WEEK 4

Feature Engineering I

Turning cleaned data into effective model inputs

Day 16	**Encoding**	Label encoding, one-hot, high-cardinality strategies
Day 17	**Scaling**	Min-Max, standardization, robust scaling, leakage
Day 18	**Binning & Discretization**	Equal-width, equal-frequency, domain-driven
Day 19	**Transformation Methods**	Log, sqrt, Box-Cox, Yeo-Johnson
Day 20	**Integrated Exercise**	Full pipeline, interactions, evaluation, bias

BASE DATASET

```
df = pd.DataFrame({"city": ["NY", "SF", "LA"], "membership": ["bronze", "silver", "gold"],
    "purchases": [3, 5, 1]})
# Encoding choices strongly affect model performance, interpretability, and bias
```

DAY 16

Encoding Categorical Features

Label Encoding, One-Hot, and High-Cardinality Strategies

OBJECTIVES

- Distinguish nominal vs ordinal categories
- Implement label and one-hot encoding
- Understand the dummy variable trap
- Handle high-cardinality categories
- Analyze encoding impact on model families

ACTIVITY

Apply one-hot encoding to a categorical feature.

Compare model behavior before and after encoding.

ASSESSMENT

Scenario-based questions: choose encoding strategies and justify decisions.

DAY 16 · ENCODING

Why Categorical Data Cannot Be Used Directly

Raw categories have no inherent numerical magnitude

Key Problems

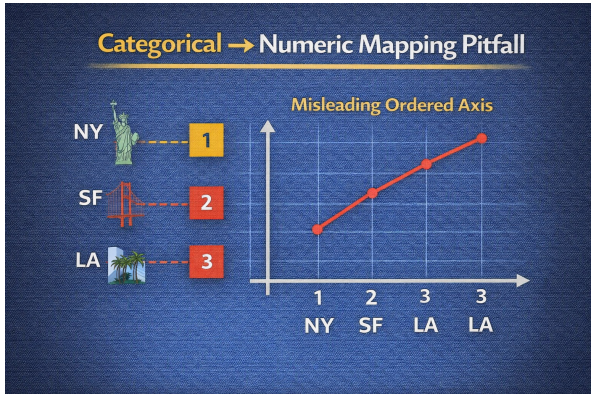
- Most ML libraries require numeric float inputs
- Implicit ordering from arbitrary integers (NY→1, SF→2) misleads linear models
- Distance metrics don't know 'NY' and 'SF' are just different labels

Risks of Naive Mapping

- Linear models: LA (3) appears 'between' SF (2) and CHI (4)
- Differences seem meaningful: CHI - NY = 3
- Trees still split on ordered thresholds

NAIVE MAPPING - RISKY

```
city_map_naive = {"NY": 1, "SF": 2, "LA": 3, "CHI": 4}
df16["city_naive_int"] = df16["city"].map(city_map_naive)
# For a linear model: implies LA is "between" SF and CHI - wrong!
```



DAY 16 · ENCODING

Nominal vs Ordinal Categories

Why the distinction matters for encoding

Nominal Categories

Categories with NO inherent order

Examples: city, color, product type

Any numeric ordering is artificial and can mislead models

Ordinal Categories

Categories with meaningful order but not equal spacing

Examples: bronze < silver < gold

Preserving order helps capture monotonic relationships

CREATING ORDINAL CODE FOR MEMBERSHIP LEVEL

```
membership_order = pd.CategoricalDtype(
    categories=["bronze", "silver", "gold"], ordered=True)

df16["membership_ord"] = df16["membership_level"].astype(membership_order).cat.codes
# Now bronze=0, silver=1, gold=2 - reflecting meaningful order
```

DAY 16 · ENCODING

Label Encoding — Definition, Uses, and Risks

Maps each category to an integer

When Label Encoding Is Acceptable

- Ordinal categories where order is meaningful
- Tree-based models on nominal categories (splits still separate groups)
- Memory-efficient single column

