idx == 0:
#             first_seed_cv_scores.append(cv_score_nn)

#     # Return the CV scores calculated only from the first seed, or calculate mean across seeds if preferred
#     # Returning first seed scores is common for model comparison
#     cv_scores_reported = first_seed_cv_scores

#     return oof_preds_all, test_preds_all, cv_scores_reported

# # Train ensemble
# print("Training ensemble models...")

# # Pass the models dictionary to the updated function
# oof_predictions, test_predictions, cv_scores_per_model = train_ensemble_with_seeds(X_scaled, y, X_test_scaled, models)

# print(f"\nIndividual model CV scores (from first seed): {cv_scores_per_model}")
# print(f"Average CV score (from first seed): {np.mean(cv_scores_per_model):.6f}")

# # Note: The stacking phase uses oof_predictions and test_predictions which are
# # now the averages over all seeds. The reported cv_scores_per_model are just
# # for information from the first seed.
```

```
import numpy as np
from sklearn.model_selection import KFold
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error
from xgboost import XGBRegressor
from catboost import CatBoostRegressor
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau

# Neural Network architecture
```

```
def create_nn(input_dim):
    model = Sequential([
        Dense(128, activation='relu', input_dim=input_dim),
        Dropout(0.3),
        Dense(64, activation='relu'),
        Dropout(0.2),
        Dense(1, activation='sigmoid')
    ])
    model.compile(optimizer='adam', loss='mean_squared_error')
    return model

# Simplified training function with ensemble models
def train_ensemble(X, y, X_test, X_scaled, X_test_scaled, seeds=[42, 123]):
    n_splits = 5
    kf = KFold(n_splits=n_splits, shuffle=True, random_state=42)
    test_preds_all = np.zeros((len(X_test), 4))
    oof_preds_all = np.zeros((len(X), 4))

    for seed in seeds:
        models = [
            RandomForestRegressor(n_estimators=100, random_state=seed),
            XGBRegressor(n_estimators=100, random_state=seed, verbosity=0),
            CatBoostRegressor(iterations=100, random_seed=seed, verbose=0)
        ]

        for i, model in enumerate(models):
            oof = np.zeros(len(X))
            test_fold_preds = np.zeros(len(X_test))
            for train_idx, val_idx in kf.split(X):
                model.fit(X.iloc[train_idx], y.iloc[train_idx])
                oof[val_idx] = model.predict(X.iloc[val_idx])
                test_fold_preds += model.predict(X_test) / n_splits

            oof = np.clip(oof, 0.01, 0.99)
            test_fold_preds = np.clip(test_fold_preds, 0.01, 0.99)
            oof_preds_all[:, i] += oof / len(seeds)
            test_preds_all[:, i] += test_fold_preds / len(seeds)

        # Train Neural Network per seed
        oof_nn = np.zeros(len(X))
        test_nn = np.zeros(len(X_test))
        for train_idx, val_idx in kf.split(X_scaled):
            model_nn = create_nn(X_scaled.shape[1])
            es = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
            rlrop = ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=5)
            model_nn.fit(
                X_scaled[train_idx], y.iloc[train_idx],
                validation_data=(X_scaled[val_idx], y.iloc[val_idx]),
                epochs=100, batch_size=64, callbacks=[es, rlrop], verbose=0
            )
            oof_nn[val_idx] = model_nn.predict(X_scaled[val_idx]).flatten()
            test_nn += model_nn.predict(X_test_scaled).flatten() / n_splits

        oof_nn = np.clip(oof_nn, 0.01, 0.99)
        test_nn = np.clip(test_nn, 0.01, 0.99)
        oof_preds_all[:, -1] += oof_nn / len(seeds)
        test_preds_all[:, -1] += test_nn / len(seeds)

    return oof_preds_all, test_preds_all
```

```
models = {
    'xgboost': XGBRegressor(),
    'lightgbm': LGBMRegressor(),
    'catboost': CatBoostRegressor(verbose=0),
    'random_forest': RandomForestRegressor()
}
```

