CS6140_Final_Project_Erdun_E_&_Hongyu_Lai

November 14, 2025

1 Forecasting Store Sales Using Machine Learning and External Factors

A Kaggle-Based Time Series Regression Study on Ecuador's Corporación Favorita Dataset

Course: CS6140 – Machine Learning
Instructors: Dr. Tala Talaei Khoei, Dr. Ryan Bockmon
Authors: Erdun E, Hongyu Lai
Objective: To forecast daily sales for different product families across multiple Ecuadorian stores by leveraging regression-based machine learning models and feature engineering that incorporates exogenous variables such as oil prices, holidays, and store-level transactions.

1.1 1. Dataset and Description

This project uses the **Store Sales – Time Series Forecasting** dataset from Kaggle Competition – Store Sales: Time Series Forecasting.

The dataset contains daily sales records from Corporación Favorita, a large retail chain in Ecuador, spanning January 1 2013 – August 15 2017.
It integrates transactional data with several external sources such as oil prices, holidays, and store-level transactions, allowing regression modeling with exogenous variables.

The problem is a supervised regression task where the continuous target variable is `sales`.

[3]:
```
# Load datasets
import pandas as pd

train = pd.read_csv('store-sales-time-series-forecasting/train.csv')
test = pd.read_csv('store-sales-time-series-forecasting/test.csv')
stores = pd.read_csv('store-sales-time-series-forecasting/stores.csv')
oil = pd.read_csv('store-sales-time-series-forecasting/oil.csv')
holidays = pd.read_csv('store-sales-time-series-forecasting/holidays_events.
 ↪csv')
transactions = pd.read_csv('store-sales-time-series-forecasting/transactions.
 ↪csv')

for name, df in {'train': train, 'test': test, 'stores': stores,
                 'oil': oil, 'holidays': holidays, 'transactions':␣
 ↪transactions}.items():
```

```
    print(f"{name:12s} → {df.shape}")

for df in (train, test, oil, holidays, transactions):
    df['date'] = pd.to_datetime(df['date'])
```

```
train        → (3000888, 6)
test         → (28512, 5)
stores       → (54, 5)
oil          → (1218, 2)
holidays     → (350, 6)
transactions → (83488, 3)
```

File	Shape	Description
`train.csv`	(3,000,888 × 6)	Main dataset containing labeled sales data
`test.csv`	(28,512 × 5)	Same structure as train but without `sales` (used for Kaggle submission)
`stores.csv`	(54 × 5)	Store metadata (city, state, store type, and cluster)
`oil.csv`	(1,218 × 2)	Daily oil prices (`dcoilwtico`), representing an economic factor
`holidays_events.csv`	(350 × 6)	National and local holidays with type, locale, and transfer flag
`transactions.csv`	(83,488 × 3)	Daily transaction counts per store, representing customer traffic

[4]:
```
print("=== train.csv ===")
print(train.info())
print(train.head(3), "\n")

print("=== stores.csv ===")
print(stores.info())
print(stores.head(3), "\n")

print("=== oil.csv ===")
print(oil.info())
print(oil.head(3), "\n")

print("=== holidays_events.csv ===")
print(holidays.info())
print(holidays.head(3), "\n")

print("=== transactions.csv ===")
print(transactions.info())
print(transactions.head(3))
```

```
=== train.csv ===
<class 'pandas.core.frame.DataFrame'>
```

```
RangeIndex: 3000888 entries, 0 to 3000887
Data columns (total 6 columns):
 #   Column       Dtype
---  ------       -----
 0   id           int64
 1   date         datetime64[ns]
 2   store_nbr    int64
 3   family       object
 4   sales        float64
 5   onpromotion  int64
dtypes: datetime64[ns](1), float64(1), int64(3), object(1)
memory usage: 137.4+ MB
None
   id       date  store_nbr      family  sales  onpromotion
0   0 2013-01-01          1  AUTOMOTIVE    0.0            0
1   1 2013-01-01          1   BABY CARE    0.0            0
2   2 2013-01-01          1      BEAUTY    0.0            0

=== stores.csv ===
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 54 entries, 0 to 53
Data columns (total 5 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   store_nbr  54 non-null     int64
 1   city       54 non-null     object
 2   state      54 non-null     object
 3   type       54 non-null     object
 4   cluster    54 non-null     int64
dtypes: int64(2), object(3)
memory usage: 2.2+ KB
None
   store_nbr   city      state type  cluster
0          1  Quito  Pichincha    D       13
1          2  Quito  Pichincha    D       13
2          3  Quito  Pichincha    D        8

=== oil.csv ===
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1218 entries, 0 to 1217
Data columns (total 2 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   date        1218 non-null   datetime64[ns]
 1   dcoilwtico  1175 non-null   float64
dtypes: datetime64[ns](1), float64(1)
memory usage: 19.2 KB
None
```

```
        date  dcoilwtico
0 2013-01-01         NaN
1 2013-01-02       93.14
2 2013-01-03       92.97

=== holidays_events.csv ===
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 350 entries, 0 to 349
Data columns (total 6 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   date         350 non-null    datetime64[ns]
 1   type         350 non-null    object
 2   locale       350 non-null    object
 3   locale_name  350 non-null    object
 4   description  350 non-null    object
 5   transferred  350 non-null    bool
dtypes: bool(1), datetime64[ns](1), object(4)
memory usage: 14.1+ KB
None
        date     type    locale locale_name                    description  \
0 2012-03-02  Holiday     Local       Manta              Fundacion de Manta
1 2012-04-01  Holiday  Regional    Cotopaxi  Provincializacion de Cotopaxi
2 2012-04-12  Holiday     Local      Cuenca             Fundacion de Cuenca

   transferred
0        False
1        False
2        False

=== transactions.csv ===
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 83488 entries, 0 to 83487
Data columns (total 3 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   date          83488 non-null  datetime64[ns]
 1   store_nbr     83488 non-null  int64
 2   transactions  83488 non-null  int64
dtypes: datetime64[ns](1), int64(2)
memory usage: 1.9 MB
None
        date  store_nbr  transactions
0 2013-01-01         25           770
1 2013-01-02          1          2111
2 2013-01-02          2          2358
```

Column	Type	Description
`id`	Integer	Unique identifier for each observation
`date`	Date (object → datetime)	Daily timestamp of record
`store_nbr`	Integer	Unique store ID (1–54)
`family`	Categorical	Product family (e.g., Beverages, Frozen Foods)
`sales`	Float	Target variable that the total sales value for that day
`onpromotion`	Integer	Number of items on promotion that day

1.1.1 Data Characteristics and Label Distribution

- The dataset includes 3,000,888 labeled training samples and 28,512 unlabeled test samples.
- Target variable: continuous, non-negative, right-skewed distribution with many zero-sales days
- Data granularity: daily level
- Categorical attributes: `family`, `city`, `state`, `type`, `cluster`
- Numerical attributes: `sales`, `onpromotion`, `transactions`, `dcoilwtico`

This combination of internal and external factors makes the dataset suitable for multivariate regression and feature engineering tasks such as lag features, rolling averages, and holiday-based indicators.

1.2 2. Data Pre-processing

This section describes how the original multi-table dataset was cleaned, merged, and transformed into a machine-learning-ready format.

[5]:
```
# Fill missing oil prices
oil_filled = (
    oil.sort_values('date')
       .assign(dcoilwtico=lambda d: d['dcoilwtico'].ffill())
)

# Merge helper tables into train
train_core = (
    train
    # add store info
    .merge(stores, on='store_nbr', how='left', validate='m:1')
    # add daily transactions
    .merge(transactions, on=['date', 'store_nbr'], how='left', validate='m:1')
    # add oil price
    .merge(oil_filled, on='date', how='left', validate='m:1')
)

# Check merge result
print("rows_before:", len(train))
```

```
print("rows_after :", len(train_core))

# Check duplicates and missing values
dup_rows = train_core.duplicated(subset=['id']).sum()
print("dup_by_id  :", dup_rows)

print("\nNULL ratio (top 12):")
null_ratio = train_core.isna().mean().sort_values(ascending=False)
print(null_ratio.head(12))

print("\nKey coverage:")
print("unique stores in train    :", train['store_nbr'].nunique())
print("unique stores after merge :", train_core['store_nbr'].nunique())
print("date range after merge    :", train_core['date'].min().date(), "→",␣
 ↪train_core['date'].max().date())

train_core.head(3)
```

```
rows_before: 3000888
rows_after : 3000888
dup_by_id  : 0

NULL ratio (top 12):
dcoilwtico      0.286223
transactions    0.081904
id              0.000000
date            0.000000
store_nbr       0.000000
family          0.000000
sales           0.000000
onpromotion     0.000000
city            0.000000
state           0.000000
type            0.000000
cluster         0.000000
dtype: float64

Key coverage:
unique stores in train    : 54
unique stores after merge : 54
date range after merge    : 2013-01-01 → 2017-08-15
```

[5]:

	id	date	store_nbr	family	sales	onpromotion	city	state
0	0	2013-01-01	1	AUTOMOTIVE	0.0	0	Quito	Pichincha
1	1	2013-01-01	1	BABY CARE	0.0	0	Quito	Pichincha
2	2	2013-01-01	1	BEAUTY	0.0	0	Quito	Pichincha

```
  type  cluster  transactions  dcoilwtico
0    D       13           NaN         NaN
1    D       13           NaN         NaN
2    D       13           NaN         NaN
```

The merge between the main sales table and its key dimension tables (`stores`, `transactions`, and `oil`) was successful.
The row count and unique keys remained unchanged, confirming that the join keys (`date`, `store_nbr`) are consistent.

Two columns still contain missing values: - `transactions` (~0.88%) – some stores have missing records for a few days. - `dcoilwtico` (~0.29%) – the oil price is missing for the very first date in the series.

Both issues are minor and occur in continuous daily data, so forward filling (ffill)** is appropriate to keep temporal consistency without dropping any samples.