When Label Encoding Is Risky

- Linear or distance-based models on nominal variables
- Imposes false order and distance
- KNN/K-Means heavily affected

LABEL ENCODING WITH sklearn

```
from sklearn.preprocessing import LabelEncoder

le_city = LabelEncoder()
df16["city_label"] = le_city.fit_transform(df16["city"])
print("Classes:", le_city.classes_)  # Sorted alphabetically: CHI, LA, NY, SF
```

```
CHI -> 0
LA -> 1
NY -> 2
SF -> 3
```

DAY 16 · ENCODING

One-Hot Encoding with pandas

Binary indicator column for each category

How It Changes Feature Space

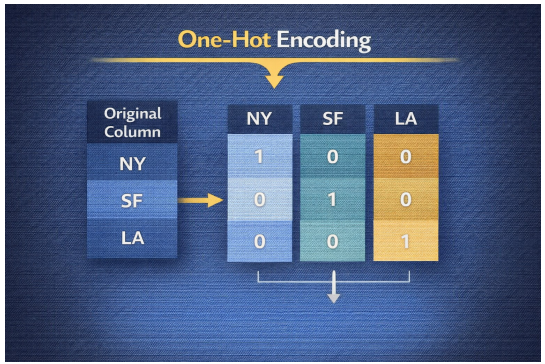
- Replaces one column with k binary columns
- Each row has 1 in exactly one column, 0 in others
- Avoids artificial ordering

ML Interpretation

- Linear models: separate coefficient per category
- Trees: can split on each dummy independently
- Distance-based: sees binary coordinates (more sparse)

ONE-HOT WITH pd.get_dummies

```
df16_ohe = pd.get_dummies(df16, columns=["city"], prefix="city")
print(df16_ohe.head())
# Creates: city_CHI, city_LA, city_NY, city_SF columns
```



DAY 16 · ENCODING

One-Hot Encoding with sklearn OneHotEncoder

Pipeline-ready with sparse matrix output

Benefits

- Fit once on training, apply same mapping to test
- Handles unseen categories with configurable behavior
- Sparse output is memory-efficient

Integration

- Works with ColumnTransformer for mixed pipelines
- Sparse output passes directly into ML estimators
- For inspection, convert to dense (avoid on large data)

OneHotEncoder USAGE

```
from sklearn.preprocessing import OneHotEncoder

ohe = OneHotEncoder(sparse=True, handle_unknown="ignore")
city_encoded = ohe.fit_transform(df16[["city"]])
print("Encoded shape:", city_encoded.shape, "Categories:", ohe.categories_)
```

DAY 16 · ENCODING

Dummy Variable Trap and Redundant Columns

Perfect multicollinearity in linear models

The Problem

Sum of all dummy columns equals 1

Including all dummies + intercept creates perfect multicollinearity

OLS estimates become non-unique, stability suffers

Solution

Drop one dummy column - becomes reference group

Coefficients interpreted relative to dropped category

Trees are robust to redundancy but dropping reduces dimensionality

DROP ONE DUMMY TO AVOID REDUNDANCY

```
df16_ohe_drop = pd.get_dummies(df16, columns=["city"], prefix="city", drop_first=True)
print(df16_ohe_drop.head())  # CHI becomes reference (dropped), only LA, NY, SF remain
```

DAY 16 · ENCODING

High-Cardinality Categorical Variables

Thousands of categories cause memory and overfitting issues

Challenges

- Huge one-hot matrices → memory issues, overfitting
- Sparse signals per category → unreliable coefficients
- Basic one-hot treats each category independently

Strategies

- Frequency encoding: replace category with its count
- Target encoding: mean target per category (leakage risk!)
- Group rare categories into 'other' or domain-based groups

