Big Mart Sales Prediction Practice Problem

Knowledge and Learning

BigMart Sales Prediction - Code Documentation

This documentation explains the full workflow of the **bigmart_prediction.py** script, which prepares data, trains machine learning models, and generates predictions for the BigMart Sales dataset.

Overview

The goal is to predict the sales **(Item_Outlet_Sales)** for various products across different stores based on available features like product category, store type, visibility, and more. The pipeline includes data preprocessing, feature engineering, model selection, fine-tuning, prediction, and exporting results.

1. Imports and Setup

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.model_selection import train_test_split, GridSearchCV, KFold, cross_val_score
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor,
VotingRegressor, StackingRegressor
from sklearn.linear_model import LinearRegression, Ridge, Lasso, ElasticNet
from sklearn.svm import SVR
from sklearn.metrics import mean_squared_error, r2_score
```

```
from xgboost import XGBRegressor
from lightgbm import LGBMRegressor
import pickle
import os
import warnings
warnings.filterwarnings('ignore')
```

Why these imports?

- **Pandas & NumPy:** For data manipulation.
- **Matplotlib & Seaborn:** Optional visualizations (though not used directly here).
- **Scikit-learn modules:** For preprocessing, modeling, evaluation, and hyperparameter tuning.
- **xgboost:** optimized implementation of gradient boosting designed for performance and speed.
- **Lightgbm:** It is used to complement other models like XGBoost, Ridge, and Random Forest in a stacking ensemble.
- **Pickle:** Save the trained model.
- **Warnings:** Suppress warnings for a cleaner output.

2. Data Preparation (prepare_data())

a. Load the Data

```
# Load the datasets

train = pd.read_csv('train.csv')
test = pd.read_csv('test.csv')
```

Loads the datasets into memory to begin preprocessing.

b. Combine Train and Test

```
# Create a copy of the datasets
  train_df = train.copy()
  test_df = test.copy()

  # Add a dataset indicator column
  train_df['source'] = 'train'
  test_df['source'] = 'test'

  # Add target column to test data with NaN values
  test_df['Item_Outlet_Sales'] = np.nan

  # Combine datasets for preprocessing
  combined_df = pd.concat([train_df, test_df], ignore_index=True)
```

Adds a `source` column to distinguish training and testing data. Test set is assigned NaN for the target variable. We are combining the datasets to apply uniform transformations before splitting them back.

c. Standardize Categorical Values

```
print("Preprocessing data...")
  # Standardize Item_Fat_Content values
  combined_df['Item_Fat_Content'] = combined_df['Item_Fat_Content'].replace(
     {'LF': 'Low Fat', 'low fat': 'Low Fat', 'reg': 'Regular'}
  )
```

Multiple labels representing the same category (**e.g., "low fat" vs "LF"**) are normalized to reduce dimensionality and improve consistency.

3. Handling Missing Values

a. Item_Weight

```
# For Item_Weight: Use median for each Item_Identifier
 item_weight_median = combined_df.groupby('Item_Identifier')['Item_Weight'].median()
 for idx, item in enumerate(combined_df['Item_Identifier']):
   if pd.isna(combined_df.at[idx, 'Item_Weight']):
     if pd.notna(item_weight_median[item]):
       combined_df.at[idx, 'Item_Weight'] = item_weight_median[item]
     else:
       # If no median exists for that Item_Identifier, use global median
       combined_df.at[idx, 'Item_Weight'] = combined_df['Item_Weight'].median()
```

1. First, it **groups** the combined_df DataFrame by Item_Identifier.For each group, it **calculates the median** of the Item_Weight column.The result is a **Series**: the index is Item_Identifier, and the value is the median Item_Weight for that identifier.
2. enumerate() is used to get both the **index (idx)** and the **value (item)** for each row. item is a string like 'FDA15', and idx is the corresponding row index.Then the if function checks if the value in the Item_Weight column at row index is **NaN** (missing).

The nested if function checks whether a **median exists** for the current **Item_Identifier (item)**:

- If the identifier has some non-missing weights in the dataset, a median will be present.
- If all weights for this identifier are missing, the result will be NaN, and this condition will be False.
- If a median **does exist** for this ***Item_Identifier***, assign that median value to the missing ***Item_Weight*** in the row.

If there's **no median** available for the identifier (perhaps because all weights were missing for it), fall back to using the **global median** of the entire **Item_Weight** column (ignoring NaNs).

Why median? Median is more robust to outliers than mean. Grouping by **Item_Identifier** ensures we impute values that are contextually correct.

b. Outlet_Size