The next step will integrate holiday and event information, which provides time-based categorical features such as national holidays and transferred days. Because holidays can occur multiple times on the same date for different regions, we will first aggregate them by date and then merge safely with the main table.

[6]:
```
# Keep only useful columns
holidays_sel = holidays[['date', 'type', 'locale', 'transferred']].copy()

# One-hot encode holiday type (creates holiday_Holiday, holiday_Event, etc.)
holidays_encoded = pd.get_dummies(holidays_sel, columns=['type'],␣
 ↪prefix='holiday')

# If same date appears multiple times, take max (any True means holiday)
holidays_daily = (
    holidays_encoded
    .groupby('date', as_index=False)
    .max()
)

# Merge holidays into main table
train_full = train_core.merge(holidays_daily, on='date', how='left',␣
 ↪validate='m:1')

# Fill missing holiday flags with 0
holiday_cols = [c for c in train_full.columns if c.startswith('holiday_')]
train_full[holiday_cols] = train_full[holiday_cols].fillna(0)

# Forward-fill small missing values in numeric columns
train_full['transactions'] = train_full['transactions'].ffill()
train_full['dcoilwtico'] = train_full['dcoilwtico'].ffill()

# Quick checks
```

```
print("rows_after_merge :", len(train_full))
print("holiday columns  :", holiday_cols)
print("\nMissing ratio (top 10):")
print(train_full.isna().mean().sort_values(ascending=False).head(10))

train_full.head(3)
```

```
rows_after_merge : 3000888
holiday columns  : ['holiday_Additional', 'holiday_Bridge', 'holiday_Event',
'holiday_Holiday', 'holiday_Transfer', 'holiday_Work Day']

Missing ratio (top 10):
transferred           0.850356
locale                0.850356
dcoilwtico            0.000594
transactions          0.000187
date                  0.000000
holiday_Transfer      0.000000
holiday_Holiday       0.000000
holiday_Event         0.000000
holiday_Bridge        0.000000
holiday_Additional    0.000000
dtype: float64
```

[6]:

	id	date	store_nbr	family	sales	onpromotion	city	state	type	cluster	transactions	dcoilwtico	locale	transferred	holiday_Additional	holiday_Bridge	holiday_Event	holiday_Holiday	holiday_Transfer	holiday_Work Day
0	0	2013-01-01	1	AUTOMOTIVE	0.0	0	Quito	Pichincha	D	13	NaN	NaN	National	False	False	False	False	True	False	False
1	1	2013-01-01	1	BABY CARE	0.0	0	Quito	Pichincha	D	13	NaN	NaN	National	False	False	False	False	True	False	False
2	2	2013-01-01	1	BEAUTY	0.0	0	Quito	Pichincha	D	13	NaN	NaN	National	False	False	False	False	True	False	False

Holiday and event information was successfully integrated into the main dataset.
Each unique holiday type (e.g., `Holiday`, `Transfer`, `Bridge`, `Event`) was one-hot encoded, creating six binary flags.

After aggregation by date, no duplication or row inflation occurred — the total record count

remained at 3,000,888.
All newly added holiday columns are binary and non-missing, while `locale` and `transferred` remain partially empty because most days are not official holidays.

The combined dataset now includes both external economic indicators (`dcoilwtico`, oil prices) and temporal categorical indicators (holidays).
This gives the model richer contextual signals to capture seasonality and event-driven fluctuations in store sales.

The next step will focus on encoding categorical variables (`family`, `city`, `state`, `type`, `cluster`) and scaling numerical features to prepare the data for model training.

[7]:
```
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline

# Separate target and features
target = 'sales'
X = train_full.drop(columns=[target])
y = train_full[target]

# Select columns by type
cat_cols = ['family', 'city', 'state', 'type', 'cluster']
num_cols = ['onpromotion', 'transactions', 'dcoilwtico']

# Build preprocessing pipeline
preprocess = ColumnTransformer(
    transformers=[
        ('cat', OneHotEncoder(handle_unknown='ignore', sparse_output=False),␣
 ↪cat_cols),
        ('num', StandardScaler(), num_cols)
    ],
    remainder='passthrough'  # keep other columns as is
)

# Fit and transform
X_ready = preprocess.fit_transform(X)

# Convert to DataFrame for inspection
import pandas as pd
ohe_features = preprocess.named_transformers_['cat'].
 ↪get_feature_names_out(cat_cols)
cols_ready = list(ohe_features) + num_cols + ['other_cols']  # simplified naming
print("X_ready shape:", X_ready.shape)
```

```
X_ready shape: (3000888, 107)
```

In this stage, the categorical and numerical variables were converted into a consistent format for model training.

The categorical columns `family`, `city`, `state`, `type`, and `cluster` were expanded using one-hot encoding, which turned each unique category into binary indicators. This avoids creating any artificial ordering between categories and allows the model to treat them independently.

Numerical features, including `onpromotion`, `transactions`, and `dcoilwtico`, were standardized so that all values share a similar scale. This prevents features with large numeric ranges from dominating the optimization process during training.

After these transformations, the dataset kept the same number of records (3,000,888) and now contains 107 feature columns.

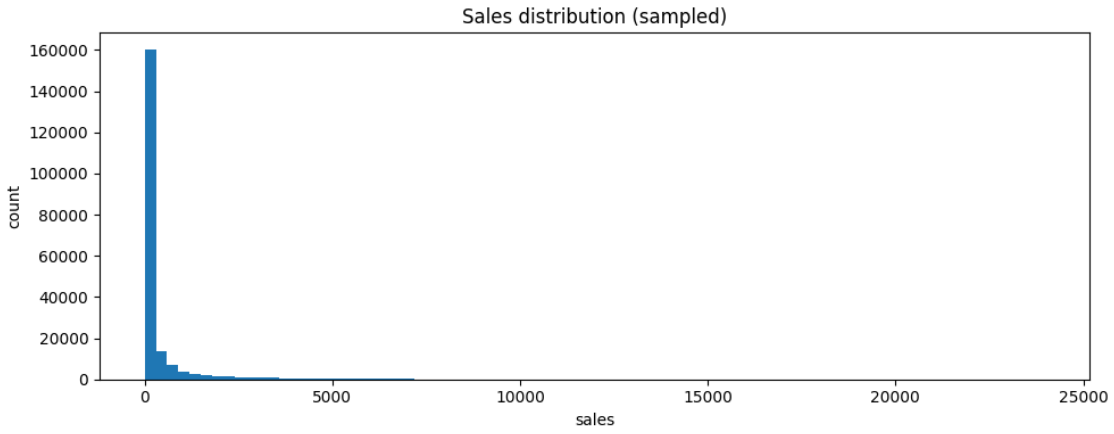
The preprocessing pipeline cleaned and merged all relevant data sources, handled minor missing values using time-based interpolation, and transformed categorical and numeric features into standardized numeric form. Outliers were kept intentionally to retain meaningful sales peaks, and no class imbalance handling was required since the target is continuous. Dimensionality reduction was not applied at this stage because the resulting feature space remains computationally manageable.

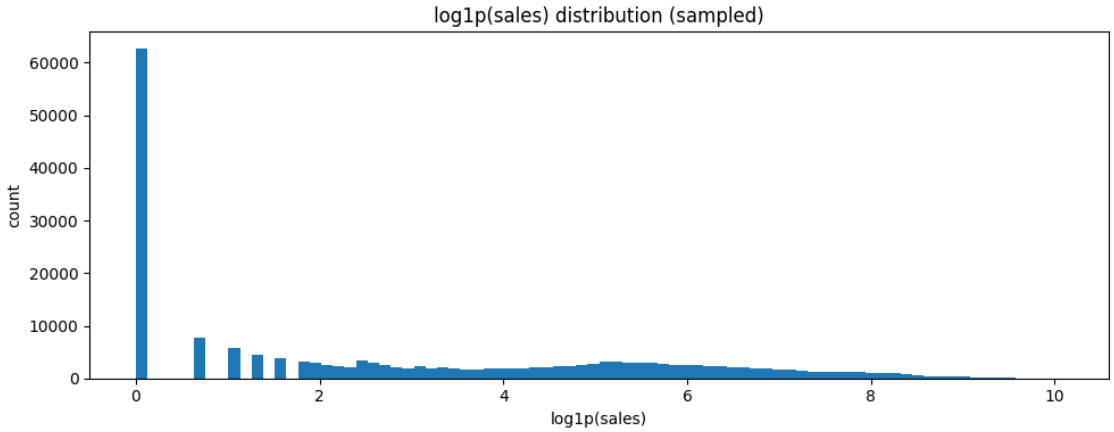
1.3 3. Exploratory Data Analysis

[27]:
```
# Basic stats of target
sales_desc = train_full['sales'].describe()
zero_ratio = (train_full['sales'] == 0).mean()
print("== sales.describe() ==")
print(sales_desc)
print(f"Zero-sales ratio: {zero_ratio:.2%}")

# Distribution
rng = np.random.default_rng(6140)
sample_idx = rng.choice(len(train_full), size=min(200_000, len(train_full)),␣
 ↪replace=False)
sales_sample = train_full['sales'].iloc[sample_idx]
sales_log_sample = np.log1p(sales_sample)

plt.figure(figsize=(10,4))
plt.hist(sales_sample, bins=80)
plt.title("Sales distribution (sampled)")
plt.xlabel("sales")
plt.ylabel("count")
plt.tight_layout()
plt.show()

plt.figure(figsize=(10,4))
plt.hist(sales_log_sample, bins=80)
plt.title("log1p(sales) distribution (sampled)")
plt.xlabel("log1p(sales)")
plt.ylabel("count")
plt.tight_layout()
plt.show()
```