FREQUENCY ENCODING

```
city_counts = df16["city"].value_counts()
df16["city_freq"] = df16["city"].map(city_counts)
# Preserves common vs rare info without dimension explosion
```

row	city_CHI	city_LA	city_NY	city_SF
0	0	0	1	0
1	0	0	0	1
2	0	0	1	0
3	0	1	0	0

row	city	city_freq
0	NY	2
1	SF	1
2	NY	2
3	LA	1

DAY 16 · ENCODING

Encoding Impact on Different Model Families

Choose encoding based on model type

Linear Models

- One-hot: separate coefficients
- Label on nominal: harmful

✓ Use one-hot

Tree-Based

- Can work with label encoding
- One-hot may or may not help

✓ Either approach works

Distance-Based

- KNN, K-Means, SVM RBF
- Arbitrary integers distort

✓ One-hot or binary

QUICK COMPARISON: NAIVE VS ONE-HOT

```
from sklearn.linear_model import LinearRegression
X_naive = df16[["city_naive_int"]]; X_ohe = pd.get_dummies(df16[["city"]], drop_first=True)
lm_naive = LinearRegression().fit(X_naive, y); lm_ohe = LinearRegression().fit(X_ohe, y)
print("R² naive:", lm_naive.score(X_naive, y), "R² one-hot:", lm_ohe.score(X_ohe, y))
```

Encoding exercise

Choose encoding based on model type

1 - Load the dataset
2 - Label encode the city column
3 - Apply one-hot encoding on the city column on a copy dataset
4 - Print the result and compare between the two methods

LABEL ENCODING WITH sklearn

```
from sklearn.preprocessing import LabelEncoder

le_city = LabelEncoder()
df16["city_label"] = le_city.fit_transform(df16["city"])
print("Classes:", le_city.classes_)  # Sorted alphabetically: CHI, LA, NY, SF
```

DAY 17

Scaling Numerical Features

Min-Max, Standardization, and Robust Scaling

OBJECTIVES

- Explain why scaling matters for distance/gradient algorithms
- Implement Min-Max, standard, and robust scaling
- Visualize distributions before and after
- Apply scaling without leaking information
- Understand scaling impact per algorithm family

ACTIVITY

Scale two numerical features and compare distributions visually.

ASSESSMENT

Justify when Min-Max preferred over standardization and vice versa.

DAY 17 · SCALING

Why Scaling Is Necessary for Many Algorithms

Unscaled features with different ranges distort models

Distortions Without Scaling

- Distance calculations (KNN, K-Means): larger-range features dominate
- Gradient descent (linear, neural nets): uneven steps
- SVM with RBF kernel: margin calculations affected

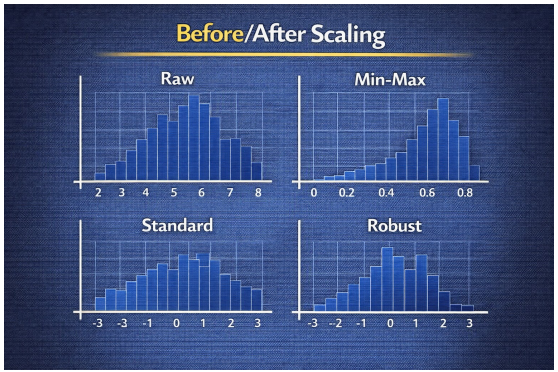
Example

- CRIM ranges 0–80
- RM (rooms) ranges 3–9

Unscaled: KNN chooses neighbors primarily based on CRIM simply due to larger numeric range

RANGE INSPECTION

```
print(df17[["CRIM", "RM"]].agg(["min", "max"]))
# If plugged directly into KNN, CRIM dominates distance calculations
```



DAY 17 · SCALING

Min-Max Scaling — Definition and Implementation

Rescales to fixed range [0, 1]

Formula

x_scaled = (x - x_min) / (x_max - x_min)

- Preserves relative ordering (monotonic)
- Compresses all values into [0, 1]

Use Cases

- Algorithms that rely on distances or bounded input
- Retain distribution shape but normalize range

Caveat: Outliers compress bulk of data toward 0

MinMaxScaler

```
from sklearn.preprocessing import MinMaxScaler
mm_scaler = MinMaxScaler()
df17_mm = df17.copy()
df17_mm[["CRIM_mm", "RM_mm"]] = mm_scaler.fit_transform(df17[["CRIM", "RM"]])
# Both now in [0, 1]
```