```
# For Outlet_Size: Fill missing values based on Outlet_Type
 outlet_size_mode = combined_df.groupby('Outlet_Type')['Outlet_Size'].apply(
   lambda x: x.mode()[0] if not x.mode().empty else 'Medium'
 )
 for idx, outlet_type in enumerate(combined_df['Outlet_Type']):
```

```python
    if pd.isna(combined_df.at[idx, 'Outlet_Size']):
        combined_df.at[idx, 'Outlet_Size'] = outlet_size_mode[outlet_type]
```

First, it **Groups** the DataFrame by Outlet_Type.Then, for each group, it applies a **lambda function** to get the mode (most frequent value) of the Outlet_Size column.

- If the mode exists (not empty), it uses the **first mode** (x.mode()[0] — just in case there are multiple modes).
- If the mode is empty (e.g., all values are NaN for that group), it **defaults to 'Medium'**.

The result is a **Series** with Outlet_Type as the index and the corresponding Outlet_Size mode as the value.

After that, starts a loop through each row's Outlet_Type, getting both the index (idx) and the value (outlet_type), and checks if the current row has a missing value (NaN) in the Outlet_Size column.

If it is missing, it fills in the value using the mode of the Outlet_Size for that specific Outlet_Type, which was calculated earlier and stored in outlet_size_mode.

Why mode? **Outlet_Size** is categorical, and mode provides the most frequent (and thus likely correct) value for each type of outlet.

4. Feature Engineering

```python
# Replace 0 visibility with mean visibility of that product
zero_visibility_indices = combined_df['Item_Visibility'] == 0
item_visibility_mean = combined_df.groupby('Item_Identifier')['Item_Visibility'].mean()
for idx in combined_df[zero_visibility_indices].index:
    item = combined_df.at[idx, 'Item_Identifier']
    combined_df.at[idx, 'Item_Visibility'] = item_visibility_mean[item]
```

First, we create a **boolean Series** that is True whatever Item_Visibility is 0. It's used to filter the DataFrame to find the rows needing replacement.
Next, we group the DataFrame by Item_Identifier.For each group, it calculates the **mean** of the Item_Visibility column (ignoring zeros and NaNs).

Result: a Series with Item_Identifier as index and the average visibility as value.

Next, we loop through the indices of rows where Item_Visibility is 0 (those True in the zero_visibility_indices mask.idx is the index of a row that needs fixing.)In that for loop, we get the Item_Identifier for the current row.(This is used to look up the corresponding **mean visibility**.)
The final line replaces the 0 visibility with the **mean visibility** for that Item_Identifier.

Why fix 0 visibility? It's unrealistic; most likely a missing value. Replacing with product-specific average is contextually more accurate.

a. Item_Category

```
# 2. Create Item_Identifier categories
combined_df['Item_Category'] = combined_df['Item_Identifier'].apply(lambda x: x[:2])
```

Why this feature? The first two characters in Item_Identifier indicate a broad category (e.g., "FD" = food), which can offer predictive value.

b. Outlet_Years

```
# 3. Add a feature for Outlet age
combined_df['Outlet_Years'] = 2013 - combined_df['Outlet_Establishment_Year']
```

Why create this? The age of the outlet could affect its sales volume—older outlets might have more established customer bases.

c. Normalized Visibility