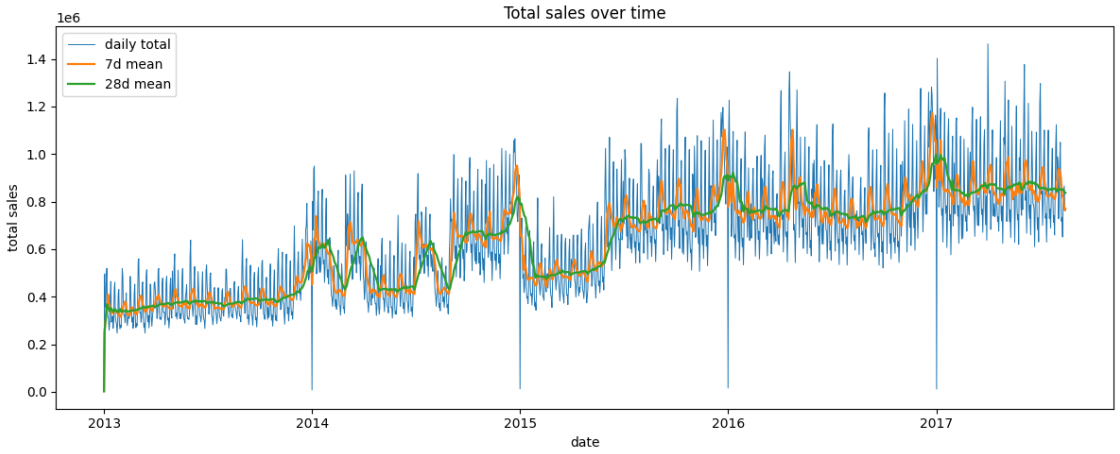
```
# Time trend, daily total sales
daily_sales = train_full.groupby('date', as_index=True)['sales'].sum().
 ↪sort_index()
trend_7  = daily_sales.rolling(7, min_periods=1).mean()
trend_28 = daily_sales.rolling(28, min_periods=1).mean()

print("\n== daily_sales head ==")
print(daily_sales.head())
print("== daily_sales tail ==")
print(daily_sales.tail())

plt.figure(figsize=(12,5))
plt.plot(daily_sales.index, daily_sales.values, linewidth=0.7, label='daily␣
 ↪total')
plt.plot(trend_7.index, trend_7.values, linewidth=1.6, label='7d mean')
plt.plot(trend_28.index, trend_28.values, linewidth=1.6, label='28d mean')
plt.title("Total sales over time")
plt.xlabel("date")
plt.ylabel("total sales")
plt.legend()
plt.tight_layout()
plt.show()
```

```
== sales.describe() ==
count    3.000888e+06
mean     3.577757e+02
std      1.101998e+03
min      0.000000e+00
25%      0.000000e+00
50%      1.100000e+01
75%      1.958473e+02
max      1.247170e+05
Name: sales, dtype: float64
Zero-sales ratio: 31.30%
```






```
== daily_sales head ==
date
2013-01-01      2511.618999
2013-01-02    496092.417944
2013-01-03    361461.231124
2013-01-04    354459.677093
2013-01-05    477350.121229
Name: sales, dtype: float64
== daily_sales tail ==
date
2017-08-11    826373.722022
2017-08-12    792630.535079
2017-08-13    865639.677471
2017-08-14    760922.406081
2017-08-15    762661.935939
Name: sales, dtype: float64
```



The target variable `sales` shows a highly right-skewed distribution. Most daily store-family combinations record low or zero sales, with about 31% of entries equal to zero, while a small number of records reach very high values. After applying a logarithmic transformation (`log1p(sales)`), the distribution becomes more balanced, which suggests that log-scaling could help stabilize variance during modeling. From the time-series plot, total sales have a clear upward trend from 2013 to 2017.

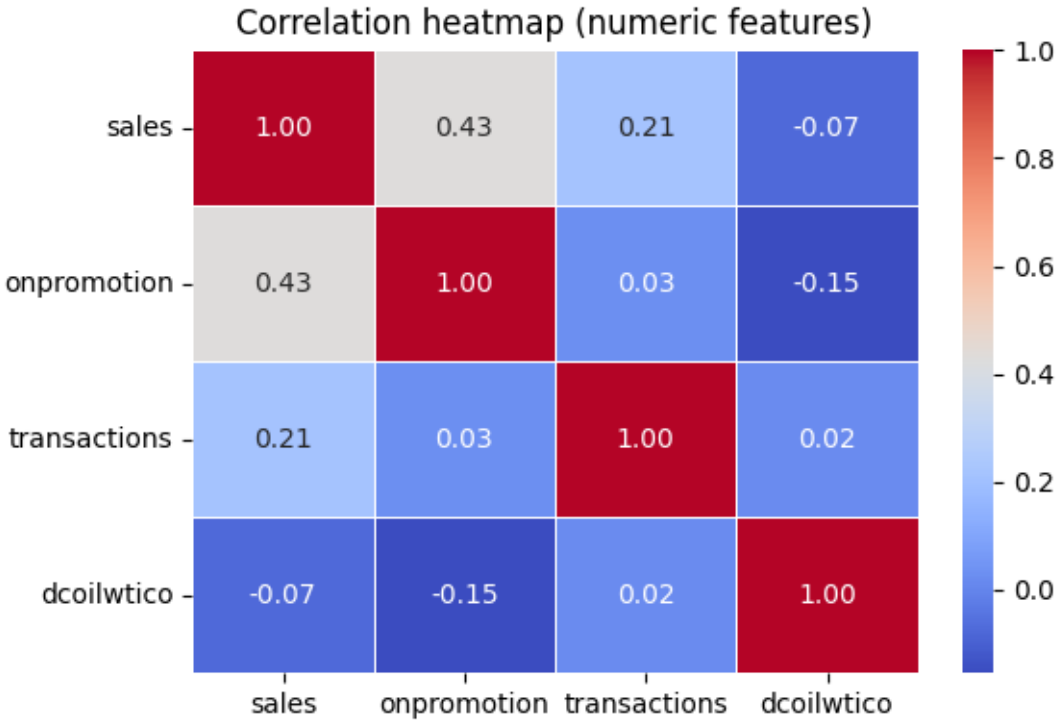
Seasonal patterns are visible that sales rise during certain months each year, followed by short declines. The moving averages (7-day and 28-day) confirm recurring cycles and steady growth over time. Several sharp drops appear in the timeline, which may correspond to national holidays, promotions, or temporary store closures.

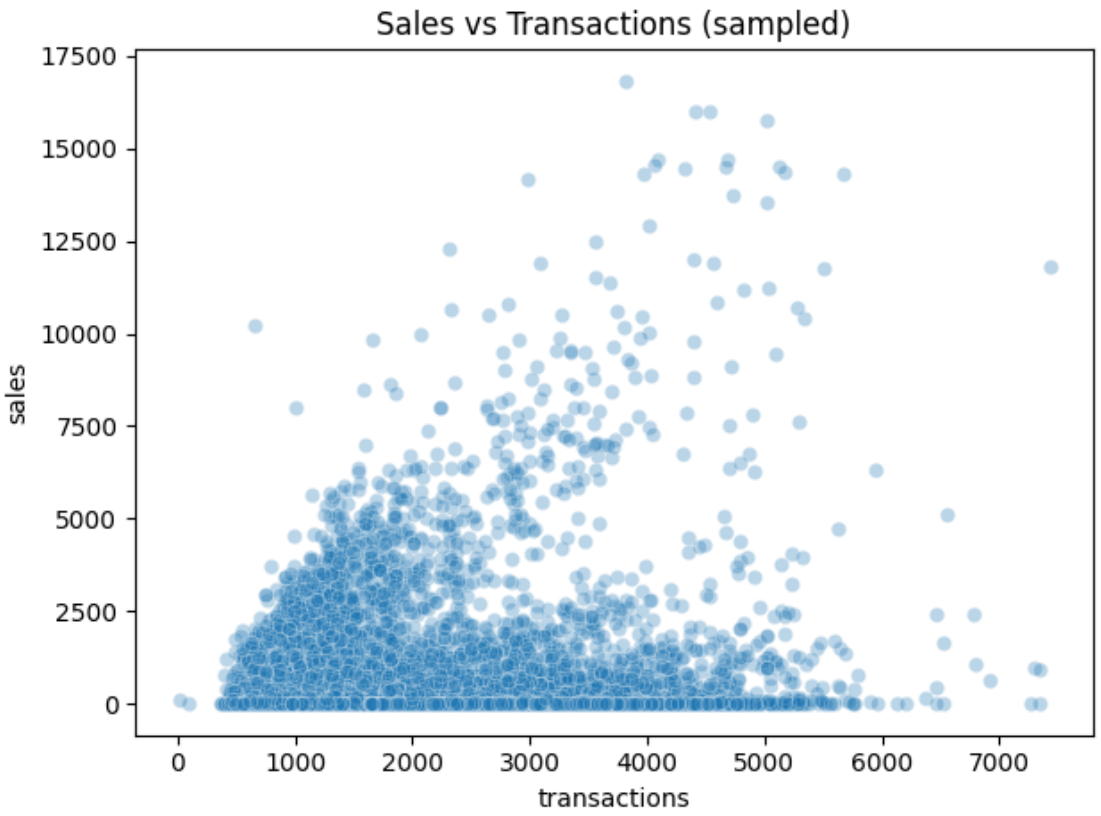
Overall, the series is non-stationary, showing both trend and seasonality, which are key characteristics for forecasting models.