DAY 17 · SCALING

Standardization (Z-Score Scaling)

Mean ≈ 0, standard deviation ≈ 1

Formula

$z = (x - \mu) / \sigma$

- Centers data and scales variance to one
- Useful for regularized models, PCA

Effects

- Ensures penalties treat features comparably in Lasso/Ridge
- Does NOT remove skewness - only rescales
- Sensitive to outliers in mean/std calculation

StandardScaler

```
from sklearn.preprocessing import StandardScaler
std_scaler = StandardScaler()
df17_std = df17.copy()
df17_std[["CRIM_std", "RM_std"]] = std_scaler.fit_transform(df17[["CRIM", "RM"]])
print(df17_std[["CRIM_std", "RM_std"]].agg(["mean", "std"]))  # mean≈0, std≈1
```

DAY 17 · SCALING

Robust Scaling — Handling Outliers Gracefully

Uses median and IQR instead of mean and std

Formula

x_robust = (x - median) / IQR

Reduces sensitivity to outliers

Use Cases

- Features with heavy tails or clear outliers
- Want scaling without outliers dominating
- Does NOT cap outliers, just reduces influence

RobustScaler

```
from sklearn.preprocessing import RobustScaler
rob_scaler = RobustScaler()
df17_rob = df17.copy()
df17_rob[["CRIM_rob", "RM_rob"]] = rob_scaler.fit_transform(df17[["CRIM", "RM"]])
```

DAY 17 · SCALING

Scaling and Data Leakage — Safe Workflows

Fit scaler on training data only

Safe Workflow

1. Split data into train and test
2. Fit scaler on train only
3. Apply fitted scaler to both train and test

Never call fit on combined train+test!

Leakage Effects

Using full dataset leaks future information into training

- Over-optimistic metrics
- Degraded production performance

LEAKAGE-SAFE SCALING

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=17)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)  # Fit on train
X_test_scaled = scaler.transform(X_test)         # Only transform test
```

DAY 17 · SCALING

Scaling Impact on Different Algorithms

Essential for some, irrelevant for others

Essential

- KNN / K-Means: distance sensitive
- SVM with RBF kernel
- Neural networks

Recommended

- Linear/logistic regression: helps convergence
- Regularized models: comparable penalties

Usually Not Needed

- Decision trees
- Random forests
- Gradient boosting

KNN WITH AND WITHOUT SCALING

```
from sklearn.neighbors import KNeighborsRegressor
knn_raw = KNeighborsRegressor(n_neighbors=5).fit(X_train, y_train)
knn_scaled = KNeighborsRegressor(n_neighbors=5).fit(X_train_scaled, y_train)
print("MSE raw:", mse(y_test, knn_raw.predict(X_test)))
print("MSE scaled:", mse(y_test, knn_scaled.predict(X_test_scaled)))  # Usually lower
```

DAY 18

Binning & Discretization

Converting Continuous Features to Categories

OBJECTIVES

- Explain why and when discretization helps
- Implement equal-width and equal-frequency binning
- Design domain-driven bins (e.g., age groups)
- Assess interpretability vs information loss
- Compare binned vs continuous in models

ACTIVITY

Bin an Age column into Child, Teen, Adult, Senior categories.

ASSESSMENT

Submit code and explanation for chosen bin thresholds.

DAY 18 • BINNING

Why Discretization Can Help Models

Replace raw values with bin indicators

Benefits

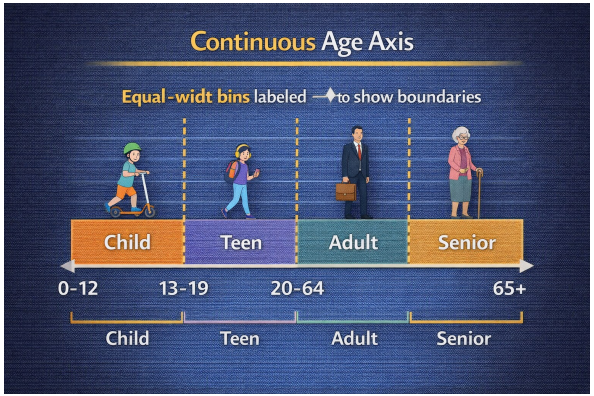
- Noise reduction: small variations within bin treated as equivalent
- Non-linear effects: bins capture step changes (age 18, 21, 65)
- Interpretability: bins are human-readable

Risks

- Information loss: differences within bin ignored
- Arbitrary boundaries can introduce artifacts
- May hurt tree models that already find optimal splits

SIMPLE EQUAL-WIDTH BINS