```
# 4. Normalized Item_Visibility within Item_Type
combined_df['Item_Visibility_Normalized'] =
combined_df.groupby('Item_Type')['Item_Visibility'].transform(
    lambda x: x / x.mean()
)
```

Why normalize? Helps reduce product-specific bias and control for variation within item types.

5. Encoding Categorical Variables

```
# Encode categorical variables
  # Label encoding for ordinal features
  label_encoder = LabelEncoder()
  combined_df['Outlet_Size_Encoded'] =
label_encoder.fit_transform(combined_df['Outlet_Size'])
  combined_df['Outlet_Location_Type_Encoded'] =
label_encoder.fit_transform(combined_df['Outlet_Location_Type'])
  # One-hot encoding for nominal categorical features
  categorical_columns = ['Item_Fat_Content', 'Item_Type', 'Outlet_Type', 'Item_Category']
  combined_df = pd.get_dummies(combined_df, columns=categorical_columns)
```

- **Label Encoding:** Ordinal variables (Outlet_Size, Outlet_Location_Type).

  Why label encoding? Used for ordinal variables like Outlet_Size and Outlet_Location_Type, where there's a natural order.

- **One-Hot Encoding:** Nominal variables (Item_Type, Outlet_Type, etc.)

  Why one-hot encoding? For nominal variables with no intrinsic order. Ensures the model doesn't assume false hierarchies.

6. Splitting Train/Test Data

```
# Prepare the final datasets for modeling
  train_final = combined_df[combined_df['source'] == 'train'].drop('source', axis=1)
  test_final = combined_df[combined_df['source'] == 'test'].drop(['source',
'Item_Outlet_Sales'], axis=1)
  # Select features for modeling
  drop_columns = ['Item_Identifier', 'Outlet_Identifier', 'Outlet_Establishment_Year',
            'Outlet_Size', 'Outlet_Location_Type']
```

```
X_train = train_final.drop(drop_columns + ['Item_Outlet_Sales'], axis=1)
y_train = train_final['Item_Outlet_Sales']
X_test = test_final.drop(drop_columns, axis=1)
return X_train, y_train, X_test, test_final['Item_Identifier'], test_final['Outlet_Identifier']
```

Train_final: Filters the combined_df DataFrame to keep only rows where the 'source' column is 'train'.Drops the 'source' column (since it's no longer needed).The result is a cleaned-up DataFrame called train_final, ready for training.

Test_final: Filters combined_df to keep only rows labeled 'test' in the 'source' column, and drops both 'source' and 'Item_Outlet_Sales':

- 'source': same reason as above — no longer needed.
- 'Item_Outlet_Sales': this is the target variable which isn't present in the test set (or shouldn't be), as we're predicting it.

X_train: Drops all the drop_columns **plus** the **target variable** Item_Outlet_Sales from the training set. This creates the feature matrix X_train — the inputs to the model.

Y_train: Extracts the **target variable** (Item_Outlet_Sales) from the training data into y_train. This is what the model will learn to predict.

X_test: Drops the same set of drop_columns from the **test set**, creating X_test. Notice it **does not drop Item_Outlet_Sales** — likely because it doesn't exist in the test set (which makes sense in a prediction scenario).

Finally, it returns all the components needed:

- X_train: features for training.
- y_train: target values for training.
- X_test: features for testing/prediction.
- test_final['Item_Identifier'], test_final['Outlet_Identifier']: identifiers to link predictions back to the original products/outlets.

7. Model Training (train_model())

a. Train-Test Split

```
# Split training data for validation
  X_train_split, X_val, y_train_split, y_val = train_test_split(
    X_train, y_train, test_size=0.2, random_state=42 )
```

By setting test_size=0.2, we are reserving only 20% of the data to evaluate model performance before final training.

random_state=42: sets a **random seed** so the split is **reproducible** (you'll get the same split every time you run it).

b. Model Benchmarks