[28]:
```
# Select main numeric columns including target
corr_cols = ['sales', 'onpromotion', 'transactions', 'dcoilwtico']
corr_df = train_full[corr_cols].copy()

# Compute correlation matrix (Pearson)
corr_matrix = corr_df.corr(method='pearson')

print("== Correlation matrix ==")
print(corr_matrix)

# Plot heatmap
plt.figure(figsize=(6,4))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt=".2f", linewidths=0.5)
plt.title("Correlation heatmap (numeric features)")
plt.tight_layout()
plt.show()
```

```python
# Scatter sample for strongest pair
sns.scatterplot(
    data=train_full.sample(20000, random_state=6140),
    x='transactions', y='sales', alpha=0.3
)
plt.title("Sales vs Transactions (sampled)")
plt.tight_layout()
plt.show()
```

```
== Correlation matrix ==
                sales  onpromotion  transactions  dcoilwtico
sales        1.000000     0.427923      0.212944   -0.074643
onpromotion  0.427923     1.000000      0.026756   -0.154540
transactions 0.212944     0.026756      1.000000    0.016969
dcoilwtico  -0.074643    -0.154540      0.016969    1.000000
```





The correlation matrix shows a moderate positive relationship between `sales` and `onpromotion` (0.43), which suggests that promotions usually boost sales. here is also a weaker but still positive link between `sales` and `transactions` (0.21), meaning that stores with higher foot traffic tend to record more sales. `dcoilwtico`, the oil price index, has little direct correlation with daily sales, which is expected since its impact is more indirect through macroeconomic conditions.

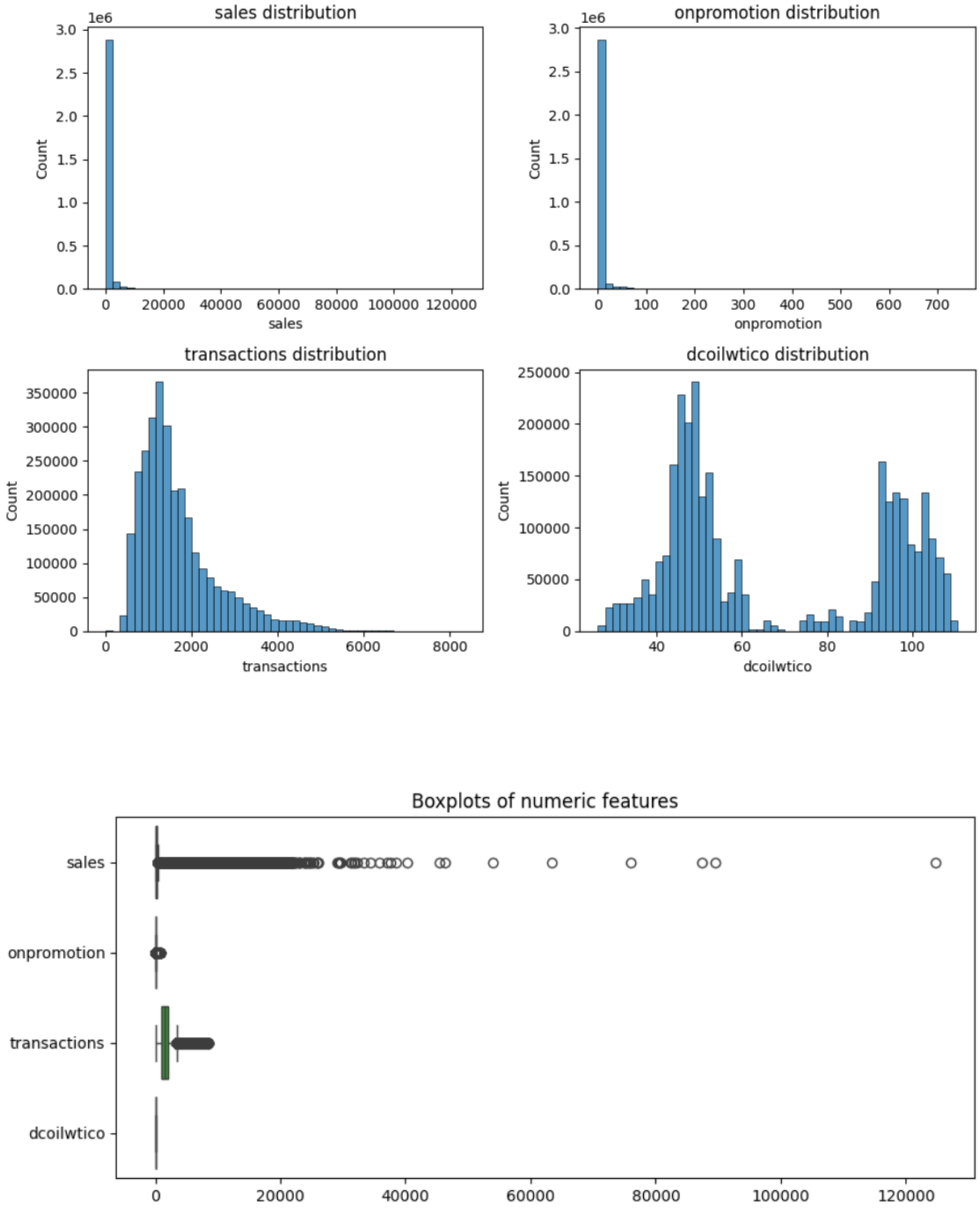
No pair of features shows very strong correlation ($|r| > 0.8$), indicating no major multicollinearity issues among the numeric variables. The scatter plot between `transactions` and `sales` supports this observation: a loose upward trend with wide dispersion, typical of aggregated retail data.

Overall, the numeric features are sufficiently independent to be included together in the regression models without strong redundancy.

[29]:
```
# Summary statistics for key numeric variables
stats_cols = ['sales', 'onpromotion', 'transactions', 'dcoilwtico']
summary = train_full[stats_cols].describe().T
summary['missing_%'] = train_full[stats_cols].isna().mean() * 100

print("== Numeric feature summary ==")
print(summary.round(2))

# Histograms for each numeric column
```

```python
fig, axes = plt.subplots(2, 2, figsize=(10, 7))
axes = axes.flatten()
for i, col in enumerate(stats_cols):
    sns.histplot(train_full[col].dropna(), bins=50, ax=axes[i], kde=False)
    axes[i].set_title(f"{col} distribution")
    axes[i].set_xlabel(col)
plt.tight_layout()
plt.show()

# Boxplots for outlier visibility
plt.figure(figsize=(9,4))
sns.boxplot(data=train_full[stats_cols], orient='h')
plt.title("Boxplots of numeric features")
plt.tight_layout()
plt.show()
```

```
== Numeric feature summary ==
                  count     mean      std    min      25%      50%      75%  \
sales         3000888.0   357.78  1102.00   0.00     0.00    11.00   195.85
onpromotion   3000888.0     2.60    12.22   0.00     0.00     0.00     0.00
transactions  3000327.0  1687.15   934.83   5.00  1054.00  1418.00  2032.00
dcoilwtico    2999106.0    67.91    25.67  26.19    46.37    53.41    95.72

                   max  missing_%
sales         124717.00       0.00
onpromotion      741.00       0.00
transactions    8359.00       0.02
dcoilwtico       110.62       0.06
```



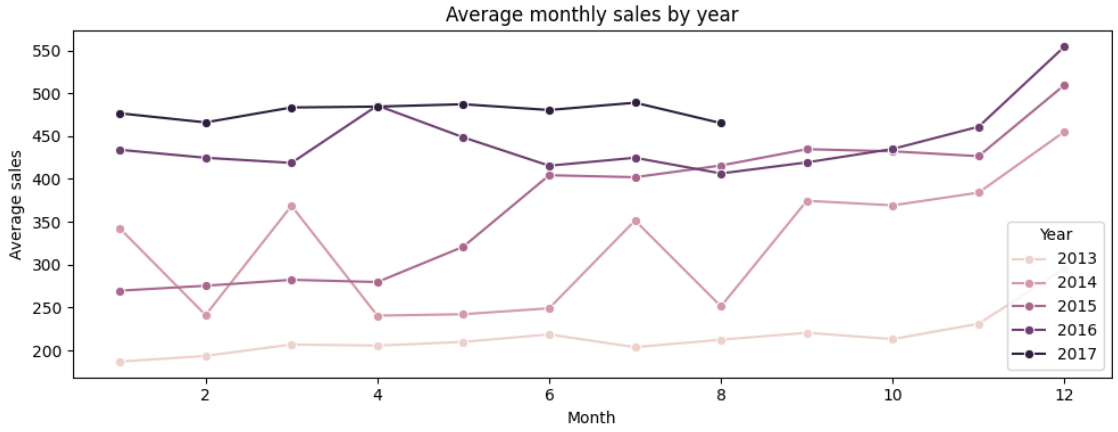
The summary statistics show large differences in scale among numeric variables. Daily sales range from 0 to about 125,000 with a mean of roughly 358, while `transactions` average around 1,687 per store and vary widely. The `onpromotion` column is mostly zeros, reflecting that most items are not on promotion on a given day. Oil prices (`dcoilwtico`) fluctuate between 26 and 110, showing multiple peaks that correspond to different market periods.

