Regression Modeling & Evaluation

(Submitted by - Mayank Singh)

This exercise involves building, tuning, and evaluating multiple **regression models** to predict customer spending. It walks through the complete ML pipeline: from data preprocessing and model training to evaluation using cross-validation and final holdout testing.

Dataset

The dataset includes customer-level features. This dataset contains data about whether or not different consumers made a purchase in response to a test mailing of a certain catalog and, in case of a purchase, how much money each consumer spent. The objective is to predict the `Purchase` amount.

Objectives

1. **Build numeric prediction models** that predict Spending based on the other available customer informationn.
2. **Compare multiple regression models** using RMSE as the evaluation metric.
3. **Tune and finalize** the best-performing model using Nested Cross-Validation.
4. **Interpret** final performance using RMSE on a holdout test set.

Notebook Breakdown

1. Loading Data

- Load and preprocess the dataset.

2. Model Comparison

- Define and evaluate multiple regression models:
  - `LinearRegression`
  - `KNeighborsRegressor`
  - `DecisionTreeRegressor`
  - `SVR`
  - `RandomForestRegressor`
  - `GradientBoostingRegressor`
  - `XGBRegressor`

    - `LGBMRegressor`
    - `MLPRegressor`
- Use **Nested Cross-Validation** with preprocessing (standard scaling).
- Compare based on **mean RMSE and standard deviation** across folds.

3. Final Model Selection & Evaluation

- Select the **best model** (e.g., `XGBoost` or `NeuralNet`) based on CV results.
- Perform **hyperparameter tuning** using `GridSearchCV`.
- Refit the model on the full training data.
- Evaluate on the **holdout test set**.
- Report:
    - Final RMSE

Evaluation Metric

- **RMSE (Root Mean Squared Error)**: Used consistently for cross-validation and final evaluation.
- Lower RMSE indicates better model performance.

Outcome

The notebook concludes with the **best-tuned model** evaluated on the holdout set.

> Built with: Python, `scikit-learn`, `XGBoost`, `LightGBM`, `Pandas`

Part A - Modeling on All Customers

Import Libraries

In [1]:
```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, GridSearchCV, KFold
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, make_scorer
from sklearn.linear_model import LinearRegression
from sklearn.neighbors import KNeighborsRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.svm import SVR
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.neural_network import MLPRegressor
```

```
from xgboost import XGBRegressor
from lightgbm import LGBMRegressor
from sklearn.compose import ColumnTransformer

import warnings
from sklearn.exceptions import ConvergenceWarning
warnings.filterwarnings("ignore")
warnings.filterwarnings("ignore", category=ConvergenceWarning)
warnings.filterwarnings("ignore", category=UserWarning, module="lightgbm")
```

Load Data

The dataset `hw3data.csv` is loaded into a pandas DataFrame. From this, three columns — `sequence_number`, `Spending`, and `Purchase` — are removed to isolate the predictors into `X_full`. The target variable `y_full` is defined separately using the `Spending` column. This setup ensures that only relevant features are used for training, while the target is clearly separated for modeling.

In [4]:
```
# === STEP 2: Load Data ===
df = pd.read_csv("hw3data.csv")
X_full = df.drop(columns=["sequence_number", "Spending", "Purchase"])
y_full = df["Spending"]
```

In [6]:
```
X_full.head()
```

Out[6]:

	US	source_a	source_c	source_b	source_d	source_e	source_m	source_o	source_h	sc
0	1	0	0	1	0	0	0	0	0	
1	1	0	0	0	0	1	0	0	0	
2	1	0	0	0	0	0	0	0	0	
3	1	0	1	0	0	0	0	0	0	
4	1	0	1	0	0	0	0	0	0	

5 rows × 22 columns

In [8]:
```
y_full.head()
```

Out[8]:
```
0    127.87
1      0.00
2    127.48
3      0.00
4      0.00
Name: Spending, dtype: float64
```

Holdout Split

The dataset is then split into training and holdout sets using an 80-20 split. This means 80% of the data will be used to train and validate models through cross-validation, while 20% will be reserved as an untouched set to evaluate final model performance. A fixed random seed ensures the results are reproducible across runs.

In [11]:
```
# === STEP 3: Holdout Split ===
X_train, X_holdout, y_train, y_holdout = train_test_split(X_full, y_full, test_size
```

RMSE Function

A custom function is defined to calculate RMSE (Root Mean Squared Error). It measures the average magnitude of prediction error. This function is wrapped with `make_scorer` to be compatible with scikit-learn's model selection tools like GridSearchCV, allowing RMSE to be used during hyperparameter tuning.

In [14]:
```
# === STEP 4: RMSE Function ===
def rmse(y_true, y_pred):
    return np.sqrt(mean_squared_error(y_true, y_pred))
rmse_scorer = make_scorer(rmse, greater_is_better=False)
```

Model Definitions

A dictionary named `models` is created to store all the machine learning models that will be evaluated. This includes a mix of linear, tree-based, distance-based, kernel-based, and neural models.

