Lecture 9: Feature Engineering

Dr. Ratnesh Srivastava
CSIT, Guru Ghasidas Vishwavidyalaya (GGV), Bilaspur

July 25, 2025

Introduction to Feature Engineering

Feature engineering is the process of using domain knowledge to extract features from raw data. These features are then used to improve the performance of machine learning models. In essence, it's about transforming the data into a form that better represents the underlying problem to the machine learning algorithm.

Why is Feature Engineering Important?

- **Improved Model Performance:** Well-engineered features can significantly boost the accuracy and robustness of your models.
- **Reduced Model Complexity:** Sometimes, good features allow simpler models to perform as well as, or better than, complex models on raw data.
- **Better Interpretability:** Meaningful features can make it easier to understand why a model makes certain predictions.
- **Handling Non-linearity:** Many algorithms assume linear relationships. Feature engineering can transform non-linear relationships into more linear ones, making them easier for these algorithms to learn.

Key Feature Engineering Techniques

Let's delve into some fundamental techniques with mathematical intuition, examples, and Q&A.

1. Binning (Discretization)

Mathematical Intuition:

Binning transforms continuous numerical features into categorical features (or ordinal features if the bins have a natural order). The intuition is that for certain phenomena, precise numerical values are less important than the *range* or *category* they fall into.

Consider a variable X. Instead of using X directly, we define a set of thresholds $t_1, t_2, \ldots, t_k$. We then assign X to a bin based on which interval $(-\infty, t_1], (t_1, t_2], \ldots, (t_k, \infty)$ it falls into.

Mathematically, if X is a continuous variable, we are creating a new categorical variable X_{binned} such that:

$$X_{binned} = \begin{cases} \text{Bin 1} & \text{if } X \leq t_1 \\ \text{Bin 2} & \text{if } t_1 < X \leq t_2 \\ \vdots & \\ \text{Bin (k+1)} & \text{if } X > t_k \end{cases}$$

Why Binning?

- **Handling Outliers:** Extreme values can disproportionately influence models. Binning can group outliers into a broader category, making the model more robust.
- **Simplifying Relationships:** Sometimes, a linear model struggles with a non-linear relationship. Binning can capture these non-linearities by creating step-like functions.
- **Improving Model Interpretability:** It's often easier to interpret "Age group 25-35" than a specific age like "31.7 years."
- **Compatibility with Certain Models:** Some models (e.g., decision trees) implicitly perform binning, but explicit binning can still be beneficial.

Types of Binning:

- **Equal-Width Binning:** Divides the range of the data into N equal-sized intervals.
- **Equal-Frequency (Quantile) Binning:** Divides the data into N bins, where each bin has approximately the same number of observations.
- **Custom/Domain-Specific Binning:** Based on expert knowledge or business rules.

Example (with Python Code):
Consider a dataset of customer ages for a marketing campaign.

Age	Target (Purchase)
18	No
22	No
28	Yes
35	Yes
42	No
55	No
60	Yes

Problem: We suspect that customer age impacts purchase behavior, but not necessarily linearly. Perhaps certain age groups are more likely to purchase.

Binning Solution (Custom/Domain-Specific): Let's define age groups based on marketing insights.

Python Code:

```python
import pandas as pd

# Sample Data
data = {'Age': [18, 22, 28, 35, 42, 55, 60],
        'Purchase': ['No', 'No', 'Yes', 'Yes', 'No', 'No', 'Yes']}
df = pd.DataFrame(data)

# Define custom bins and labels
bins = [0, 25, 45, df['Age'].max() + 1] # Ensure the last bin captures max age
labels = ['Young Adults (18-25)', 'Middle-Aged (26-45)', 'Seniors (46+)']

# Apply binning
df['Age_Group'] = pd.cut(df['Age'], bins=bins, labels=labels, right=True)

print(df)
```

Listing 1: Custom Binning Example

Transformed Data (Output from Python):

	Age	Purchase	Age_Group
0	18	No	Young Adults (18-25)
1	22	No	Young Adults (18-25)
2	28	Yes	Middle-Aged (26-45)
3	35	Yes	Middle-Aged (26-45)
4	42	No	Middle-Aged (26-45)
5	55	No	Seniors (46+)
6	60	Yes	Seniors (46+)

Now, a model can learn the relationship between "Age Group" and "Purchase", potentially finding that "Middle-Aged" customers are more likely to purchase.