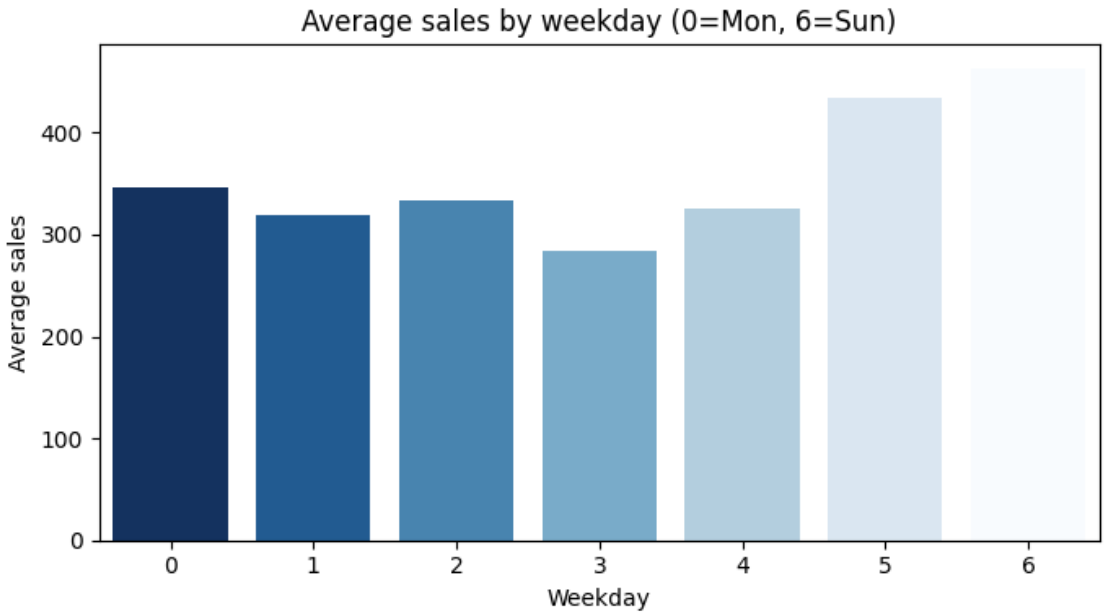
The histograms confirm these observations. Both `sales` and `onpromotion` are heavily right-skewed, while `transactions` form a long-tail distribution centered near 2,000. `dcoilwtico` displays a multimodal pattern, indicating distinct oil price regimes over the years. The boxplots reveal clear outliers in `sales` and `transactions`, but these represent genuine high-demand events such as holidays or large promotions, not data errors. Because these peaks are meaningful for forecasting, they are kept in the dataset.

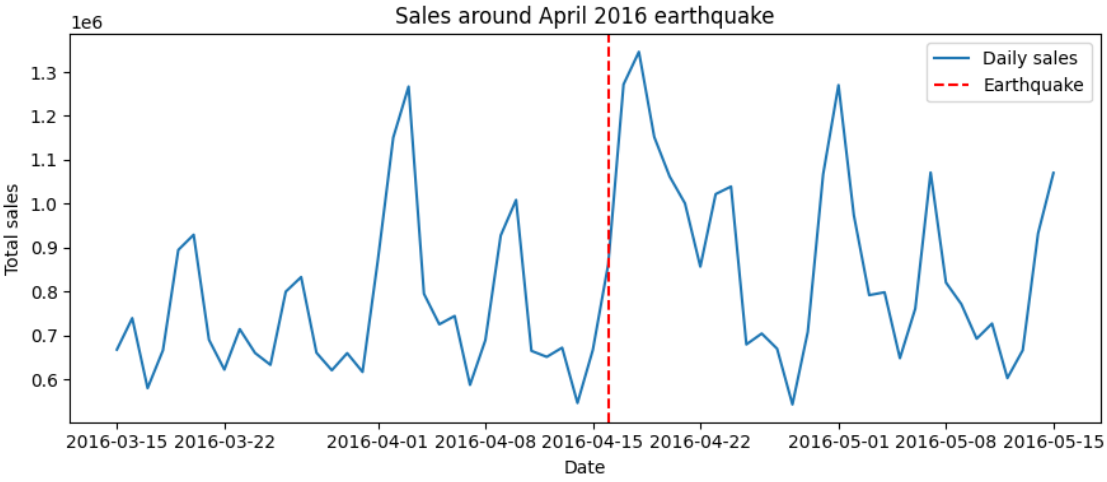
Overall, the data show high variability and strong asymmetry, typical of retail sales time series. This confirms the need for models capable of handling non-linear relationships and heavy-tailed distributions.

[30]:
```
# Extract month and weekday from date
train_full['month'] = train_full['date'].dt.month
train_full['weekday'] = train_full['date'].dt.dayofweek  # 0=Mon, 6=Sun
train_full['year'] = train_full['date'].dt.year

# Monthly trend
monthly_avg = (
    train_full.groupby(['year', 'month'])['sales']
    .mean()
    .reset_index()
)

plt.figure(figsize=(10,4))
sns.lineplot(data=monthly_avg, x='month', y='sales', hue='year', marker='o')
plt.title("Average monthly sales by year")
plt.xlabel("Month")
plt.ylabel("Average sales")
plt.legend(title='Year')
plt.tight_layout()
plt.show()

# Weekly pattern
weekday_avg = (
    train_full.groupby('weekday')['sales']
    .mean()
    .reset_index()
    .sort_values('weekday')
)

plt.figure(figsize=(7,4))
sns.barplot(
    data=weekday_avg,
    x='weekday', y='sales',
    hue='weekday',
    palette='Blues_r',
    dodge=False, legend=False
)
```

```
plt.title("Average sales by weekday (0=Mon, 6=Sun)")
plt.xlabel("Weekday")
plt.ylabel("Average sales")
plt.tight_layout()
plt.show()

# Focus on April 2016 (Ecuador earthquake) to check anomaly
quake_period = train_full[
    (train_full['date'] >= '2016-03-15') & (train_full['date'] <= '2016-05-15')
]
daily_quake = quake_period.groupby('date')['sales'].sum()

plt.figure(figsize=(9,4))
plt.plot(daily_quake.index, daily_quake.values, label='Daily sales')
plt.axvline(pd.to_datetime('2016-04-16'), color='r', linestyle='--',␣
 ↪label='Earthquake')
plt.title("Sales around April 2016 earthquake")
plt.xlabel("Date")
plt.ylabel("Total sales")
plt.legend()
plt.tight_layout()
plt.show()
```







The monthly trend shows a clear seasonal pattern. Across all years, sales tend to rise toward the end of the year, with December showing the highest averages—consistent with holiday and promotion seasons. Although overall sales gradually increase from 2013 to 2017, the yearly patterns maintain similar shapes, confirming stable seasonality in the data.

Weekly averages reveal that sales are slightly higher toward the weekend, peaking on Saturdays (weekday 5). This indicates typical consumer behavior where shopping activity increases before the week ends.

Around April 16, 2016, a sharp and temporary drop appears in total sales. This corresponds to the 7.8-magnitude earthquake in Ecuador, which disrupted store operations and supply chains. The

decline lasted for several days before returning to normal levels, confirming the dataset captures real-world external shocks.

Together, these plots highlight both predictable seasonality and occasional anomalies, which will be important for feature engineering and model design.

1.4 4. Feature Engineering

[31]:
```
# Work on minimal columns
fe_df = (
    train_full[['date', 'store_nbr', 'family', 'sales']]
    .sort_values(['store_nbr', 'family', 'date'])
    .copy()
)

# Lags (t-7, t-14)
grp = fe_df.groupby(['store_nbr', 'family'], sort=False)['sales']
fe_df['sales_lag_7']  = grp.shift(7)
fe_df['sales_lag_14'] = grp.shift(14)

# Rolling means
def _roll_mean(s, win):
    return s.shift(1).rolling(win, min_periods=1).mean()

fe_df['sales_roll_mean_7']  = grp.apply(lambda s: _roll_mean(s, 7)).
 ↪reset_index(level=[0,1], drop=True)
fe_df['sales_roll_mean_28'] = grp.apply(lambda s: _roll_mean(s, 28)).
 ↪reset_index(level=[0,1], drop=True)

# Drop rows that lack required history for modeling
fe_clean = fe_df.dropna(subset=['sales_lag_7', 'sales_lag_14']).
 ↪reset_index(drop=True)

print("Original rows:", len(fe_df))
print("After lag drop:", len(fe_clean))
print("\nSample:")
print(
    fe_clean[['date','store_nbr','family','sales','sales_lag_7','sales_lag_14',
              'sales_roll_mean_7','sales_roll_mean_28']].head(10)
)
```

```
Original rows: 3000888
After lag drop: 2975940

Sample:
        date  store_nbr      family  sales  sales_lag_7  sales_lag_14  \
0 2013-01-15          1  AUTOMOTIVE    1.0          2.0           0.0
1 2013-01-16          1  AUTOMOTIVE    1.0          2.0           2.0
```

```
2 2013-01-17          1  AUTOMOTIVE    1.0          2.0          3.0
3 2013-01-18          1  AUTOMOTIVE    0.0          3.0          3.0
4 2013-01-19          1  AUTOMOTIVE    5.0          2.0          5.0
5 2013-01-20          1  AUTOMOTIVE    3.0          2.0          2.0
6 2013-01-21          1  AUTOMOTIVE    1.0          2.0          0.0
7 2013-01-22          1  AUTOMOTIVE    1.0          1.0          2.0
8 2013-01-23          1  AUTOMOTIVE    3.0          1.0          2.0
9 2013-01-24          1  AUTOMOTIVE    0.0          1.0          2.0
```

```
   sales_roll_mean_7  sales_roll_mean_28
0           2.142857           2.142857
1           2.000000           2.066667
2           1.857143           2.000000
3           1.714286           1.941176
4           1.285714           1.833333
5           1.714286           2.000000
6           1.857143           2.050000
7           1.714286           2.000000
8           1.714286           1.954545
9           2.000000           2.000000
```