It covers: `Linear Regression`, `k-Nearest Neighbors (KNN)`, `Decision Tree`, `Support Vector Regression (SVR)`, a `Neural Network (MLPRegressor)`, `Random Forest`, `Gradient Boosting`, `XGBoost`, and `LightGBM`. This broad range of models allows for comparison between simple and complex learners and helps identify which types perform best for predicting spending behavior.

In [17]:
```
# === STEP 5: Model Definitions ===
models = {
    'LinearRegression': LinearRegression(),
    'KNN': KNeighborsRegressor(),
    'DecisionTree': DecisionTreeRegressor(),
    'SVM': SVR(),
    'NeuralNet': MLPRegressor(max_iter=200, random_state=42),
    'RandomForest': RandomForestRegressor(),
    'GradientBoosting': GradientBoostingRegressor(),
    'XGBoost': XGBRegressor(objective='reg:squarederror', verbosity=0),
    'LightGBM': LGBMRegressor(force_col_wise=True, enable_categorical=False,verbose
}
```

Hyperparamter Grid

Alongside the models, a dictionary called `param_grids` is defined to specify the hyperparameters to tune for each model during cross-validation.

- For all models, a range of values is specified for key parameters such as number of neighbors, tree depth, and regularization strength.
- These grids ensure that each model is optimized fairly before performance is compared. The tuning process will be handled later via GridSearchCV.

In [20]:
```
param_grids = {
    'LinearRegression': {},

    'KNN': {
        'n_neighbors': [3, 5, 7],
        'weights': ['uniform', 'distance']
    },

    'DecisionTree': {
        'max_depth': [5, 10, None],
        'min_samples_split': [2, 5]
    },

    'SVM': {
        'C': [0.1, 1, 10],
        "kernel": ['linear', 'rbf', 'poly']
    },

    'NeuralNet': {
        'hidden_layer_sizes': [(32,), (64,), (64, 32)],
        'alpha': [0.0001, 0.001, 0.01],
        'learning_rate_init': [0.0005, 0.001],
        'activation': ['relu', 'tanh'],
        'solver': ['adam']
    },

    'RandomForest': {
        'n_estimators': [100, 200],
        'max_depth': [10, None],
        'max_features': ['sqrt'],
        'min_samples_leaf': [1, 2]
    },

    'GradientBoosting': {
        'n_estimators': [100, 200],
        'learning_rate': [0.05, 0.1],
        'max_depth': [3, 5],
        'subsample': [0.8],
        'min_samples_leaf': [1, 2]
    },

    'XGBoost': {
        'n_estimators': [100, 200],
        'learning_rate': [0.05, 0.1],
        'max_depth': [3, 5],
```

```
        'subsample': [0.8],
        'colsample_bytree': [0.8],
        'reg_alpha': [0],
        'reg_lambda': [1]
    },

    'LightGBM': {
        'n_estimators': [100, 200],
        'learning_rate': [0.05, 0.1],
        'max_depth': [5, 10],
        'num_leaves': [31, 64],
        'subsample': [0.8],
        'colsample_bytree': [0.8],
        'reg_alpha': [0],
        'reg_lambda': [1]
    }
}
```

Nested Cross-Validation

- This block implements a nested cross-validation loop to evaluate and tune each model using RMSE as the performance metric. The outer loop (5-fold) is used for evaluating generalization error, while the inner loop (3-fold) is used for hyperparameter tuning via `GridSearchCV`.
- For each model in the dictionary, the data is split into training and validation sets within the outer loop. Before training, the numerical features are standardized using `StandardScaler` — the scaling is applied only on the training data and then transformed onto the validation set to prevent data leakage.
- Within each outer fold, a grid search is performed to find the best hyperparameters using the inner cross-validation loop. The best model from the inner CV is then used to predict on the outer fold's validation set. The RMSE is computed and stored for that fold.
- After all folds are completed, the average RMSE and its standard deviation across outer folds are computed for each model. This setup ensures fair and unbiased model comparison, with preprocessing handled carefully outside the pipeline to give full control over the transformations.

In [23]:
```
numeric_cols  = ["Freq", "last_update_days_ago", "1st_update_days_ago"]
binary_cols = [col for col in X_train.columns if col not in numeric_cols]
```

In [25]:
```
numeric_cols
```

Out[25]:
```
['Freq', 'last_update_days_ago', '1st_update_days_ago']
```

In [27]:
```
binary_cols
```

Out[27]:
```
['US',
 'source_a',
 'source_c',
 'source_b',
 'source_d',
 'source_e',
 'source_m',
 'source_o',
 'source_h',
 'source_r',
 'source_s',
 'source_t',
 'source_u',
 'source_p',
 'source_x',
 'source_w',
 'Web order',
 'Gender=male',
 'Address_is_res']
```