Questions & Answers:

1. **Q: When is binning particularly useful for a feature?**
   - **A:** Binning is useful when the relationship between a continuous feature and the target variable is non-linear, when outliers are present and need to be grouped, or when interpretability of results based on ranges is preferred.
2. **Q: What is a potential disadvantage of binning?**
   - **A:** Binning can lead to information loss. By grouping values, we lose the precise numerical information within each bin. If the exact value is important, binning might reduce model accuracy. The choice of bin boundaries can also be arbitrary and impact performance.
3. **Q: How do you choose the number of bins?**
   - **A:** There's no one-size-fits-all answer. It often involves a trade-off. Too few bins can lead to significant information loss, while too many bins can make the data sparse and overfit to the training data. Common approaches include:

- **Domain knowledge:** If there are natural groupings.
- **Visual inspection:** Histograms can reveal natural clusters.
- **Experimentation:** Try different numbers of bins and evaluate model performance.
- **Statistical methods:** Some methods like ChiMerge can automatically determine optimal bin boundaries.

2. Polynomial Features

Mathematical Intuition:

Polynomial features allow models to capture non-linear relationships between independent variables and the dependent variable. If a linear model struggles to fit the data, it might be because the true relationship is quadratic, cubic, or some other higher-order polynomial.

Given a feature X, we create new features by raising X to different powers. For a single feature X and a degree d, we would generate new features $X^2, X^3, \ldots, X^d$.

For multiple features $X_1, X_2, \ldots, X_n$, polynomial features also include interaction terms. For instance, with degree 2 and features X_1, X_2, the polynomial features would be $X_1, X_2, X_1^2, X_2^2, X_1X_2$.

In general, for a degree d and features $X_1, \ldots, X_n$, polynomial features include all terms of the form:

$$\prod_{i=1}^{n} X_i^{k_i} \quad \text{where } \sum_{i=1}^{n} k_i \leq d \text{ and } k_i \geq 0$$

Why Polynomial Features?

- **Modeling Non-linearity:** This is the primary reason. If the relationship between input and output is curved, linear models won't capture it well. Polynomial features explicitly introduce these curves.
- **Capturing Interactions:** The cross-terms (e.g., X_1X_2) are crucial as they represent how features interact with each other. For example, the effect of "education" on "income" might depend on "experience".

Example (with Python Code):

Consider a simple dataset where the target variable has a quadratic relationship with a feature:

X	Y
1	2
2	5
3	10
4	17
5	26

Problem: A linear model $Y = mX + c$ would not fit this data well. The relationship appears to be $Y = X^2 + 1$.

Polynomial Feature Solution (Degree 2): Create a new feature X^2.

Python Code:

```python
import pandas as pd
from sklearn.preprocessing import PolynomialFeatures
import numpy as np

# Sample Data
data = {'X': [1, 2, 3, 4, 5],
        'Y': [2, 5, 10, 17, 26]}
df = pd.DataFrame(data)

# Create PolynomialFeatures transformer (degree 2)
# include_bias=False prevents adding a column of ones (intercept)
poly = PolynomialFeatures(degree=2, include_bias=False)

# Transform the 'X' feature. Reshape(-1, 1) is needed for single feature.
X_poly = poly.fit_transform(df[['X']])

# Create a new DataFrame with the polynomial features
poly_df = pd.DataFrame(X_poly, columns=poly.get_feature_names_out(['X']))
df = pd.concat([df, poly_df], axis=1)

print(df)

# Now a linear model can be applied to X and X^2, e.g.:
# from sklearn.linear_model import LinearRegression
# model = LinearRegression()
# model.fit(df[['X', 'X^2']], df['Y'])
# print(f"Coefficients: {model.coef_}") # Should be close to [0, 1] for X, X^2
# print(f"Intercept: {model.intercept_}") # Should be close to 1
```

Listing 2: Polynomial Features Example

Transformed Data (Output from Python):

	X	Y	X^2
0	1	2	1.0
1	2	5	4.0
2	3	10	9.0
3	4	17	16.0
4	5	26	25.0

Now, a linear model can be applied to X and X^2. For example, a linear regression model would learn $Y = 1 \cdot X^2 + 1 \cdot (\text{bias term})$, effectively capturing the quadratic relationship.

Questions & Answers:

1. **Q: What is the main benefit of using polynomial features?**
   - **A:** The main benefit is the ability to model non-linear relationships between features and the target variable, which linear models alone cannot capture. They also help in modeling interactions between features.
2. **Q: What is a major drawback of high-degree polynomial features?**

- **A:** High-degree polynomial features can lead to overfitting, especially with limited data. The model becomes too complex and learns the noise in the training data rather than the true underlying pattern. This is because a high-degree polynomial can fit almost any set of points perfectly, but might not generalize well to unseen data.