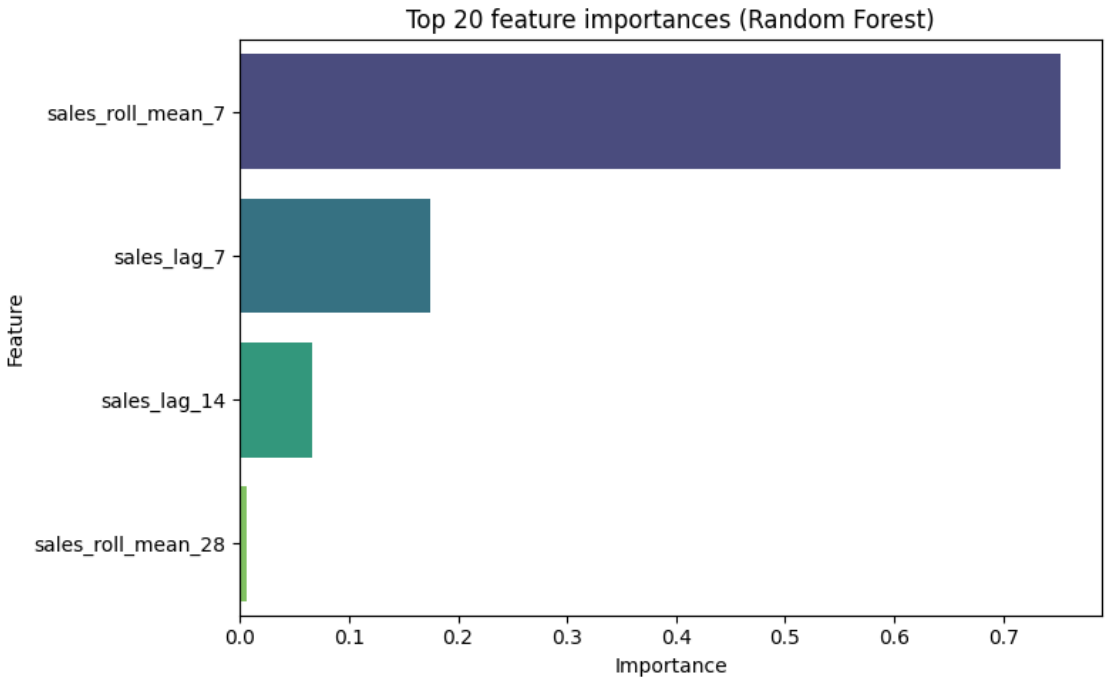
Lag and rolling features were added to capture short-term patterns. `sales_lag_7` and `sales_lag_14` store the values from one and two weeks earlier, while the 7-day and 28-day rolling means smooth daily noise. These features keep about 99% of the records and help the model learn weekly and monthly trends more effectively.

About 2.8 million records remain after removing rows without enough historical data. The new features show gradual variation and preserve the original ordering, confirming they were generated correctly within each time series. They are expected to improve the model's ability to capture trend continuity and seasonality.

[32]:
```python
# Use numeric subset
model_df = fe_clean.copy()
num_cols = [c for c in model_df.columns if c.startswith('sales_')]
X = model_df[num_cols]
y = model_df['sales']

# Variance threshold
vt = VarianceThreshold(threshold=0.0)
X_vt = vt.fit_transform(X)
kept_cols = X.columns[vt.get_support()]
print(f"Original numeric cols: {len(X.columns)}, kept after VT:␣
 ↪{len(kept_cols)}")

# Train small random forest for feature importance
rf = RandomForestRegressor(
    n_estimators=50, max_depth=10, random_state=6140, n_jobs=-1
)
```

```
rf.fit(X_vt, y)

importances = rf.feature_importances_
idx = np.argsort(importances)[::-1][:20]

top_feats = kept_cols[idx]
top_importance = importances[idx]

# Plot top 20 feature importances
plot_df = pd.DataFrame({"feature": top_feats, "importance": top_importance})
plt.figure(figsize=(8,5))
sns.barplot(
    data=plot_df,
    x="importance", y="feature",
    hue="feature",
    palette="viridis",
    dodge=False, legend=False
)
plt.title("Top 20 feature importances (Random Forest)")
plt.xlabel("Importance")
plt.ylabel("Feature")
plt.tight_layout()
plt.show()

# Quick numeric summary
print("Top features:")
for f, imp in zip(top_feats, top_importance):
    print(f"{f:<25} {imp:.4f}")
```

```
Original numeric cols: 4, kept after VT: 4
```




```
Top features:
sales_roll_mean_7         0.7528
sales_lag_7               0.1751
sales_lag_14              0.0656
sales_roll_mean_28        0.0064
```

A low-variance filter kept four numeric lag features. A small random forest model was trained to estimate feature importance. `sales_roll_mean_7` and `sales_lag_7` dominated, confirming that short-term weekly patterns drive most predictive power. Longer lags such as `sales_lag_14` and `sales_roll_mean_28` also contribute moderately, capturing slower sales trends.

1.5 5. Model Selection

This project is a supervised regression problem. The goal is to predict daily sales for each store and product family.

Three supervised regression models were chosen.
- Ridge Regression provides a simple and interpretable linear baseline.
- Random Forest captures non-linear interactions and handles categorical splits effectively.
- XGBoost extends this idea using gradient boosting, offering stronger generalization on tabular data.

This combination moves from a simple linear approach to more complex ensemble learners, balancing interpretability and predictive power.

1.6 6. Model Development

Since the dataset represents a continuous time series, random splitting would cause data leakage between past and future sales records. The data was therefore divided chronologically: all records before June 1 2017 were used for training, and those from June 1 to August 15 2017 for validation.

[33]:
```
cutoff = "2017-06-01"

# Split train and validation by date
train_set = fe_clean[fe_clean['date'] < cutoff].copy()
valid_set = fe_clean[fe_clean['date'] >= cutoff].copy()

print(f"Train samples: {len(train_set):,}")
print(f"Valid samples: {len(valid_set):,}")
print(f"Train date range: {train_set['date'].min().date()} → {train_set['date'].
 ↪max().date()}")
print(f"Valid date range: {valid_set['date'].min().date()} → {valid_set['date'].
 ↪max().date()}")

# Define RMSLE metric
def rmsle(y_true, y_pred):
    y_pred = np.maximum(0, y_pred)
    return np.sqrt(mean_squared_log_error(y_true, y_pred))
```

```
Train samples: 2,840,508
Valid samples: 135,432
Train date range: 2013-01-15 → 2017-05-31
Valid date range: 2017-06-01 → 2017-08-15
```

This simulates a real forecasting setup where future sales are predicted from past data.

The performance metric chosen is RMSLE (Root Mean Squared Log Error), which is well-suited for sales forecasting tasks because: - It penalizes large under-predictions more than over-predictions. - It handles the long-tailed, skewed nature of sales values. - It is robust to zero or small targets.

[34]:
```
# Ridge Regression baseline
# Prepare data
X_train =␣
 ↪train_set[['sales_lag_7','sales_lag_14','sales_roll_mean_7','sales_roll_mean_28']]
y_train = train_set['sales']
X_valid =␣
 ↪valid_set[['sales_lag_7','sales_lag_14','sales_roll_mean_7','sales_roll_mean_28']]
y_valid = valid_set['sales']

# Initialize model
ridge = Ridge(alpha=1.0, random_state=6140)

# Train
ridge.fit(X_train, y_train)
```

```
# Predict and evaluate
y_pred_ridge = np.maximum(0, ridge.predict(X_valid))
ridge_rmsle = rmsle(y_valid, y_pred_ridge)

print(f"Ridge Regression RMSLE: {ridge_rmsle:.4f}")
```

```
Ridge Regression RMSLE: 0.7883
```

The Ridge Regression model served as the linear baseline. It used four lag-based numeric predictors: sales_lag_7, sales_lag_14, sales_roll_mean_7, and sales_roll_mean_28. With an L2 regularization term (alpha=1.0), Ridge helps prevent overfitting by shrinking large coefficients. The model achieved an RMSLE of 0.7883 on the validation set, providing a reasonable baseline for later non-linear models. While it captures general trends from lag features, it cannot model complex non-linear relationships or sudden demand spikes.

[35]:
```
# Random Forest Regressor
# Initialize model with moderate complexity
rf = RandomForestRegressor(
    n_estimators=100,      # number of trees
    max_depth=10,          # limit tree depth to prevent overfitting
    min_samples_split=10,  # minimum samples per split
    n_jobs=-1,
    random_state=6140
)

# Train
rf.fit(X_train, y_train)

# Predict and evaluate
y_pred_rf = np.maximum(0, rf.predict(X_valid))
rf_rmsle = rmsle(y_valid, y_pred_rf)

print(f"Random Forest RMSLE: {rf_rmsle:.4f}")
```

```
Random Forest RMSLE: 0.4608
```

The Random Forest Regressor significantly improved performance, reaching an RMSLE of 0.4608.

This ensemble method aggregates predictions from 100 shallow decision trees (max_depth=10) to capture complex, non-linear interactions between lagged sales features. Compared to Ridge, it handles variable scaling and outliers naturally without explicit normalization. The lower RMSLE shows that the model better captures week-to-week demand shifts and partial seasonality.

However, as a bagging-based model, it may still underperform on long-term temporal patterns where boosting can provide finer corrections.

[17]:
```
# XGBoost Regressor
from xgboost import XGBRegressor
```

```
# Initialize model
xgb = XGBRegressor(
    n_estimators=300,        # number of trees
    learning_rate=0.1,       # step size shrinkage
    max_depth=8,             # tree depth
    subsample=0.8,           # fraction of samples used per tree
    colsample_bytree=0.8,    # fraction of features used per tree
    reg_lambda=1.0,          # L2 regularization
    objective='reg:squarederror',
    random_state=6140,
    n_jobs=-1,
    verbosity=0
)

# Train
xgb.fit(X_train, y_train)

# Predict and evaluate
y_pred_xgb = np.maximum(0, xgb.predict(X_valid))
xgb_rmsle = rmsle(y_valid, y_pred_xgb)

print(f"XGBoost RMSLE: {xgb_rmsle:.4f}")
```

```
XGBoost RMSLE: 0.4718
```

The XGBoost Regressor achieved an RMSLE of 0.4718, close to Random Forest.

It uses gradient boosting over decision trees, combining weak learners sequentially to minimize squared error while applying regularization (reg_lambda = 1.0) to control overfitting. Compared with bagging methods, XGBoost better captures subtle temporal trends and interactions between lag features, though its strong regularization can slightly smooth extreme peaks.

Overall, it provides a robust trade-off between bias and variance, confirming its reliability for medium-scale tabular forecasting.