```
df18["age_bin_3"] = pd.cut(df18["age"], bins=3)
print(df18[["age", "age_bin_3"]].head())
# pd.cut chooses boundaries based on min/max - may not align with meaningful ages
```



What `pd.cut(..., bins=3)` does

- It looks at the **minimum and maximum** age in `df18["age"]`.
- It splits that full range into **3 equal-width intervals**.
- Each age is replaced by an **interval label** like:
  - `(0.999, 24.0]`
  - `(24.0, 47.0]`
  - `(47.0, 70.0]`

So `age_bin_3` is a categorical value showing which interval the age falls into.

DAY 18 · BINNING

Equal-Width Binning with pd.cut

Divides range into intervals of equal size

How It Works

- Split [min, max] into N equal-sized intervals
- Simple but may create unbalanced bins if distribution skewed

Custom Edges

Specify domain-informed edges instead of purely equal-width

Example: [0, 18, 35, 50, 100] for age ranges

EQUAL-WIDTH WITH CUSTOM EDGES

```
bin_edges = [0, 18, 35, 50, 100]
labels = ["Child", "YoungAdult", "Adult", "Senior"]
df18["age_bins_width"] = pd.cut(df18["age"], bins=bin_edges, labels=labels, right=False)
print(df18["age_bins_width"].value_counts())
```

DAY 18 · BINNING

Equal-Frequency Binning with pd.qcut

Roughly same number of observations per bin

How It Works

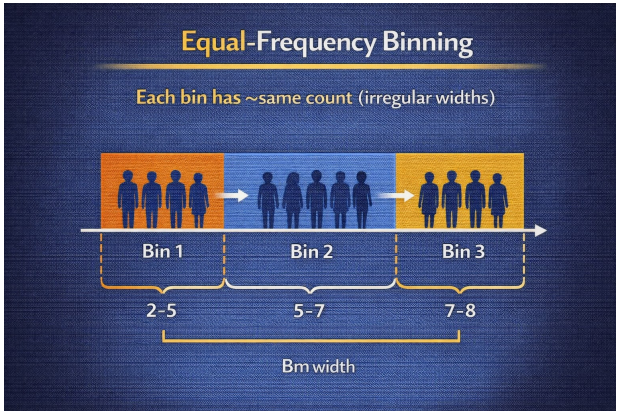
- Quantile-based: each bin gets ~same count
- Stabilizes estimates but produces irregular bin widths
- Boundaries may not align with meaningful values

Benefits

- Each bin has similar statistical power
- Good for modeling and stability
- Works well when equal-width would create sparse bins

QUARTILE-BASED BINS

```
df18["age_bins_quantiles"] = pd.qcut(df18["age"], q=4, labels=["Q1", "Q2", "Q3", "Q4"])
print(df18["age_bins_quantiles"].value_counts())  # ~Equal counts per bin
```



Domain-Driven Binning — Age Groups Example

Meaningful bins aligned with policy or behavior

Domain Knowledge

- Children, teens, adults, seniors align with many business/policy thresholds
- Legal ages vary by region (13, 18, 21, 65…)
- Interpretable to stakeholders

Best Practice

- Document bin boundaries and rationale
- Ensure ranges reflect domain-specific rules
- Easy to explain in reports and audits

AGE BINS: CHILD, TEEN, ADULT, SENIOR