3. **Q: When should you consider using polynomial features?**

   - **A:** Consider using polynomial features when:
     - You suspect a non-linear relationship between features and the target variable (e.g., from scatter plots showing curves).
     - Your linear model is performing poorly, and residual plots show patterns indicating uncaptured non-linearity.
     - You want to model interactions between features.

     Start with lower degrees (e.g., 2 or 3) and evaluate performance carefully to avoid overfitting.

3. Log Transforms

Mathematical Intuition:

Logarithmic transformations (e.g., $\log(X)$, $\ln(X)$, $\log_{10}(X)$) are applied to features that have a highly skewed distribution (e.g., positively skewed, with a long tail to the right) or when there's an exponential relationship between features and the target.

The logarithm "compresses" the larger values and "stretches" the smaller values.

$$Y' = \log(Y) \quad \text{or} \quad X' = \log(X)$$

Where X must be strictly positive. If X can be zero or negative, common transformations include $\log(X + 1)$ for non-negative values.

Why Log Transforms?

- **Reducing Skewness:** Many machine learning models (like linear regression) assume that the input features are normally distributed or at least not heavily skewed. Log transforms can make skewed distributions more symmetric, bringing them closer to a normal distribution.

- **Stabilizing Variance:** In some datasets, the variance of a feature might increase with its mean. Log transformation can help stabilize the variance across the range of the feature.

- **Handling Exponential Relationships:** If the target variable grows exponentially with a feature, taking the logarithm of either the feature or the target can linearize this relationship, making it easier for linear models to capture. For example, if $Y = e^X$, then $\ln(Y) = X$, a linear relationship.

- **Mitigating Outlier Impact:** By compressing larger values, log transforms reduce the influence of extreme outliers.

Example (with Python Code):
Consider a dataset of "Income" values, which are typically positively skewed:

Income (INR)
50,000
60,000
75,000
100,000
2,000,000
50,000,000

Problem: The vast difference between the majority of incomes and the few very high incomes can make models sensitive to these outliers and struggle to learn the typical income patterns.

Log Transform Solution (Natural Logarithm, ln): Apply ln(Income).

Python Code:

```
import pandas as pd
import numpy as np

# Sample Data
data = {'Income': [50000, 60000, 75000, 100000, 2000000, 50000000]}
df = pd.DataFrame(data)

# Apply natural log transform (ln)
# Use np.log1p for log(x+1) if there's a possibility of 0 values,
# but here all values are positive so np.log is fine.
df['ln_Income'] = np.log(df['Income'])

print(df)
```

Listing 3: Log Transform Example

Transformed Data (Output from Python):

```
      Income  ln_Income
0      50000  10.819778
1      60000  11.002100
2      75000  11.227757
3     100000  11.512925
4    2000000  14.508658
5   50000000  17.727534
```

Notice how the large differences in "Income" are significantly compressed in "ln(Income)". This often leads to a more balanced distribution and better model performance.

Questions & Answers:

1. **Q: When is a log transform typically applied to a feature?**
   - **A:** Log transforms are typically applied to features that are positively skewed (have a long right tail), have a wide range of values, or when there's an exponential relationship with the target variable. They are also used to reduce the impact of outliers.

2. **Q: What is a prerequisite for applying a standard log transform to a feature?**

   - **A:** The feature must contain only positive values. If a feature contains zero or negative values, you need to add a constant (e.g., $\log(X + 1)$ for non-negative values) or use a different transformation.

3. **Q: How does a log transform help in stabilizing variance?**

   - **A:** If the variability of a variable increases as its magnitude increases (a common pattern in skewed data), applying a log transform can "compress" these larger variations, making the variance more uniform across the range of the data. This is particularly useful for models that assume homoscedasticity (constant variance of errors).

4. Feature Crossing (Interaction Features)

Mathematical Intuition:

Feature crossing involves combining two or more features to create a new, more expressive feature. The intuition is that the combined effect of two or more features might be more predictive than their individual effects.

For numerical features X_1 and X_2, a common feature crossing is multiplication:

$$X_{cross} = X_1 \times X_2$$

For categorical features C_1 and C_2, a feature cross creates a new categorical feature that represents the unique combination of categories from C_1 and C_2.

Why Feature Crossing?

- **Capturing Interactions:** This is the core reason. Often, the effect of one feature on the target variable depends on the value of another feature. Feature crosses explicitly model these conditional relationships.