```
    # Base models with carefully tuned hyperparameters - sticking with stable models first
  base_models = [
    ('ridge', Ridge(alpha=0.5, random_state=42)),
    ('lasso', Lasso(alpha=0.001, random_state=42)),
    ('en', ElasticNet(alpha=0.001, l1_ratio=0.5, random_state=42)),
    ('gbr', GradientBoostingRegressor(learning_rate=0.05, n_estimators=200, max_depth=4,
random_state=42)),
    ('rf', RandomForestRegressor(n_estimators=200, max_features='sqrt', max_depth=15,
random_state=42))
  ]

  # Try to add XGBoost and LightGBM if evaluation succeeds
  try:
    xgb_model = XGBRegressor(learning_rate=0.05, n_estimators=300, max_depth=5,
colsample_bytree=0.7, random_state=42)
    xgb_model.fit(X_train_split, y_train_split)
    xgb_pred = xgb_model.predict(X_val)
    xgb_rmse = np.sqrt(mean_squared_error(y_val, xgb_pred))
    print(f"XGB test: RMSE = {xgb_rmse:.4f}")
    base_models.append(('xgb', xgb_model))
  except Exception as e:
    print(f"XGBoost model failed: {e}")
```

```
try:
    lgb_model = LGBMRegressor(learning_rate=0.05, n_estimators=300, num_leaves=31,
random_state=42)
    lgb_model.fit(X_train_split, y_train_split)
    lgb_pred = lgb_model.predict(X_val)
    lgb_rmse = np.sqrt(mean_squared_error(y_val, lgb_pred))
    print(f"LGBM test: RMSE = {lgb_rmse:.4f}")
    base_models.append(('lgbm', lgb_model))
except Exception as e:
    print(f"LightGBM model failed: {e}")
```

Why multiple models?

- Linear Regression: Baseline with interpretability.
- Random Forest: Robust to overfitting, handles non-linearities well.
- Gradient Boosting: Excellent performance with tuning.
- Ridge Regression: Captures **linear relationships** between features and target.
- Lasso Regression: Helps reduce **dimensionality** in datasets with many features.
- ElasticNet Regression: Great when you have **many features** and some are correlated.

The next two try except function is is trying to **train and evaluate two additional advanced models**:

1. **XGBoost (Extreme Gradient Boosting)**
2. **LightGBM (Light Gradient Boosting Machine)**

If the evaluation (RMSE) on a validation set is successful, the models are appended to the base ensemble list **base_models.**

- **XGBoost** is highly accurate, robust, and handles missing data efficiently.
- **LightGBM** is faster when it comes to training models, and is suitable for handling bigger data.

c. Training and RMSE Evaluation

```
# Train the model
    model.fit(X_train_split, y_train_split)

    # Predictions on validation set
    val_pred = model.predict(X_val)

    # Calculate RMSE
    val_rmse = np.sqrt(mean_squared_error(y_val, val_pred))

    print(f"{name} - Validation RMSE: {val_rmse:.4f}")

    if val_rmse < best_rmse:
        best_rmse = val_rmse
        best_model = model
        best_model_name = name

  print(f"Best model: {best_model_name} with RMSE: {best_rmse:.4f}")
```

Why RMSE(Root Mean Square Error)? A popular metric for regression problems; penalizes larger errors more than MAE.

d. Model Selection & Fine-Tuning

```
# Fine-tune the best model if it's Gradient Boosting or Random Forest
  if best_model_name in ['Gradient Boosting', 'Random Forest']:
    print("Fine-tuning the best model...")

    if best_model_name == 'Gradient Boosting':
        param_grid = {
            'n_estimators': [100, 200, 300],
            'learning_rate': [0.05, 0.1, 0.15],
```