```
age_edges = [0, 13, 18, 65, 120]
age_labels = ["Child", "Teen", "Adult", "Senior"]
df18["age_group"] = pd.cut(df18["age"], bins=age_edges, labels=age_labels, right=False)
print(df18["age_group"].value_counts())
```

DAY 19

Transformation Methods

Log, Sqrt, Box-Cox, and Yeo-Johnson

OBJECTIVES

- Recognize skewed distributions and why they're problematic
- Apply np.log1p and np.sqrt to reduce skew
- Use PowerTransformer for Box-Cox/Yeo-Johnson
- Compare before/after distributions visually
- Understand impact on linear model assumptions

ACTIVITY

Apply log transformation to skewed feature and visualize before/after.

ASSESSMENT

Explain how transformations interact with linearity assumptions.

DAY 19 · TRANSFORMATIONS

Skewed Distributions and Why They Matter

Long tails affect model behavior

Why Skew Is Problematic

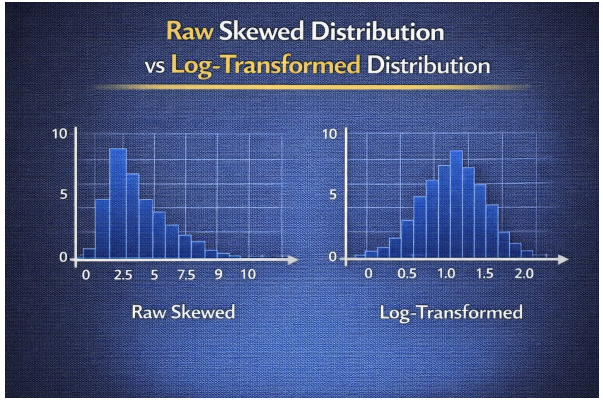
- Linear regression: outliers exert large influence on coefficients
- Gaussian-based models assume/prefer approximate normality
- Visualization and interpretation harder with extreme skew

Common Skewed Features

- Income, spend, transaction amounts
- Time-on-site, page views, clicks
- Positive values with long right tail

MEASURING SKEWNESS

```
print(df19["spend"].describe())  # Note large max vs median
sns.histplot(df19["spend"], kde=True)
plt.title("Spend distribution (raw)")  # Long right tail
```



DAY 19 · TRANSFORMATIONS

Log Transform with np.log1p

Compress large values, spread small values

Properties

- log1p(x) = log(1 + x) — safe for zeros
- Compresses large values
- Spreads small values
- Makes relative differences more linear

Use Cases

- Positive-valued features with long right tails
- Income, time-on-site, click counts
- When relative differences matter more than absolute

LOG-TRANSFORMING SPEND

```
df19["spend_log1p"] = np.log1p(df19["spend"])

fig, axes = plt.subplots(1, 2, figsize=(10, 4))
sns.histplot(df19["spend"], ax=axes[0]); axes[0].set_title("Spend raw")
sns.histplot(df19["spend_log1p"], ax=axes[1]); axes[1].set_title("Spend log1p")  # More symmetric
```

DAY 19 · TRANSFORMATIONS

Square-Root Transformation

Less aggressive than log for count-like data

Properties

- sqrt(x) reduces skew less than log
- Keeps stronger distinction between medium and high values
- Works well for counts with zeros

Use Cases

- Count features: events, clicks, logins
- When you want to reduce skew moderately
- More interpretable than log for some stakeholders

SQRT TRANSFORMATION OF TRANSACTIONS

```
df19["transactions_sqrt"] = np.sqrt(df19["transactions"])

fig, axes = plt.subplots(1, 2, figsize=(10, 4))
sns.histplot(df19["transactions"], ax=axes[0], discrete=True); axes[0].set_title("Transactions raw")
sns.histplot(df19["transactions_sqrt"], ax=axes[1]); axes[1].set_title("Transactions sqrt")
```

DAY 19 · TRANSFORMATIONS

Transformation Impact on Linear Models

Better fit in transformed space

Without Transformation

- Model fits linear relationship in raw space
- Skew and heavy tails distort coefficients
- Poor fit, unstable predictions

With Transformation

- Model sees more linear relationship
- Better fit, more stable coefficients
- Usually higher R^2 on transformed features

LINEAR REGRESSION COMPARISON

