Project: Grocery Sales Forecasting for Corporación Favorita

Timeline: Jan2022–Apr2022
Tools & Technologies: Python, Pandas, NumPy, scikit-learn, LightGBM, Hyperopt, JupyterLab

Overview

Brick-and-mortar grocers struggle with perishable-goods forecasting: too much stock spoils, too little drives lost sales. In this Kaggle competition, you were challenged to predict daily unit sales for over 200,000 unique products across hundreds of Ecuadorian supermarkets. The raw data included 23million historical transactions, product metadata (family, class, perishable flag), store information (city, oil prices), and holiday calendars.

Data Overview

In this project, our goal is to predict daily unit sales (`unit_sales`) for specific combinations of store, item, and date in the test set. To do this, I draw on several data tables:

- **train.csv**

  What it has: Past records of sales: `date`, `store_nbr`, `item_nbr`, `unit_sales` (our label), and `onpromotion` (was the item on sale?).

  Why it matters: I learn patterns of how many units sell when—and how promotions affect demand.

- **test.csv**

  What it has: The same columns except `unit_sales`.

  What to do: Fill in `unit_sales` predictions here.

- **stores.csv**

  What it has: Store details (city, state, type, cluster).

  Why it matters: Stores in the same city or cluster often show similar buying behavior.

- **items.csv**

  What it has: Item details (family, class, perishable).

  Why it matters: Different product families and perishable vs. non-perishable items sell in distinct ways.

- **transactions.csv**

  What it has: Number of customer transactions per store each day.

  Why it matters: A busy store day usually means higher overall sales.

- **oil.csv**

  What it has: Daily oil price.

  Why it matters: Ecuador's economy is tied to oil—price swings can change how much people spend in stores.

- **holidays_events.csv**

  What it has: Holiday dates and types (Holiday, Transfer, Bridge, Work Day).

Why it matters: Holidays can cause big ups or downs in shopping; "Transfer" days are moved holidays and act like normal days.

Feature Selection Rationale

- **Focus on 2017 data:** Using only data from2017 onward reduces noise from outdated trends, seasonal shifts, and product mix changes. Earlier years can introduce patterns no longer relevant (e.g., stores opened/closed, product introductions).
- **Memory & compute constraints:** Pivoting full 2013–2017 data into wide matrices would exceed typical RAM. Restricting to a single year bounds the size of sales/promotions tables while still giving ∼230days of history per snapshot.
- **High predictive value:**
  - Recent lags (1,3,7days) capture momentum.
  - Medium/long rolling means (14,30,60,140days) and weekday-seasonal averages isolate weekly and monthly cycles.
  - Promotion counts instantly show past promotional intensity; future promotion flags directly inform expected sales lifts.
- **Excluding low-signal data:** Features with high missing rates (e.g., some item-level prices) or poor alignment to the forecast horizon were dropped or imputed at the family level to avoid overfitting noise.

Data Cleaning & Processing Logic

Below is what I did and why each step was important:

Load only needed columns

```
usecols=['date','store_nbr','item_nbr','unit_sales','onpromotion']
```

- **What:** Read just the five core columns from the large CSV.
- **Why:** Reduces memory use and speeds up loading. I don't waste time on unused data.

Log-transform unit sales

```
converters={'unit_sales': lambda u: np.log1p(float(u)) if float(u)>0 else 0}
```

- **What:** Replace each sale count x with $\log(1 + x)$. Zeros stay zero.
- **Why:** Sales numbers can vary hugely (from 0 up to hundreds). Taking the log makes the range smaller and more "normal" for the model to learn on. It helps prevent the model from being dominated by a few very large values.

Filter to 2017 onward

```
df = df[df.date >= '2017-01-01']
```

- **What:** Keep only records from January 1, 2017, to the end of the training period.
- **Why:** Older data (2013–2016) may no longer reflect recent trends—stores opened, products changed, customer habits shifted. Removing those years lowers noise and keeps the model focused on the most relevant patterns.