In [31]:
```python
# === STEP 6: Nested CV ===
outer_cv = KFold(n_splits=5, shuffle=True, random_state=42)
inner_cv = KFold(n_splits=3, shuffle=True, random_state=42)

results = {}
best_models = {}

for name in models.keys():
    model = models[name]
    param_grid = param_grids[name]
    outer_scores = []
    print(f"\nTraining {name}...")

    for train_idx, val_idx in outer_cv.split(X_train):
        X_tr, X_val = X_train.iloc[train_idx], X_train.iloc[val_idx]
        y_tr, y_val = y_train.iloc[train_idx], y_train.iloc[val_idx]

        preprocessor = ColumnTransformer([
            ("scale_numeric", StandardScaler(), numeric_cols),
            ("passthrough_binary", "passthrough", binary_cols)
        ])

        X_tr_scaled = preprocessor.fit_transform(X_tr)
        X_val_scaled = preprocessor.transform(X_val)

        gs = GridSearchCV(model, param_grid, scoring=rmse_scorer, cv=inner_cv)
        gs.fit(X_tr_scaled, y_tr)

        best_model = gs.best_estimator_
        y_pred = best_model.predict(X_val_scaled)
        outer_rmse = rmse(y_val, y_pred)
        outer_scores.append(outer_rmse)

    mean_rmse = np.mean(outer_scores)
    std_rmse = np.std(outer_scores)
    results[name] = outer_scores
```

```
    print(f"{name}: Mean RMSE = {mean_rmse:.4f}, Std = {std_rmse:.4f}")
```

```
Training LinearRegression...
LinearRegression: Mean RMSE = 126.1608, Std = 15.9257

Training KNN...
KNN: Mean RMSE = 133.6503, Std = 20.4117

Training DecisionTree...
DecisionTree: Mean RMSE = 148.9928, Std = 22.7721

Training SVM...
SVM: Mean RMSE = 134.6211, Std = 15.0443

Training NeuralNet...
NeuralNet: Mean RMSE = 122.4092, Std = 16.8263

Training RandomForest...
RandomForest: Mean RMSE = 130.2882, Std = 17.2483

Training GradientBoosting...
GradientBoosting: Mean RMSE = 121.5868, Std = 20.3617

Training XGBoost...
XGBoost: Mean RMSE = 121.0807, Std = 19.8520

Training LightGBM...
LightGBM: Mean RMSE = 129.1533, Std = 18.6504
```

Final Model Comparison Summary

Below is the summary of RMSE and standard deviation across all models from nested cross-validation:

Model	RMSE	Std Dev
Linear Regression	126.16	15.93
KNN	133.65	20.41
Decision Tree	148.99	22.77
SVM	134.62	15.04
Neural Net	122.41	16.83
Random Forest	130.29	17.25
Gradient Boosting	121.59	20.36
XGBoost	121.08	19.85
LightGBM	129.15	18.65

Results:

- Among all models, **XGBoost** achieved the lowest RMSE of **121.08**, closely followed by **Gradient Boosting (121.59)** and **Neural Net (122.41)**. These models effectively capture complex non-linear patterns, leading to strong performance.
- **Decision Tree** had the highest RMSE of **148.99**, along with a high standard deviation, indicating weak and unstable performance. **KNN** and **SVM** also underperformed, showing limitations in modeling capacity for this dataset.
- Classical models such as **Linear Regression** showed reasonable performance, but were clearly outperformed by ensemble and neural models. **Random Forest** and **LightGBM** performed better but still fell short of the top-performing models.
- **XGBoost** stands out as the most robust and accurate model across all folds. Based on its strong generalization performance, it is selected as the **final model** for further tuning and holdout evaluation.

In [34]:
```
# === STEP 7: Compare Results ===
results_df = pd.DataFrame(results)
results_df.loc['Mean'] = results_df.mean()
results_df.loc['Std'] = results_df.std()
print("\nFinal Nested CV RMSE Comparison:")
results_df.round(2).tail(2)
```

```
Final Nested CV RMSE Comparison:
```

Out[34]:

	LinearRegression	KNN	DecisionTree	SVM	NeuralNet	RandomForest	GradientE
Mean	126.16	133.65	148.99	134.62	122.41	130.29	
Std	15.93	20.41	22.77	15.04	16.83	17.25	

Final XGBoost Model Run

To finalize the model selection, we performed nested cross-validation using XGBoost with a carefully pruned hyperparameter grid. The outer loop was used to evaluate generalization performance, while the inner loop performed hyperparameter tuning via GridSearchCV. Preprocessing was handled using a ColumnTransformer, which scaled numeric features and passed binary variables as-is - ensuring no data leakage across folds.

- The RMSEs across the five outer folds were:

Fold	RMSE
Fold 1	106.12
Fold 2	130.76

Fold	RMSE
Fold 3	98.56
Fold 4	126.01
Fold 5	152.17

- The average RMSE across these folds was **122.72**, with a standard deviation of **18.98**, indicating strong generalization with reasonable consistency across data splits.
- After identifying the best-performing model (from Fold 3), we retrained it on the entire training set and evaluated it on the unseen holdout set. The final **Holdout RMSE was 128.43**, which aligns well with the cross-validation results.
- This confirms that the XGBoost model maintains robust predictive performance and generalizes effectively to new, unseen customer data.