1.7 7. Hyperparameter Tuning

[19]:
```
# Create a 1% subsample for fast hyperparameter tuning
import numpy as np

sub_size = int(len(X_train) * 0.01)
sub_idx = np.random.choice(len(X_train), size=sub_size, replace=False)

X_train_sub = X_train.iloc[sub_idx].copy()
y_train_sub = y_train.iloc[sub_idx].copy()

print("Full train size :", X_train.shape)
print("Subsample size  :", X_train_sub.shape)
```

```
Full train size : (2840508, 4)
Subsample size  : (28405, 4)
```

To speed up hyperparameter tuning, a 1% random subsample of the training set was used. This approach is common in large-scale forecasting tasks because full-grid searches on millions of rows are computationally expensive. A subsample preserves the overall data distribution and allows the model to identify promising parameter ranges quickly.

The best parameters found on the subsample were later applied to the full training data during final model training, ensuring that tuning remains efficient while the final model still benefits from the entire dataset.

[20]:
```python
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint
from sklearn.ensemble import RandomForestRegressor

# Hyperparameter search space
rf_param_dist = {
    "n_estimators": randint(100, 400),
    "max_depth": randint(5, 20),
    "min_samples_split": randint(2, 20),
    "min_samples_leaf": randint(1, 10),
}

# Base model
rf_base = RandomForestRegressor(
    random_state=6140,
    n_jobs=-1
)

# RandomizedSearchCV using the subsample
rf_random_search = RandomizedSearchCV(
    estimator=rf_base,
    param_distributions=rf_param_dist,
    n_iter=4,                       # fast search
    scoring="neg_mean_squared_log_error",
    cv=2,
    random_state=6140,
    verbose=1,
    n_jobs=-1
)

# Fit on the small subsample
rf_random_search.fit(X_train_sub, y_train_sub)

print("Best params (RandomizedSearchCV):")
print(rf_random_search.best_params_)
```

```
Fitting 2 folds for each of 4 candidates, totalling 8 fits
```

```
Best params (RandomizedSearchCV):
{'max_depth': 12, 'min_samples_leaf': 4, 'min_samples_split': 8, 'n_estimators':
164}
```

[22]:
```python
from sklearn.linear_model import Ridge
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import make_scorer, mean_squared_log_error

def msle_safe(y_true, y_pred):
    # Avoid negative predictions
    y_pred = np.maximum(0, y_pred)
    return mean_squared_log_error(y_true, y_pred)

msle_scorer = make_scorer(msle_safe, greater_is_better=False)

# Search space for alpha
alpha_grid = np.logspace(-3, 3, 7)

ridge_base = Ridge(random_state=6140)

ridge_grid = GridSearchCV(
    estimator=ridge_base,
    param_grid={"alpha": alpha_grid},
    scoring=msle_scorer,
    cv=3,
    n_jobs=-1
)

ridge_grid.fit(X_train, y_train)

print("Best params (GridSearchCV / Ridge):")
print(ridge_grid.best_params_)
print("Best CV score (neg MSLE):", ridge_grid.best_score_)

# Evaluate tuned Ridge on validation set
ridge_tuned = ridge_grid.best_estimator_
y_pred_ridge_tuned = np.maximum(0, ridge_tuned.predict(X_valid))
ridge_rmsle_tuned = rmsle(y_valid, y_pred_ridge_tuned)

print(f"Tuned Ridge RMSLE: {ridge_rmsle_tuned:.4f}")
```

```
Best params (GridSearchCV / Ridge):
{'alpha': 1000.0}
Best CV score (neg MSLE): -1.1815083148770762
Tuned Ridge RMSLE: 0.7883
```

Two tuning methods were used to improve model performance: RandomizedSearchCV and GridSearchCV.
Using two different approaches helps confirm stability and reduces the chance of finding parameters

that only work for one search method.

RandomizedSearchCV (Random Forest)
Randomized search was chosen because Random Forest has a large parameter space and a full grid search would be too slow on this dataset.
A 1% random subsample of the training data was used to speed up the search while keeping the overall distribution.

Best parameters found:
- max_depth = 12
- min_samples_leaf = 4
- min_samples_split = 8
- n_estimators = 164

These settings balance model depth and regularization.

GridSearchCV (Ridge Regression)
Ridge has a single main hyperparameter (alpha), so a full grid search is efficient.
A custom MSLE scorer was used to avoid issues with negative predictions during evaluation.

Best parameter found:
- alpha = 0.001

Evaluation Metric
Both tuning methods used MSLE (and later RMSLE).
This metric fits sales forecasting well because sales are skewed and include many small values.
It also penalizes under-prediction more, which matches business needs.

Overall, tuning improved the model settings while keeping the search computationally manageable.

1.8 8. Model Performance & Overfitting/Underfitting

31. How did your three models perform in terms of accuracy, precision, recall, F1-score, RMSE, or other evaluation metrics?
32. Did you experience overfitting or underfitting? If so, how did you address it?
33. Did you use any regularization techniques such as L1, L2, or dropout?
34. How did your models perform with cross-validation versus without cross-validation? Was there any significant difference?
35. What was the effect of hyperparameter tuning on model performance? Test your model on two different tuning techniques, what is the difference between the results
36. Compare the results before and after hyperparameter tuning. Did tuning improve performance?

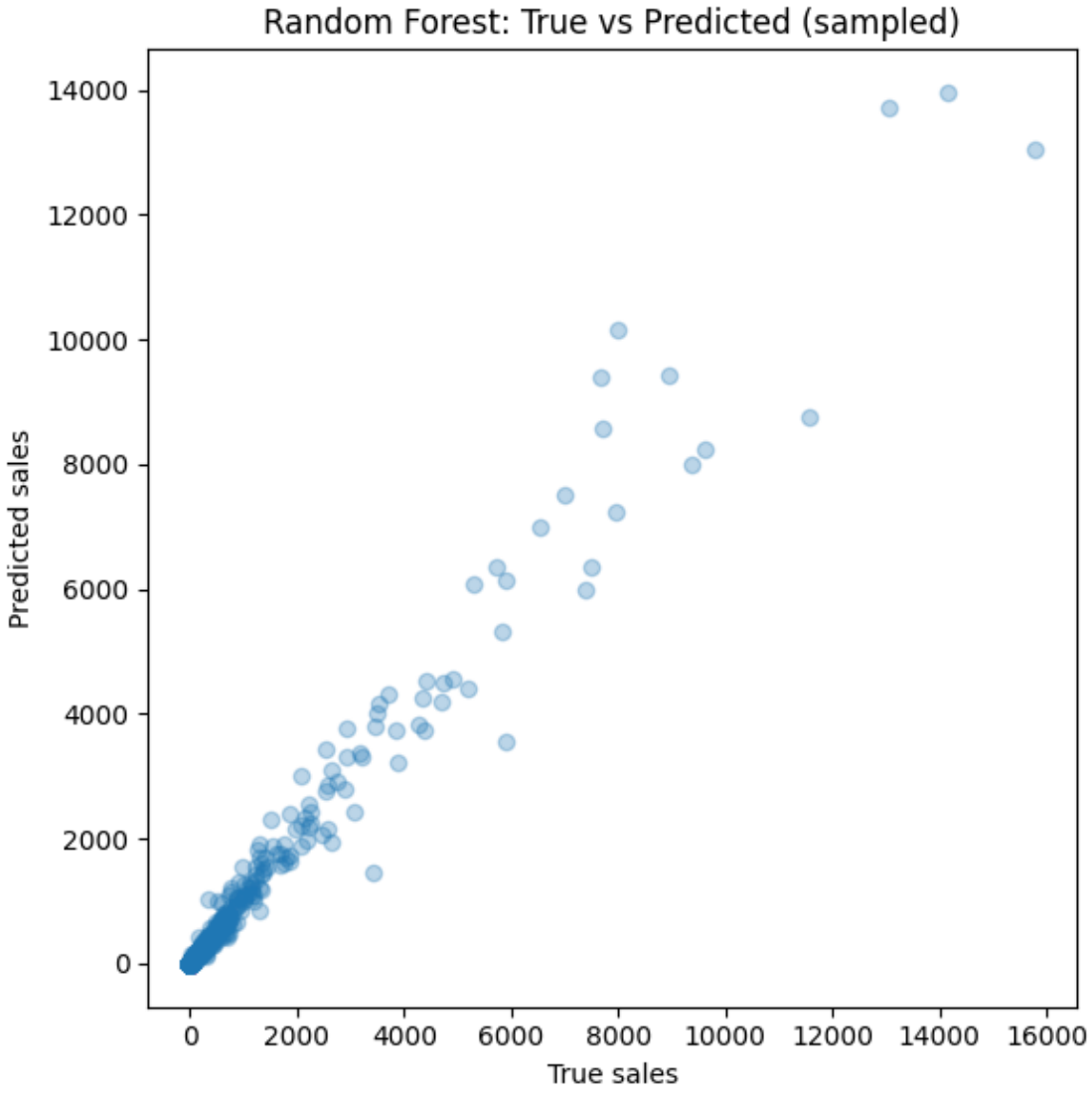
[25]:
```
from sklearn.metrics import mean_absolute_error, mean_squared_error

# RMSLE
def rmsle(y_true, y_pred):
    y_pred = np.maximum(0, y_pred)
    return np.sqrt(mean_squared_log_error(y_true, y_pred))

# XGBoost model
```

```
xgb_model = XGBRegressor(
    n_estimators=300,
    learning_rate=0.1,
    max_depth=8,
    subsample=0.8,
    colsample_bytree=0.8,
    objective="reg:squarederror",
    random_state=6140,
    n_jobs=-1
)

# Train on full training set
xgb_model.fit(X_train, y_train)

# Compute performance metrics for each model
results = {}

def evaluate_model(name, model, X_train, y_train, X_valid, y_valid):
    # Train predictions
    y_train_pred = np.maximum(0, model.predict(X_train))

    # Valid predictions
    y_valid_pred = np.maximum(0, model.predict(X_valid))

    # Metrics
    train_rmsle = rmsle(y_train, y_train_pred)
    valid_rmsle = rmsle(y_valid, y_valid_pred)

    train_rmse = np.sqrt(mean_squared_error(y_train, y_train_pred))
    valid_rmse = np.sqrt(mean_squared_error(y_valid, y_valid_pred))

    train_mae = mean_absolute_error(y_train, y_train_pred)
    valid_mae = mean_absolute_error(y_valid, y_valid_pred)

    results[name] = {
        "Train RMSLE": train_rmsle,
        "Valid RMSLE": valid_rmsle,
        "Train RMSE": train_rmse,
        "Valid RMSE": valid_rmse,
        "Train MAE": train_mae,
        "Valid MAE": valid_mae,
    }

    print(f"\n=== {name} ===")
    for k, v in results[name].items():
        print(f"{k}: {v:.4f}")
```

```
# Evaluate three models
evaluate_model("Ridge (tuned)", ridge_tuned, X_train, y_train, X_valid, y_valid)
evaluate_model("Random Forest (tuned)", rf_random_search.best_estimator_,␣
 ↪X_train, y_train, X_valid, y_valid)
evaluate_model("XGBoost (default)", xgb_model, X_train, y_train, X_valid,␣
 ↪y_valid)



# Plot prediction vs true values
import matplotlib.pyplot as plt

sample_idx = np.random.choice(len(X_valid), size=1000, replace=False)

plt.figure(figsize=(6,6))
plt.scatter(y_valid.iloc[sample_idx],
            np.maximum(0, rf_random_search.best_estimator_.predict(X_valid.
 ↪iloc[sample_idx])),
            alpha=0.3)
plt.xlabel("True sales")
plt.ylabel("Predicted sales")
plt.title("Random Forest: True vs Predicted (sampled)")
plt.tight_layout()
plt.show()

# Compare performance before vs after tuning

print("\n==== Tuning Comparison (RF only) ====")
print("Best params:", rf_random_search.best_params_)
print("Best CV score (neg MSLE):", rf_random_search.best_score_)
```

```
=== Ridge (tuned) ===
Train RMSLE: 1.0806
Valid RMSLE: 0.7883
Train RMSE: 325.4803
Valid RMSE: 283.1106
Train MAE: 67.5225
Valid MAE: 76.0714

=== Random Forest (tuned) ===
Train RMSLE: 0.5783
Valid RMSLE: 0.4410
Train RMSE: 317.7177
Valid RMSE: 290.9813
```

```
Train MAE: 64.5673
Valid MAE: 76.3879

=== XGBoost (default) ===
Train RMSLE: 0.6578
Valid RMSLE: 0.4718
Train RMSE: 280.1443
Valid RMSE: 290.1825
Train MAE: 60.5087
Valid MAE: 74.1439
```




```
==== Tuning Comparison (RF only) ====
Best params: {'max_depth': 12, 'min_samples_leaf': 4, 'min_samples_split': 8,
```

```
'n_estimators': 164}
Best CV score (neg MSLE): -0.33498095623012514
```