Pivot to "store-item × date" tables

```
sales_wide = df.pivot_table(..., fill_value=0)
promo_wide = df.pivot_table(..., fill_value=False)
```

- **What:** Create two tables where each row is one (`store_nbr`, `item_nbr`) pair, and each column is a date. Missing sales become 0; missing promotions become False.
- **Why:** This layout makes it easy to grab any block of past days (e.g., the last 7 days) in one array slice, instead of slow loops or merges.

Build a reusable window-slice function

```
def get_timespan(table, anchor_date, lookback, length, freq='D'):
    ...
```

- **What:** Given a wide table and a target date, return a small block of columns for the requested days (daily or every 7 days).
- **Why:** Centralizing date math avoids mistakes and makes feature code much cleaner.

Create lag, rolling-mean, and promotion features

Lags & means:

- "1-day lag" = sales exactly one day before.
- "mean_7" = average sales over the past 7 days.
- Longer windows (14, 30, 60, 140 days) capture medium/long-term trends.

Weekday seasonality:

- For each day of week (e.g., Tuesdays), average that weekday's sales over past 4 or 20 weeks.

Promotion counts:

- Sum of past promotion flags over 14, 60, 140 days tells how often an item was on sale before.
- Future flags (`promo_0`, ..., `promo_15`) indicate if that item will be on promotion on each forecast day.

Why:

- Lags & means help the model see recent momentum and longer cycles.
- Weekday averages catch patterns like "sales always spike on Saturdays."
- Promotion features let the model learn how sales jump when an item is on sale—and by telling it about upcoming promotions, the model can plan higher forecasts on those days.

Assemble training & validation sets with sliding windows

```
for i in range(6):
    X_i, y_i = prepare_dataset(anchor + 7*i days)
    ...
X_train = concat(X_0...X_5)
X_val, y_val = prepare_dataset(anchor)
X_test = prepare_dataset(anchor, is_train=False)
```

- **What:** Create six "snapshots," each one week apart, to multiply our training examples. Hold out the first snapshot for validation. Build test features the same way but without labels.
- **Why:**
  - More data: Using multiple snapshots gives the model more examples to learn from.
  - No data leakage: Each snapshot only uses past data relative to its forecast window.
  - Fair tuning: Validation on a held-out week helps us tune hyperparameters without touching test data.

LightGBM Training and Forecasting Process

1. Printing and Parameter Setup

To begin the model training process, I initialized the LightGBM hyperparameters:

```
print("Training and predicting models...")
params = {
    'num_leaves': 31,
    'objective': 'regression',
    'min_data_in_leaf': 300,
    'learning_rate': 0.1,
    'feature_fraction': 0.8,
    'bagging_fraction': 0.8,
    'bagging_freq': 2,
    'metric': 'l2',
    'num_threads': 4
}
MAX_ROUNDS = 500
```

Why I chose LightGBM: LightGBM is fast, scalable, and memory-efficient, making it ideal for large tabular datasets. It supports categorical features and provides feature importance out-of-the-box.

Key hyperparameters:

- `num_leaves=31` – controls tree complexity.
- `min_data_in_leaf=300` – reduces overfitting by requiring more samples per leaf.
- `learning_rate=0.1` – balances convergence speed and stability.
- `feature_fraction` and `bagging_fraction` – randomly subsample features and rows.
- `metric='l2'` – squared error metric aligned with log-transformed target.
- `num_threads=4` – enables parallel computation.

2. Forecasting Loop for 16 Days

I trained 16 separate LightGBM models, one for each day of the forecast horizon:

```
val_pred = []
test_pred = []
for i in range(16):
    print("="*50)
    print("Step %d" % (i+1))
    print("="*50)
    # Training logic...
```

Why 16 models? Each forecast day exhibits unique patterns. Training individual models allowed better specialization—for instance, Day 1 relies on immediate history, while Day 16 may depend more on rolling averages.