In [41]:
```
# ======= RMSE Scorer =======
rmse = lambda y_true, y_pred: np.sqrt(mean_squared_error(y_true, y_pred))
rmse_scorer = make_scorer(rmse, greater_is_better=False)

# ======= XGBoost Grid =======
xgb_grid = {
    'n_estimators': [100, 200],
    'learning_rate': [0.01, 0.05, 0.1],
    'max_depth': [3, 5, 7],
    'subsample': [0.8],
    'colsample_bytree': [0.8],
    'reg_alpha': [0, 0.1],
    'reg_lambda': [1, 2],
    'min_child_weight': [1, 3],
    'gamma': [0, 0.1]
}

# ======= Nested CV Setup =======
outer_cv = KFold(n_splits=5, shuffle=True, random_state=42)
inner_cv = KFold(n_splits=3, shuffle=True, random_state=42)

outer_scores = []
best_models = []

for fold, (train_idx, val_idx) in enumerate(outer_cv.split(X_train)):
    print(f"\nOuter Fold {fold+1}/5")

    X_tr, X_val = X_train.iloc[train_idx], X_train.iloc[val_idx]
    y_tr, y_val = y_train.iloc[train_idx], y_train.iloc[val_idx]

    preprocessor = ColumnTransformer([
        ("scale_numeric", StandardScaler(), numeric_cols),
        ("passthrough_binary", "passthrough", binary_cols)
    ])
```

```
    X_tr_scaled = preprocessor.fit_transform(X_tr)
    X_val_scaled = preprocessor.transform(X_val)

    xgb = XGBRegressor(objective="reg:squarederror", verbosity=0, random_state=42)
    grid = GridSearchCV(
        xgb,
        xgb_grid,
        scoring=rmse_scorer,
        cv=inner_cv,
        n_jobs=-1,
        verbose=0
    )
    grid.fit(X_tr_scaled, y_tr)

    best_model = grid.best_estimator_
    y_val_pred = best_model.predict(X_val_scaled)
    val_rmse = rmse(y_val, y_val_pred)

    best_models.append((best_model, preprocessor, val_rmse))
    outer_scores.append(val_rmse)

    print(f"Fold {fold+1} RMSE: {val_rmse:.4f}")

# ======= Nested CV Summary =======
mean_rmse = np.mean(outer_scores)
std_rmse = np.std(outer_scores)

print("\nNested CV RMSE Results (XGBoost):")
for i, score in enumerate(outer_scores):
    print(f"Fold {i+1}: {score:.4f}")
print(f"Mean RMSE: {mean_rmse:.4f}")
print(f"Std Dev:   {std_rmse:.4f}")

# ======= Select Best Model from Outer CV =======
best_index = np.argmin(outer_scores)
best_estimator, best_scaler, best_score = best_models[best_index]
print(f"\nBest outer fold model: Fold {best_index+1} with RMSE = {best_score:.4f}")
print("Best Hyperparameters:")
for param, value in best_estimator.get_params().items():
    print(f"  {param}: {value}")

# ======= Refit on Full Training Set =======
preprocessor = ColumnTransformer([
        ("scale_numeric", StandardScaler(), numeric_cols),
        ("passthrough_binary", "passthrough", binary_cols)
    ])
X_train_scaled = preprocessor.fit_transform(X_train)
X_holdout_scaled = preprocessor.transform(X_holdout)

final_model = best_estimator.set_params(**best_estimator.get_params())
final_model.fit(X_train_scaled, y_train)

# ======= Holdout Evaluation =======
y_holdout_pred = final_model.predict(X_holdout_scaled)
holdout_rmse = rmse(y_holdout, y_holdout_pred)
print(f"\nHoldout RMSE: {holdout_rmse:.4f}")
```

```
Outer Fold 1/5
Fold 1 RMSE: 106.1169

Outer Fold 2/5
Fold 2 RMSE: 130.7626

Outer Fold 3/5
Fold 3 RMSE: 98.5594

Outer Fold 4/5
Fold 4 RMSE: 126.0075

Outer Fold 5/5
Fold 5 RMSE: 152.1651

Nested CV RMSE Results (XGBoost):
Fold 1: 106.1169
Fold 2: 130.7626
Fold 3: 98.5594
Fold 4: 126.0075
Fold 5: 152.1651
Mean RMSE: 122.7223
Std Dev:   18.9837

Best outer fold model: Fold 3 with RMSE = 98.5594
Best Hyperparameters:
  objective: reg:squarederror
  base_score: None
  booster: None
  callbacks: None
  colsample_bylevel: None
  colsample_bynode: None
  colsample_bytree: 0.8
  device: None
  early_stopping_rounds: None
  enable_categorical: False
  eval_metric: None
  feature_types: None
  feature_weights: None
  gamma: 0
  grow_policy: None
  importance_type: None
  interaction_constraints: None
  learning_rate: 0.05
  max_bin: None
  max_cat_threshold: None
  max_cat_to_onehot: None
  max_delta_step: None
  max_depth: 3
  max_leaves: None
  min_child_weight: 1
  missing: nan
  monotone_constraints: None
  multi_strategy: None
  n_estimators: 200
  n_jobs: None
```

```
num_parallel_tree: None
random_state: 42
reg_alpha: 0.1
reg_lambda: 1
sampling_method: None
scale_pos_weight: None
subsample: 0.8
tree_method: None
validate_parameters: None
verbosity: 0
```

```
Holdout RMSE: 128.4273
```