```
# Advanced stacking with multiple meta-models
from sklearn.linear_model import Ridge, ElasticNet
from sklearn.ensemble import GradientBoostingRegressor

meta_models = {
    'lgb_meta': LGBMRegressor(n_estimators=300, learning_rate=0.05, max_depth=4, random_state=42),
    'ridge_meta': Ridge(alpha=1.0),
```

```
    'elastic_meta': ElasticNet(alpha=0.1, l1_ratio=0.7, random_state=42),
    'gb_meta': GradientBoostingRegressor(n_estimators=200, learning_rate=0.05, max_depth=3, random_state=42)
}
```

```
# Train meta-models
X_meta_train, X_meta_val, y_meta_train, y_meta_val = train_test_split(oof_predictions, y, test_size=0.2, random_state=42)

best_meta_score = 0
best_meta_name = None
best_meta_model = None

for name, meta_model in meta_models.items():
    meta_model.fit(X_meta_train, y_meta_train)
    meta_val_pred = meta_model.predict(X_meta_val)
    meta_val_pred = np.clip(meta_val_pred, 0.01, 0.99)

    meta_score = 100 * (1 - np.sqrt(mean_squared_error(y_meta_val, meta_val_pred)))
    print(f"{name} meta-model score: {meta_score:.6f}")

    if meta_score > best_meta_score:
        best_meta_score = meta_score
        best_meta_name = name
        best_meta_model = meta_model

print(f"\nBest meta-model: {best_meta_name} with score: {best_meta_score:.6f}")
```

```
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of testing was 0.000985 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 1275
[LightGBM] [Info] Number of data points in the train set: 16000, number of used features: 5
[LightGBM] [Info] Start training from score 0.511378
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
```

```
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
```

```
# Generate final predictions
if best_meta_score > np.mean(cv_scores_per_model):
    print("Using meta-model for final predictions")
    final_predictions = best_meta_model.predict(test_predictions)
else:
    print("Using weighted average for final predictions")
    # Optimal weights based on CV scores
    weights = np.array(cv_scores_per_model) / np.sum(cv_scores_per_model)
    final_predictions = np.average(test_predictions, axis=1, weights=weights)
```

```
Using weighted average for final predictions
```

```
# Final post-processing
final_predictions = np.clip(final_predictions, 0.01, 0.99)
```

```
# Adaptive calibration
train_mean = y.mean()
pred_mean = final_predictions.mean()
calibration_factor = train_mean / pred_mean
final_predictions = final_predictions * calibration_factor
final_predictions = np.clip(final_predictions, 0.01, 0.99)
```

```
# Create submission
submission = pd.DataFrame({
    'id': test['id'],
    'efficiency': final_predictions
})
print(f"\nSubmission statistics:")
print(f"Mean: {submission['efficiency'].mean():.6f}")
print(f"Std: {submission['efficiency'].std():.6f}")
print(f"Min: {submission['efficiency'].min():.6f}")
print(f"Max: {submission['efficiency'].max():.6f}")
```

```

Submission statistics:
Mean: 0.510576
Std: 0.093398
Min: 0.244888
Max: 0.896117
```

```
# Save submission
submission.to_csv('/content/submission_enhanced.csv', index=False)
# submission.to_csv('/content/drive/MyDrive/Solar_Panel_Dataset/submission_enhanced.csv', index=False)
```

```
# Download submission
from google.colab import files
files.download('/content/submission_enhanced.csv')

print(f"\nFinal ensemble score estimate: {best_meta_score:.6f}")
print("Enhanced submission file created successfully!")
```

```

Final ensemble score estimate: 89.555503
Enhanced submission file created successfully!
```

```
import pandas as pd

# Load your CSV file
df = pd.read_csv('submission_enhanced.csv')

# Boost efficiency values by 0.02, and clip within the range [0.01, 0.99]
df['efficiency'] = (df['efficiency'] + 0.02).clip(lower=0.01, upper=0.99)

# Save the new enhanced version
df.to_csv('submission_enhanced_boosted.csv', index=False)
```

```
Start coding or generate with AI.
```