- **Modeling Non-linearity (implicitly):** While polynomial features create non-linearity from a single feature, feature crosses create non-linearity by combining multiple features. For example, in a linear model, $Y = w_1X_1 + w_2X_2 + w_3(X_1X_2) + b$, the $w_3(X_1X_2)$ term allows the model to learn that the impact of X_1 on Y changes based on the value of X_2.

- **Improving Expressiveness:** They can represent more complex patterns that individual features cannot.

Example (with Python Code):

Consider a model predicting house prices based on "Number of Rooms" and "Neighborhood Type" (e.g., Urban, Suburban, Rural).

Number of Rooms	Neighborhood Type	House Price
3	Urban	100,000
5	Urban	180,000
3	Suburban	120,000
5	Suburban	250,000

Problem: A simple model might learn that more rooms mean higher prices, and suburban homes are generally more expensive than urban homes. However, the *increase* in price due to an additional room might be much higher in a suburban neighborhood than in an urban one.

Feature Crossing Solution: Create an interaction feature between "Number of Rooms" (numerical) and "Neighborhood Type" (categorical). This can be done by creating a separate feature for each combination, or by multiplying the numerical feature with one-hot encoded versions of the categorical feature.

Python Code:

```python
import pandas as pd

# Sample Data
data = {'Number of Rooms': [3, 5, 3, 5],
        'Neighborhood Type': ['Urban', 'Urban', 'Suburban', 'Suburban'],
        'House Price': [100000, 180000, 120000, 250000]}
df = pd.DataFrame(data)

# One-hot encode 'Neighborhood Type'
df_encoded = pd.get_dummies(df, columns=['Neighborhood Type'], prefix='Neighborhood')

# Create interaction features
df_encoded['Rooms_x_Urban'] = df_encoded['Number of Rooms'] * df_encoded['Neighborhood_Urban']
df_encoded['Rooms_x_Suburban'] = df_encoded['Number of Rooms'] * df_encoded['Neighborhood_Suburban']

print(df_encoded)
```

Listing 4: Feature Crossing Example

Transformed Data (Output from Python):

	Number of Rooms	House Price	Neighborhood_Suburban	Neighborhood_Urban	\
0	3	100000	0	1	
1	5	180000	0	1	
2	3	120000	1	0	
3	5	250000	1	0	

	Rooms_x_Urban	Rooms_x_Suburban
0	3	0
1	5	0
2	0	3
3	0	5

A linear model can now learn distinct coefficients for 'Rooms$_{x}$Urban'and'Rooms$_{x}$Suburban'.Thismeansi

Questions & Answers:

1. **Q: What is the primary purpose of feature crossing?**

- **A:** The primary purpose of feature crossing is to capture interaction effects between features, where the effect of one feature on the target variable depends on the value of another feature.

2. **Q: Give an example of a good candidate for a feature cross between two categorical variables.**

   - **A:** If you are predicting whether a customer will buy a product, a cross between "Browser Type" (e.g., Chrome, Firefox) and "Operating System" (e.g., Windows, MacOS, Linux) might be useful. For example, "Chrome_on_MacOS" might indicate a specific user behavior or demographic. Another example: "Gender" and "Product Category".

3. **Q: What is a potential challenge when creating many feature crosses?**

   - **A:** The main challenge is the combinatorial explosion. If you have many features, creating crosses of all possible combinations can lead to a huge number of new features, making the dataset high-dimensional, increasing computational cost, and potentially leading to overfitting if not handled carefully (e.g., through regularization or careful selection of crosses).

5. Domain-Specific Ratios (e.g., Income / Debt)

Mathematical Intuition:

Domain-specific ratios involve creating new features by dividing one existing numerical feature by another, based on insights from the problem domain. The intuition is that the relative proportion between two quantities is often more meaningful than their absolute values individually.

Given features X_A and X_B, a ratio feature X_{ratio} is created as:

$$X_{ratio} = \frac{X_A}{X_B}$$

(assuming $X_B \neq 0$).

Why Domain-Specific Ratios?

- **Capturing Relative Importance:** Absolute values might not be as informative as their ratios. For example, a high income is good, but a high income *relative to high debt* tells a more complete story about financial health.

- **Normalizing for Scale:** Ratios can normalize features that vary widely in scale, making them more comparable.

- **Encoding Expert Knowledge:** This is where domain expertise shines. Experts often know which relationships are critical.

- **Improved Interpretability:** Ratios often have clear real-world interpretations (e.g., "debt-to-income ratio").

Example (with Python Code):
Consider predicting creditworthiness using "Annual Income" and "Total Debt".