Part B - Modeling Only for Purchasers

Load Data

The dataset `hw3data.csv` is loaded into a pandas DataFrame. Here, we filtered for customers who had `Purchase = 1` for our modeling. From this, three columns — `sequence_number`, `Spending`, and `Purchase` — are removed to isolate the predictors into `X_full`. The target variable `y_full` is defined separately using the `Spending` column. This setup ensures that only relevant features are used for training, while the target is clearly separated for modeling.

In [44]:
```
# === STEP 2: Load Data ===
df = pd.read_csv("hw3data.csv")
df = df[df['Purchase']==1]
X_full = df.drop(columns=["sequence_number", "Spending", "Purchase"])
y_full = df["Spending"]
```

Holdout Split

The dataset is then split into training and holdout sets using an 80-20 split. This means 80% of the data will be used to train and validate models through cross-validation, while 20% will be reserved as an untouched set to evaluate final model performance. A fixed random seed ensures the results are reproducible across runs.

In [47]:
```
# === STEP 3: Holdout Split ===
X_train, X_holdout, y_train, y_holdout = train_test_split(X_full, y_full, test_size
```

RMSE Function

A custom function is defined to calculate RMSE (Root Mean Squared Error). It measures the average magnitude of prediction error. This function is wrapped with `make_scorer` to be

compatible with scikit-learn's model selection tools like GridSearchCV, allowing RMSE to be used during hyperparameter tuning.

In [50]:
```
# === STEP 4: RMSE Function ===
def rmse(y_true, y_pred):
    return np.sqrt(mean_squared_error(y_true, y_pred))
rmse_scorer = make_scorer(rmse, greater_is_better=False)
```

Model Definitions

A dictionary named `models` is created to store all the machine learning models that will be evaluated. This includes a mix of linear, tree-based, distance-based, kernel-based, and neural models.

It covers: `Linear Regression`, `k-Nearest Neighbors (KNN)`, `Decision Tree`, `Support Vector Regression (SVR)`, a `Neural Network (MLPRegressor)`, `Random Forest`, `Gradient Boosting`, `XGBoost`, and `LightGBM`. This broad range of models allows for comparison between simple and complex learners and helps identify which types perform best for predicting spending behavior.

In [53]:
```
# === STEP 5: Model Definitions ===
models = {
    'LinearRegression': LinearRegression(),
    'KNN': KNeighborsRegressor(),
    'DecisionTree': DecisionTreeRegressor(),
    'SVM': SVR(),
    'NeuralNet': MLPRegressor(max_iter=200, random_state=42),
    'RandomForest': RandomForestRegressor(),
    'GradientBoosting': GradientBoostingRegressor(),
    'XGBoost': XGBRegressor(objective='reg:squarederror', verbosity=0),
    'LightGBM': LGBMRegressor(force_col_wise=True, enable_categorical=False,verbose
}
```

Hyperparamter Grid

Alongside the models, a dictionary called `param_grids` is defined to specify the hyperparameters to tune for each model during cross-validation.

- For models like KNN, Decision Tree, and SVM, a range of values is specified for key parameters such as number of neighbors, tree depth, and regularization strength.
- These grids ensure that each model is optimized fairly before performance is compared. The tuning process will be handled later via GridSearchCV.

In [56]:
```
param_grids = {
    'LinearRegression': {},

    'KNN': {
```

```
        'n_neighbors': [3, 5, 7],
        'weights': ['uniform', 'distance']
    },

    'DecisionTree': {
        'max_depth': [5, 10, None],
        'min_samples_split': [2, 5]
    },

    'SVM': {
        'C': [0.1, 1, 10],
        "kernel": ['linear', 'rbf', 'poly']
    },

    'NeuralNet': {
        'hidden_layer_sizes': [(32,), (64,), (64, 32)],
        'alpha': [0.0001, 0.001, 0.01],
        'learning_rate_init': [0.0005, 0.001],
        'activation': ['relu', 'tanh'],
        'solver': ['adam']
    },

    'RandomForest': {
        'n_estimators': [100, 200],
        'max_depth': [10, None],
        'max_features': ['sqrt'],
        'min_samples_leaf': [1, 2]
    },

    'GradientBoosting': {
        'n_estimators': [100, 200],
        'learning_rate': [0.05, 0.1],
        'max_depth': [3, 5],
        'subsample': [0.8],
        'min_samples_leaf': [1, 2]
    },

    'XGBoost': {
        'n_estimators': [100, 200],
        'learning_rate': [0.05, 0.1],
        'max_depth': [3, 5],
        'subsample': [0.8],
        'colsample_bytree': [0.8],
        'reg_alpha': [0],
        'reg_lambda': [1]
    },

    'LightGBM': {
        'n_estimators': [100, 200],
        'learning_rate': [0.05, 0.1],
        'max_depth': [5, 10],
        'num_leaves': [31, 64],
        'subsample': [0.8],
        'colsample_bytree': [0.8],
        'reg_alpha': [0],
        'reg_lambda': [1]
```

```
    }
}
```