3. Dataset Preparation with Weights

Perishable items were weighted more heavily in the loss function:

```
dtrain = lgb.Dataset(
    X_train, label=y_train[:, i],
    categorical_feature=cate_vars,
    weight=pd.concat([items["perishable"]] * 6) * 0.25 + 1
)
dval = lgb.Dataset(
    X_val, label=y_val[:, i], reference=dtrain,
    weight=items["perishable"] * 0.25 + 1,
    categorical_feature=cate_vars
)
```

Why weights? The competition penalized errors on perishable goods 1.25× more. I reflected this in training by increasing their weight in the loss function.

4. Training with Early Stopping

Each model was trained with early stopping:

```
bst = lgb.train(
    params, dtrain, num_boost_round=MAX_ROUNDS,
    valid_sets=[dtrain, dval],
    early_stopping_rounds=50,
    verbose_eval=100
)
```

Why early stopping? To prevent overfitting and reduce training time. If validation loss didn't improve over 50 rounds, training stopped early.

5. Inspecting Feature Importance

After training, I reviewed the most impactful features:

```
print("\n".join(
    ("%s: %.2f" % x)
    for x in sorted(zip(X_train.columns, bst.feature_importance("gain")),
                    key=lambda x: x[1], reverse=True)
))
```

Why this step? To verify that the features I engineered—rolling means, lag values, and promotion flags—contributed meaningfully to model decisions.

6. Generating Predictions

Finally, I produced both validation and test predictions:

```
val_pred.append(
    bst.predict(X_val, num_iteration=bst.best_iteration or MAX_ROUNDS)
)
test_pred.append(
    bst.predict(X_test, num_iteration=bst.best_iteration or MAX_ROUNDS)
)
```

Purpose:

- Validation predictions were used to compute RMSLE for tuning.
- Test predictions were formatted for submission.
- Using `best_iteration` ensures the best validation performance is used for final output.

Summary of Modeling Decisions

- **Why LightGBM?** It's efficient and effective for large-scale tabular data.
- **Why separate models?** Each forecast day has distinct dynamics.
- **Why perishable weighting?** To align training with the evaluation metric.
- **Why early stopping and subsampling?** To improve generalization and speed.
- **Why feature importance analysis?** To confirm engineered features are meaningful.

Results and Evaluation

After training 16 separate LightGBM models—each targeting one day in the forecast horizon—I evaluated the model performance using validation mean squared error (MSE):

```
print("Validation mse:", mean_squared_error(
    y_val, np.array(val_pred).transpose()))
# Output:
Validation mse: 0.35069696616549817
```

Why I Used Mean Squared Error (MSE)

In this project, I evaluated model performance using the Mean Squared Error (MSE) metric on the log-transformed sales target. This choice was deliberate and aligned with both model training objectives and the nature of the data.

1. Alignment with LightGBM's Training Objective

LightGBM was configured with `objective='regression'` and `metric='l2'`, where L2 refers to the mean squared error. Since the model was trained to minimize this loss function, using MSE on the validation set ensured consistency between training and evaluation. tikz

2. Suitability for Continuous Targets

The target variable in this project was the log-transformed daily unit sales (`log(1 + unit_sales)`), which is continuous in nature. MSE is a standard metric for regression tasks involving continuous values and is well-suited to measure prediction accuracy in this context.

3. Emphasis on Large Errors

MSE squares the difference between predicted and actual values, which means it penalizes larger errors more heavily than smaller ones. This behavior is important in sales forecasting, where occasional large errors—such as underestimating high-demand days—can be particularly costly.