```
X_raw = df19[["spend"]]; X_log = df19[["spend_log1p"]]
lm_raw = LinearRegression().fit(X_raw, y)
lm_log = LinearRegression().fit(X_log, y)
print("R² raw:", lm_raw.score(X_raw, y), "R² log:", lm_log.score(X_log, y))
```

DAY 19 • TRANSFORMATIONS

When NOT to Transform

Interpretability and business logic considerations

Interpretation Challenges

- Log-transformed coefficient is in log units, not dollars
- Stakeholders may prefer models in original scale
- Back-transformation can be confusing

When to Avoid

- Simple descriptive models where interpretability matters
- Business rules use raw thresholds (credit limits)
- Can keep both raw and transformed, let regularization decide

KEEPING BOTH RAW AND TRANSFORMED

```
df19["spend_both"] = df19["spend"]  # Raw
# Already have spend_log1p
# Be careful: avoid multicollinearity issues
```

DAY 20

Integrated Feature Engineering

Combining All Techniques into a Pipeline

OBJECTIVES

- Combine encoding, scaling, binning, transformations
- Explore feature interactions
- Evaluate engineered features properly
- Identify and avoid feature leakage
- Reflect on bias introduced through feature choices

ACTIVITY

Build full feature pipeline with baseline vs enhanced comparison.

ASSESSMENT

Written reflection on feature impact and bias considerations.

DAY 20 · INTEGRATED EXERCISE

Feature Interactions — Why Combine Features?

Individual features may not capture relevant patterns

Intuition

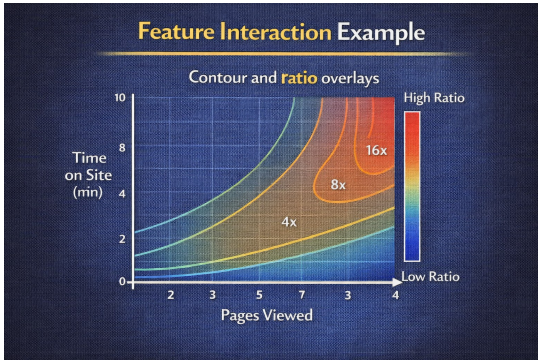
- pages_viewed and time_on_site individually useful
- Their ratio pages_per_minute captures engagement intensity
- Interactions approximate non-linear relationships for linear models

Risks

- Feature explosion: too many interactions → overfit
- Harder interpretability
- Document which interactions and why

SIMPLE INTERACTION FEATURES

```
df20["pages_per_min"] = df20["pages_viewed"] / (df20["session_minutes"] + 1e-3)
df20["is_mobile_high_spend"] = (
    (df20["device_type"] == "mobile") & (df20["basket_value"] > df20["basket_value"].median())
).astype(int)  # Mix numeric and categorical
```



DAY 20 · INTEGRATED EXERCISE

Evaluating Engineered Features — Beyond Accuracy

Check predictive power, stability, and fairness

What to Check

- Predictive performance (accuracy, AUC, RMSE)
- Calibration of probabilities
- Stability across cross-validation folds
- Fairness across subgroups

Compare Baseline vs Enhanced

- Train model with raw features only
- Train model with engineered features
- Measure improvement and interpret

COMPARING MODELS WITH RAW VS ENGINEERED

```
base_features = ["pages_viewed", "session_minutes", "basket_value"]
engineered_features = base_features + ["pages_per_min", "is_mobile_high_spend"]
clf = LogisticRegression(max_iter=1000).fit(X_train[engineered_features], y_train)
print("AUC with engineered:", roc_auc_score(y_test, clf.predict_proba(X_test)[:, 1]))
```

DAY 20 · INTEGRATED EXERCISE

Avoiding Feature Leakage in Engineering

Don't use information unavailable at prediction time

Leakage Examples

- Using post-outcome events (total purchases after signup to predict churn)
- Target encoding on full data including test
- Scaling with mean/std from combined train+test

Prevention

- Fit all transformers on training data only
- Use out-of-fold strategies for target encoding
- Think: would this feature be available at prediction time?

UNSAFE VS SAFE TARGET ENCODING (ILLUSTRATIVE)