Nested Cross-Validation

- This block implements a nested cross-validation loop to evaluate and tune each model using RMSE as the performance metric. The outer loop (5-fold) is used for evaluating generalization error, while the inner loop (3-fold) is used for hyperparameter tuning via `GridSearchCV`.
- For each model in the dictionary, the data is split into training and validation sets within the outer loop. Before training, the numerical features are standardized using `StandardScaler` — the scaling is applied only on the training data and then transformed onto the validation set to prevent data leakage.
- Within each outer fold, a grid search is performed to find the best hyperparameters using the inner cross-validation loop. The best model from the inner CV is then used to predict on the outer fold's validation set. The RMSE is computed and stored for that fold.
- After all folds are completed, the average RMSE and its standard deviation across outer folds are computed for each model. This setup ensures fair and unbiased model comparison, with preprocessing handled carefully outside the pipeline to give full control over the transformations.

In [62]:
```
# === STEP 6: Nested CV ===
outer_cv = KFold(n_splits=5, shuffle=True, random_state=42)
inner_cv = KFold(n_splits=3, shuffle=True, random_state=42)

results = {}
best_models = {}

for name in models.keys():
    model = models[name]
    param_grid = param_grids[name]
    outer_scores = []
    print(f"\nTraining {name}...")

    for train_idx, val_idx in outer_cv.split(X_train):
        X_tr, X_val = X_train.iloc[train_idx], X_train.iloc[val_idx]
        y_tr, y_val = y_train.iloc[train_idx], y_train.iloc[val_idx]

        preprocessor = ColumnTransformer([
            ("scale_numeric", StandardScaler(), numeric_cols),
            ("passthrough_binary", "passthrough", binary_cols)
        ])

        X_tr_scaled = preprocessor.fit_transform(X_tr)
        X_val_scaled = preprocessor.transform(X_val)
```

```
        gs = GridSearchCV(
            model,
            param_grid,
            scoring=rmse_scorer,
            cv=inner_cv
        )
        gs.fit(X_tr_scaled, y_tr)

        best_model = gs.best_estimator_
        y_pred = best_model.predict(X_val_scaled)
        outer_rmse = rmse(y_val, y_pred)
        outer_scores.append(outer_rmse)

    mean_rmse = np.mean(outer_scores)
    std_rmse = np.std(outer_scores)
    results[name] = outer_scores
    best_models[name] = best_model

    print(f"{name}: Mean RMSE = {mean_rmse:.4f}, Std = {std_rmse:.4f}")
```

```
Training LinearRegression...
LinearRegression: Mean RMSE = 157.8116, Std = 20.8627

Training KNN...
KNN: Mean RMSE = 162.6334, Std = 20.0380

Training DecisionTree...
DecisionTree: Mean RMSE = 184.6255, Std = 13.0663

Training SVM...
SVM: Mean RMSE = 166.6539, Std = 23.2166

Training NeuralNet...
NeuralNet: Mean RMSE = 154.6944, Std = 19.9925

Training RandomForest...
RandomForest: Mean RMSE = 157.9787, Std = 22.5538

Training GradientBoosting...
GradientBoosting: Mean RMSE = 155.0353, Std = 19.2456

Training XGBoost...
XGBoost: Mean RMSE = 153.8904, Std = 19.2866

Training LightGBM...
LightGBM: Mean RMSE = 161.0574, Std = 23.2587
```

Final Model Comparison Summary

Below is the summary of RMSE and standard deviation across all models from nested cross-validation:

Model	RMSE	Std Dev
Linear Regression	157.81	20.86
KNN	162.63	20.04
Decision Tree	184.63	13.07
SVM	166.65	23.22
Neural Net	154.69	19.99
Random Forest	157.98	22.55
Gradient Boosting	155.04	19.25
XGBoost	153.89	19.29
LightGBM	161.06	23.26

Results:

- **XGBoost** achieved the best performance with the lowest RMSE of **153.89**, closely followed by **Neural Net (154.69)** and **Gradient Boosting (155.04)**. All three performed strongly, thanks to their ability to model complex non-linear relationships.
- **Decision Tree** performed the worst (RMSE = 184.63), with a relatively low standard deviation — consistently underperforming across folds.
- **SVM** and **KNN** had relatively high RMSEs, suggesting they may not be ideal for this particular regression task.
- Traditional models like **Linear Regression** and **Random Forest** offered reasonable performance, but were outperformed by boosting and neural architectures.
- Based on the trade-off between performance and consistency, **XGBoost** can be selected as the final model for retraining and holdout evaluation.