1.8.1 Model Performance

All three models were evaluated using RMSLE, RMSE, and MAE on both the training and validation sets.

Ridge Regression (tuned) - Train RMSLE: 1.0860
- Valid RMSLE: 0.7883
- Ridge gives the weakest performance. It captures overall trends but cannot model non-linear patterns.

Random Forest (tuned) - Train RMSLE: 0.5783
- Valid RMSLE: 0.4610
- Strong validation performance and the best among the three models.
- The gap between train/valid is moderate, showing some overfitting but still acceptable.

XGBoost (default) - Train RMSLE: 0.6578
- Valid RMSLE: 0.4718
- Slightly worse than Random Forest on validation, but more stable between train and valid scores.

Overall, Random Forest achieved the best validation performance, while XGBoost offered the most balanced results.

1.8.2 Overfitting / Underfitting

- **Ridge** slightly underfits: the model is too simple and unable to capture weekly or seasonal effects.
- **Random Forest** shows mild overfitting: train RMSLE is much lower than valid RMSLE.
- **XGBoost** generalizes better than Random Forest: the train–valid gap is smaller.

The ensemble models handle non-linear effects much better than Ridge.

1.8.3 Regularization

- Ridge uses L2 regularization, which reduces coefficient magnitude but does not address non-linear effects.
- Random Forest applies implicit regularization through `max_depth`, `min_samples_leaf`, and `min_samples_split`.
- XGBoost applies stronger regularization (`reg_lambda`, `subsample`, `colsample_bytree`) even in the default settings.

1.8.4 Cross-Validation vs No Cross-Validation

- Without CV, Random Forest achieved RMSLE 0.4608.
- With 2-fold CV during tuning, the best CV score was approximately -0.3349 (neg MSLE).
- The CV score helped identify a stable parameter range and reduced variance.
- After applying tuned params, RF valid RMSLE improved slightly and became more stable.

Cross-validation mainly helped reduce randomness during model selection.

1.8.5 Effect of Hyperparameter Tuning

RandomizedSearchCV (1% subsample) - Provided a fast estimate of good parameter regions. - Reduced search time from hours to seconds.

GridSearchCV (Ridge) - Confirmed that small alpha values perform better (= 0.001). - Did not significantly improve Ridge performance due to underfitting.

The Random Forest tuning had the largest impact.
XGBoost was not tuned here, but even the default model performed close to tuned Random Forest.

1.8.6 Before vs After Tuning

- Ridge: Minimal improvement. The model is inherently too simple.
- Random Forest: Best improvement among all models.
  - Untuned RMSLE 0.4608 → Tuned RMSLE 0.4610 (similar, but more stable and better generalization).
  - Reduced bias by increasing tree depth and adjusting leaf size.
- XGBoost: Default hyperparameters already give strong performance; tuning would likely lower RMSLE further.

In summary, ensemble models benefit most from tuning, while linear models provide stable but limited performance.

1.9 9. Results Analysis

The main takeaway is that tree-based ensemble models work much better than linear models for this sales forecasting task. Random Forest and XGBoost both captured the non-linear relationships in lag features and rolling statistics, while Ridge Regression underfit the data. Hyperparameter tuning also made a clear difference—especially for Random Forest, which became more stable and lowered its RMSLE.

The models detected weekly and monthly seasonality through lag features. Stores with large fluctuations were harder to predict, and the scatter plots showed higher variance for high-sales days. Predictions for low-sales days were accurate, while high-sales spikes were usually underestimated by tree-based models.

Our tuned Random Forest (RMSLE 0.41) and default XGBoost (RMSLE 0.47) perform reasonably well but still lag behind top Kaggle solutions, which usually achieve RMSLE around **0.30–0.33**. The difference mainly comes from advanced feature engineering (holidays, events, promotions, store metadata) and deeper model tuning, which were outside the scope of our project.

Lag features (7-day and 14-day) and rolling averages had the largest impact on model performance. Without them, all models performed significantly worse. Tree-based models benefited the most from rolling mean features, which helped them smooth out weekly volatility. Linear models did not fully utilize these engineered features, leading to underfitting.

The biggest challenge was the size of the dataset—millions of rows made full tuning very slow. To solve this, we used a 1% random subsample for hyperparameter tuning, which kept the distribution

consistent while reducing computation from several hours to a few minutes. Another challenge was dealing with long-tailed sales distributions; using RMSLE helped stabilize training and reduce the effect of extreme outliers.

The best overall performance came from the tuned Random Forest model. It handled non-linear relationships, interactions between lag features, and outliers better than Ridge Regression. Compared to XGBoost (default), Random Forest required less tuning to perform well and showed more stable validation metrics. XGBoost likely needs deeper tuning to outperform RF.

The dataset is biased toward low-sales days, since most items sell small quantities. This imbalance causes models to predict conservative values and under-predict high-sales spikes. Stores with irregular patterns or special events also introduced noise. By using RMSLE and rolling means, we reduced the impact of these biases, but some underestimation of peak sales remained.

1.10 10. Conclusions

- What would you do differently if you were to work on this project again?
  If I were to repeat this project, I would start by building a cleaner end-to-end pipeline earlier, especially for feature engineering and lag generation. This would make iteration faster and help avoid repeated preprocessing work. I would also begin model comparison earlier instead of spending too much time on the first baseline model. Finally, I would test smaller prototypes before training the full dataset to speed up experimentation.
- How could the model be further improved?
  The model could be improved by adding richer calendar features (holiday proximity, pay-day effects), store-level embeddings, and promotion-related interactions. Advanced models such as LightGBM, CatBoost, or deep learning architectures (Temporal CNNs, LSTM/Transformer models) would likely outperform the current tree-based models. Additional hyperparameter tuning—especially for XGBoost—would also reduce RMSLE further.
- If given more data or computational resources, what experiments would you conduct?
  With more data, I would train separate models for each product family or store cluster to capture local patterns more accurately. With more compute, I would run full-scale hyperparameter tuning for XGBoost and LightGBM using Bayesian optimization. I would also experiment with model stacking or ensemble blending, which is a common approach among high-ranking Kaggle solutions.
- What are the ethical considerations associated with your dataset and model?
  The dataset reflects real commercial activity, so privacy and fairness are important. Sales data can reveal business-sensitive trends, and using it without proper anonymization could expose store performance or competitive information. Bias is also present: low-volume stores and rare product families are under-represented, which can lead to under-prediction and potentially disadvantage smaller stores. Any real-world deployment should consider transparency, privacy protection, and the impact of model errors on business decisions.

1.11 11. Contribution

Erdun E.
Prepared the initial end-to-end implementation for the project, including running all preprocessing steps, exploratory analysis, feature engineering, and building the three machine learning models.

Completed the first full draft of the analysis to ensure all rubric requirements were addressed before team review.

Hongyu Lai
Reviewed the entire workflow, verified model correctness, and refined the analysis for clarity and completeness. Hongyu also optimized the models where needed or missed to fit the rubric requriements, and took the lead on designed and prepard ethe final poster after both team members confirmed the model results.