```
            'max_depth': [3, 4, 5],
            'min_samples_split': [2, 5, 10]
        }
        base_model = GradientBoostingRegressor(random_state=42)
    else:  # Random Forest
        param_grid = {
            'n_estimators': [100, 200, 300],
            'max_depth': [None, 10, 20],
            'min_samples_split': [2, 5, 10],
            'min_samples_leaf': [1, 2, 4]
        }
        base_model = RandomForestRegressor(random_state=42)

    grid_search = GridSearchCV(
        base_model,
        param_grid=param_grid,
        cv=5,
        scoring='neg_root_mean_squared_error',
        verbose=1,
        n_jobs=-1
    )

    grid_search.fit(X_train, y_train)

    print(f"Best parameters: {grid_search.best_params_}")
    print(f"Best RMSE: {-grid_search.best_score_:.4f}")

    # Return the best model from grid search
    best_model = grid_search.best_estimator_
else:
    # Train on full training data
    best_model.fit(X_train, y_train)

return best_model
```

The first few lines in the code just checks if the name of the best model is one that supports fine-tuning.We've already determined best_model_name earlier during model evaluation in the RMSE section.

Fine-Tuning Setup (Gradient Boosting):

Defines the hyperparameter grid for GradientBoostingRegressor.And sets ranges for:

- n_estimators: number of boosting stages.
- learning_rate: how much to correct each stage.
- max_depth: depth of trees.
- min_samples_split: minimum samples to split a node.

Initializes the base model with a random seed for reproducibility.

Fine-Tuning Setup (Random Forest):

It has a similar setup to Gradient Boosting, but specific to RandomForestRegressor. It includes min_samples_leaf, which controls the minimum number of samples at each leaf node.

GridSearchCV: Used to perform cross-validated grid search.

cv=5: 5-fold cross-validation.
scoring='neg_root_mean_squared_error': RMSE is negated because GridSearchCV treats higher scores as better.
n_jobs=-1: uses all available cores for speed.
verbose=1: prints progress as it trains.

Then, we train multiple models using the parameter grid and cross-validation.

After that, it gives output for the **best combination** of hyperparameters and converts the negative RMSE back to positive for interpretation. From that, it stores the **best performing model** found during the search.

If the best model isn't one of the above, just fit it to the full training data without tuning.

At the end of the code, it returns the finalized, trained model (either tuned or directly fitted).

8. Predictions and Submission (make_prediction())

a. Predict & Save CSV

```
# Predict on test data
test_predictions = model.predict(X_test)

# Create submission file
submission = pd.DataFrame({
    'Item_Identifier': item_ids,
    'Outlet_Identifier': outlet_ids,
    'Item_Outlet_Sales': test_predictions
})

# Save submission file
submission.to_csv('submission.csv', index=False)
print("Submission file created: submission.csv")
```

Creates a Kaggle-compatible file with required predictions.

b. Save Model and return submission file

```
# Save the model
os.makedirs('models', exist_ok=True)
with open('models/best_model.pkl', 'wb') as f:
    pickle.dump(model, f)
print("Model saved: models/best_model.pkl")

return submission
```

Why pickle? For later reuse, either for testing, further tuning, or deployment.

9. Main Function

```
def main():
```

```python
"""Main function to run the entire process"""
print("BigMart Sales Prediction")
print("-----------------------")

# Prepare data
X_train, y_train, X_test, item_ids, outlet_ids = prepare_data()

# Train model
best_model = train_model(X_train, y_train)

# Make predictions and create submission
submission = make_prediction(best_model, X_test, item_ids, outlet_ids)

print("Done!")
```

The **main()** function orchestrates the whole pipeline:

1. Loads and prepares data.
2. Trains and tunes models.
3. Generates submission and saves model.

Conclusion

This document provides a robust and modular solution to the problem statement - Big Mart Sales Prediction. The codebase carefully handles missing data, encodes categorical variables appropriately, and leverages ensemble models with tuning for strong predictive performance.