In [65]:
```
# === STEP 7: Compare Results ===
results_df = pd.DataFrame(results)
results_df.loc['Mean'] = results_df.mean()
results_df.loc['Std'] = results_df.std()
print("\nFinal Nested CV RMSE Comparison:")
results_df.round(2).tail(2)
```

```
Final Nested CV RMSE Comparison:
```

Out[65]:

	LinearRegression	KNN	DecisionTree	SVM	NeuralNet	RandomForest	GradientB
Mean	157.81	162.63	184.63	166.65	154.69	157.98	
Std	20.86	20.04	13.07	23.22	19.99	22.55	

Final XGBoost Model Run

To finalize the model selection, we performed nested cross-validation using XGBoost with a carefully defined hyperparameter grid. The outer loop was used to evaluate generalization performance, while the inner loop performed hyperparameter tuning via GridSearchCV. ColumnTransformer with `StandardScaler` was applied only on the numeric features within each fold to prevent data leakage.

- The RMSEs across the five outer folds were:

Fold	RMSE
Fold 1	191.1722
Fold 2	147.4471
Fold 3	145.6280
Fold 4	151.0174
Fold 5	139.3723

Results:

- The average RMSE across these folds was **154.93**, with a standard deviation of **18.51**, indicating stable performance with modest variation across folds.
- After identifying the best-performing model (from Fold 5), we retrained it on the entire training set and evaluated it on the unseen holdout set. The final **Holdout RMSE was 185.87**, which, while slightly higher than the cross-validation average, remains within a reasonable range.
- These results confirm that the **XGBoost model** is a strong candidate for this regression task, showing competitive performance both in validation and on unseen data.

In [67]:
```
# ======= RMSE Scorer =======
rmse = lambda y_true, y_pred: np.sqrt(mean_squared_error(y_true, y_pred))
rmse_scorer = make_scorer(rmse, greater_is_better=False)

# ======= XGBoost Grid =======
xgb_grid = {
    'n_estimators': [100, 200],
    'learning_rate': [0.01, 0.05, 0.1],
    'max_depth': [3, 5, 7],
    'subsample': [0.8],
    'colsample_bytree': [0.8],
    'reg_alpha': [0, 0.1],
    'reg_lambda': [1, 2],
```

```
    'min_child_weight': [1, 3],
    'gamma': [0, 0.1]
}

# ======= Nested CV Setup =======
outer_cv = KFold(n_splits=5, shuffle=True, random_state=42)
inner_cv = KFold(n_splits=3, shuffle=True, random_state=42)

outer_scores = []
best_models = []

for fold, (train_idx, val_idx) in enumerate(outer_cv.split(X_train)):
    print(f"\nOuter Fold {fold+1}/5")

    X_tr, X_val = X_train.iloc[train_idx], X_train.iloc[val_idx]
    y_tr, y_val = y_train.iloc[train_idx], y_train.iloc[val_idx]

    preprocessor = ColumnTransformer([
        ("scale_numeric", StandardScaler(), numeric_cols),
        ("passthrough_binary", "passthrough", binary_cols)
    ])

    X_tr_scaled = preprocessor.fit_transform(X_tr)
    X_val_scaled = preprocessor.transform(X_val)

    xgb = XGBRegressor(objective="reg:squarederror", verbosity=0, random_state=42)
    grid = GridSearchCV(
        xgb,
        xgb_grid,
        scoring=rmse_scorer,
        cv=inner_cv,
        n_jobs=-1,
        verbose=0
    )
    grid.fit(X_tr_scaled, y_tr)

    best_model = grid.best_estimator_
    y_val_pred = best_model.predict(X_val_scaled)
    val_rmse = rmse(y_val, y_val_pred)

    best_models.append((best_model, preprocessor, val_rmse))
    outer_scores.append(val_rmse)

    print(f"Fold {fold+1} RMSE: {val_rmse:.4f}")

# ======= Nested CV Summary =======
mean_rmse = np.mean(outer_scores)
std_rmse = np.std(outer_scores)

print("\nNested CV RMSE Results (XGBoost):")
for i, score in enumerate(outer_scores):
    print(f"Fold {i+1}: {score:.4f}")
print(f"Mean RMSE: {mean_rmse:.4f}")
print(f"Std Dev:   {std_rmse:.4f}")

# ======= Select Best Model from Outer CV =======
```

```python
best_index = np.argmin(outer_scores)
best_estimator, best_scaler, best_score = best_models[best_index]
print(f"\nBest outer fold model: Fold {best_index+1} with RMSE = {best_score:.4f}")
print("Best Hyperparameters:")
for param, value in best_estimator.get_params().items():
    print(f"  {param}: {value}")

# ======= Refit on Full Training Set =======
preprocessor = ColumnTransformer([
        ("scale_numeric", StandardScaler(), numeric_cols),
        ("passthrough_binary", "passthrough", binary_cols)
    ])
X_train_scaled = preprocessor.fit_transform(X_train)
X_holdout_scaled = preprocessor.transform(X_holdout)

final_model = best_estimator.set_params(**best_estimator.get_params())
final_model.fit(X_train_scaled, y_train)

# ======= Holdout Evaluation =======
y_holdout_pred = final_model.predict(X_holdout_scaled)
holdout_rmse = rmse(y_holdout, y_holdout_pred)
print(f"\nHoldout RMSE: {holdout_rmse:.4f}")
```

```
Outer Fold 1/5
Fold 1 RMSE: 191.1722

Outer Fold 2/5
Fold 2 RMSE: 147.4471

Outer Fold 3/5
Fold 3 RMSE: 145.6280

Outer Fold 4/5
Fold 4 RMSE: 151.0174

Outer Fold 5/5
Fold 5 RMSE: 139.3723

Nested CV RMSE Results (XGBoost):
Fold 1: 191.1722
Fold 2: 147.4471
Fold 3: 145.6280
Fold 4: 151.0174
Fold 5: 139.3723
Mean RMSE: 154.9274
Std Dev:   18.5115

Best outer fold model: Fold 5 with RMSE = 139.3723
Best Hyperparameters:
  objective: reg:squarederror
  base_score: None
  booster: None
  callbacks: None
  colsample_bylevel: None
  colsample_bynode: None
  colsample_bytree: 0.8
  device: None
  early_stopping_rounds: None
  enable_categorical: False
  eval_metric: None
  feature_types: None
  feature_weights: None
  gamma: 0
  grow_policy: None
  importance_type: None
  interaction_constraints: None
  learning_rate: 0.05
  max_bin: None
  max_cat_threshold: None
  max_cat_to_onehot: None
  max_delta_step: None
  max_depth: 3
  max_leaves: None
  min_child_weight: 3
  missing: nan
  monotone_constraints: None
  multi_strategy: None
  n_estimators: 100
  n_jobs: None
```

```
 num_parallel_tree: None
 random_state: 42
 reg_alpha: 0
 reg_lambda: 2
 sampling_method: None
 scale_pos_weight: None
 subsample: 0.8
 tree_method: None
 validate_parameters: None
 verbosity: 0

Holdout RMSE: 185.8695
```