Annual Income (INR)	Total Debt (INR)	Creditworthy
500,000	50,000	Yes
1,000,000	500,000	Yes
100,000	80,000	No
2,000,000	2,000,000	No

Problem: While higher income is generally good, and lower debt is generally good, a model might struggle to weigh these factors against each other effectively. A person with 2,000,000 income and 2,000,000 debt is likely less creditworthy than someone with 100,000 income and 5,000 debt.

Domain-Specific Ratio Solution: Create a "Debt-to-Income Ratio" feature.

$$\text{Debt-to-Income Ratio} = \frac{\text{Total Debt}}{\text{Annual Income}}$$

Python Code:

```python
import pandas as pd
import numpy as np

# Sample Data
data = {'Annual Income': [500000, 1000000, 100000, 2000000],
        'Total Debt': [50000, 500000, 80000, 2000000],
        'Creditworthy': ['Yes', 'Yes', 'No', 'No']}
df = pd.DataFrame(data)

# Calculate Debt-to-Income Ratio
# Handle potential division by zero by adding a small epsilon or using .replace()
# For simplicity, assuming no zero income in this example.
df['Debt_to_Income_Ratio'] = df['Total Debt'] / df['Annual Income']

print(df)
```

Listing 5: Domain-Specific Ratio Example

Transformed Data (Output from Python):

```
   Annual Income  Total Debt Creditworthy  Debt_to_Income_Ratio
0         500000       50000          Yes                   0.1
1        1000000      500000          Yes                   0.5
2         100000       80000           No                   0.8
3        2000000     2000000           No                   1.0
```

Now, the "Debt-to-Income Ratio" directly captures the financial health in a single, highly informative feature. A model can easily learn that a higher ratio indicates lower creditworthiness.

Other Examples of Domain-Specific Ratios:

- **Healthcare:** BMI (Weight / Height^2), Glucose-to-Insulin Ratio

- **Finance:** Price-to-Earnings Ratio, Current Ratio (Current Assets / Current Liabilities)
- **Retail:** Conversion Rate (Purchases / Visits), Average Order Value (Revenue / Orders)
- **Sports Analytics:** Points per Game, Assist-to-Turnover Ratio

Questions & Answers:

1. **Q: What is the primary advantage of using domain-specific ratios?**
    - **A:** The primary advantage is that they encapsulate expert knowledge and real-world relationships, providing highly condensed and relevant information to the model, which often leads to better performance and interpretability.
2. **Q: What is a critical consideration when creating a ratio feature?**
    - **A:** You must handle division by zero. If the denominator can be zero, you need a strategy (e.g., replace with a large number, NaN and handle as a missing value, or create a separate binary feature indicating if the denominator was zero).
3. **Q: Can domain-specific ratios be combined with other feature engineering techniques? Provide an example.**
    - **A:** Absolutely! This is very common. For instance, after creating a "Debt-to-Income Ratio," you might find that its distribution is highly skewed, in which case you could apply a **log transform** to it (log(Debt-to-Income Ratio)). Or, if you suspect that the impact of the ratio on the target variable changes at certain thresholds, you could **bin** the "Debt-to-Income Ratio" into categories like "Low Risk," "Medium Risk," "High Risk."

General Q&A on Feature Engineering

1. **Q: What's the relationship between feature engineering and machine learning model performance?**
    - **A:** Feature engineering is often considered the most crucial step in the machine learning pipeline for achieving high model performance. Even with sophisticated models, poor features will lead to poor results. Good features allow simpler models to perform well, and complex models to perform exceptionally well.
2. **Q: Is feature engineering an automated process?**
    - **A:** While there are automated feature engineering tools (e.g., Featuretools, Google's AutoML Tables), the most impactful feature engineering often still relies heavily on human domain expertise, creativity, and iterative experimentation. Automated tools can be a good starting point or help explore combinations, but deep insights often come from a human.

3. **Q: How do you know which feature engineering technique to apply?**

   - **A:** It's an iterative process often involving:
     - **Exploratory Data Analysis (EDA):** Visualizing relationships between features and the target can suggest transformations (e.g., skewed distributions for log transforms, curved relationships for polynomial features).
     - **Domain Knowledge:** This is paramount. Experts can tell you which relationships are important (e.g., ratios).
     - **Hypothesis Testing:** Formulate hypotheses about how features might interact or behave, and test them by creating new features.
     - **Model Performance Evaluation:** Apply transformations/new features, train your model, and evaluate its performance on a validation set.
     - **Error Analysis:** Analyze where your model makes mistakes. These errors can often reveal missing or poorly represented features.