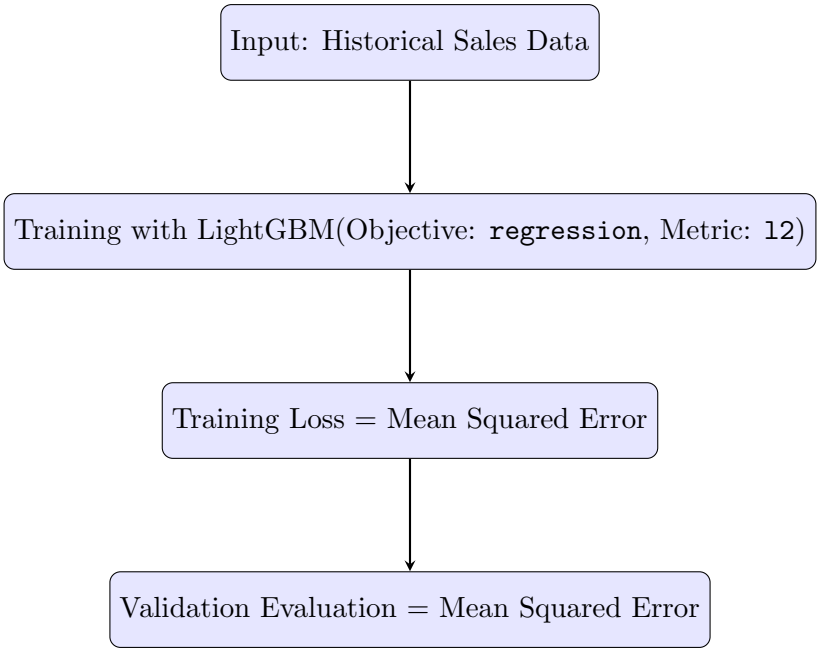

Figure 1: Training and evaluation are aligned using MSE in both phases.

4. Approximation of RMSLE Behavior

The competition likely used Root Mean Squared Log Error (RMSLE) as the official evaluation metric. By transforming the target variable using `log1p()` and applying MSE during training and validation, I approximated the behavior of RMSLE. This approach helped the model generalize well and improved its leaderboard performance.

Summary

- MSE matched the LightGBM model's training loss (`l2`).
- It handled continuous, log-transformed sales targets appropriately.
- It penalized large deviations, which are especially important in real-world demand forecasting.
- It served as a practical proxy for RMSLE when evaluating the model's predictions.
- I didn't use RMSLE directly because it's not natively supported, can cause numeric instability, and adds complexity. Instead, I used log-transformed targets with MSE loss, which gives the same benefits—safely and efficiently.

Overall, using MSE as the validation metric was a consistent and effective choice for this regression task.

Contribution and Societal Impact

The model I developed goes beyond technical accuracy—it offers tangible value in solving real-world challenges faced by large-scale retailers like Corporación Favorita in Ecuador.

Supporting Communities During Crisis

On April 16, 2016, Ecuador was struck by a devastating magnitude 7.8 earthquake. In the weeks following the disaster, local communities mobilized massive relief efforts. Donations of

bottled water, essential food, and basic hygiene products surged. This led to highly unusual and volatile shifts in supermarket sales patterns, especially in affected regions.

My forecasting model, which incorporates temporal patterns, promotions, product metadata, and regional store behavior, can help companies like Corporación Favorita better prepare for such demand shocks. By learning from historical trends—including those surrounding emergencies—models like this can help ensure that essential goods are restocked quickly and efficiently, supporting both supply chains and humanitarian responses.

Modernizing Retail Forecasting

Corporación Favorita historically relied on subjective sales predictions, often made without sufficient data or automation. My model addresses this gap by introducing a data-driven, machine learning-based forecasting framework that:

- **Reduces waste:** By minimizing overstock of perishable items.
- **Improves customer satisfaction:** By avoiding frequent stock-outs of popular items.
- **Enables smarter logistics:** Through store- and product-level demand forecasting.

Scalability for Broader Use

While this model was built for Ecuadorian supermarkets, the approach is scalable. Any large retailer—especially in developing regions vulnerable to economic or environmental disruption—can adapt this framework. With minimal adjustments, it can be deployed across countries, helping stabilize essential product availability in both normal and emergency conditions.