```
# UNSAFE (leaky): target mean per city on full data
city_mean_target = df20.groupby("city")["converted"].mean()
df20["city_target_mean_leaky"] = df20["city"].map(city_mean_target)  # DON'T DO THIS
```

DAY 20 · INTEGRATED EXERCISE

Bias Introduced Through Feature Choices

Encoding and grouping decisions affect fairness

How Bias Enters

- Detailed encoding on geography/demographics allows fine-grained discrimination
- Interaction features may highlight group differences
- Target encoding embeds historical bias

Ethical Checks

- Ask: is this feature necessary and proportional?
- Evaluate performance and error rates across subgroups
- Document and justify all feature decisions

CHECKING SUBGROUP PERFORMANCE

```
for device in df20_test["device_type"].unique():
    mask = df20_test["device_type"] == device
    auc = roc_auc_score(df20_test.loc[mask, target], df20_test.loc[mask, "pred"])
    print(f"Device {device}: AUC = {auc:.3f}")  # Flag if performance differs strongly
```

Week 4 Complete!

Feature Engineering I — From Cleaning to Representation

Encoding	Label, one-hot, frequency encoding; dummy trap; high-cardinality strategies
Scaling	Min-Max, standardization, robust; leakage-safe workflows; per-algorithm impact
Binning	Equal-width, equal-frequency, domain-driven; interpretability vs information loss
Transformations	Log, sqrt, Box-Cox, Yeo-Johnson; handling skew; impact on linear models
Integration	Pipelines, interactions, evaluation, leakage avoidance, bias awareness

Key Takeaway

Thoughtful feature construction backed by clear reasoning, code, and awareness of model and ethical implications is the foundation for advanced ML systems.

PATTERN TO CARRY FORWARD

```
def engineer_features(df): df = pd.get_dummies(df, columns=["city"]); df["spend_log1p"] =
np.log1p(df["spend"]); return df
```

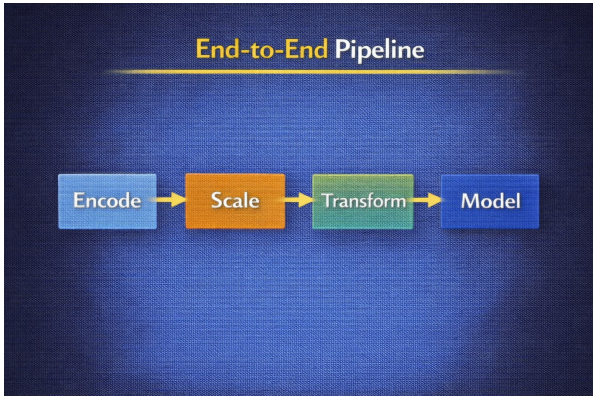
DAY 20 · INTEGRATED EXERCISE

Building the Full Pipeline

Combine all techniques with ColumnTransformer

Pipeline Components

1. Categorical: encode (OneHotEncoder)
2. Numeric: scale (StandardScaler)
3. Binned features
4. Transformed features
5. Integrations



sklearn PIPELINE WITH ColumnTransformer

```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.pipeline import Pipeline

numeric_features = ["pages_viewed", "session_minutes", "basket_value"]
categorical_features = ["city", "device_type"]

numeric_transformer = Pipeline(steps=[("scaler", StandardScaler())])
categorical_transformer = Pipeline(steps=[("ohe", OneHotEncoder(handle_unknown="ignore"))])

preprocess = ColumnTransformer(transformers=[
    ("num", numeric_transformer, numeric_features),
    ("cat", categorical_transformer, categorical_features),
])
```