Part C

Statistical Comparison: Dependent Variable `Spending` in Task (a) vs. Task (b)

Metric	Task (a): All Data	Task (b): Purchase = 1 Only	Difference
Mean	102.56	205.09	**+102.53 (doubled)**
Std Dev	186.75	220.77	**+34.02**
Variance	34,875.49	48,740.12	**+13,864.63**

Interpretation

- The **standard deviation and variance** are significantly higher in Task (b)
- This proves that:

> Predicting `Spending` becomes **more variable and challenging** when the dataset is restricted to actual purchasers

---

Conclusion

When we remove the many "non-purchase" records from the dataset (i.e., `Purchase = 0`), the task becomes much more difficult for predictive models. This is reflected in the **increased standard deviation and variance** of the target variable `Spending`. In the restricted dataset, consumers spend a **wider and more unpredictable** range of amounts, increasing noise and reducing model accuracy.

In [22]:
```python
# Load data
data = pd.read_csv("hw3data.csv")

# Full dataset stats
spending_all = data["Spending"]
```

```python
mean_all = spending_all.mean()
std_all = spending_all.std()
var_all = spending_all.var()

# Restricted dataset stats (Purchase = 1)
spending_restricted = df[df["Purchase"] == 1]["Spending"]
mean_restricted = spending_restricted.mean()
std_restricted = spending_restricted.std()
var_restricted = spending_restricted.var()

# Display results
print("=== Full Dataset (Task a) ===")
print(f"Mean:     {mean_all:.2f}")
print(f"Std Dev:  {std_all:.2f}")
print(f"Variance: {var_all:.2f}")

print("\n=== Restricted Dataset (Task b) ===")
print(f"Mean:     {mean_restricted:.2f}")
print(f"Std Dev:  {std_restricted:.2f}")
print(f"Variance: {var_restricted:.2f}")
```

```
=== Full Dataset (Task a) ===
Mean:     102.56
Std Dev:  186.75
Variance: 34875.49

=== Restricted Dataset (Task b) ===
Mean:     205.09
Std Dev:  220.77
Variance: 48740.12
```

Final Takeaways by Model

Model	Insight
XGBoost	Good performer in both tasks - handles complexity and generalizes well even when 0s are removed
NeuralNet	Still strong, but performance edge reduced when predicting true spend values only
GradientBoosting / RF	Reliable but less dominant without 0-heavy patterns
LinearRegression	Performed well on all data, but weak on actual purchasers due to inability to capture complex spending patterns
SVM / KNN / Trees	Sensitive to noise and lack of simple patterns - struggled more in Task (b)

Why All Models Perform Worse in Task (b)

- **Loss of Easy-to-Predict "0s"**
  In Task (a), many consumers had `Spending = 0`. Most models learned to predict **zero**, which lowered RMSE.

- **Higher Variability in Task (b)**
  Among `Purchase = 1`, the `Spending` values are spread out → more noise, harder regression.

- **Smaller Dataset in Task (b)**
  We are training on fewer rows, so models are more likely to overfit or underperform.

In [ ]: