Feature Engineering for Machine Learning: Industry-Level Comprehensive Guide (2025)

Introduction

Feature engineering is the art and science of transforming raw data into meaningful features that improve machine learning model performance. It is often considered the most critical step in the ML pipeline, as even the most sophisticated algorithms cannot overcome poorly engineered features.

In industry settings, data scientists and ML engineers spend 40-60% of their project time on feature engineering. The quality of features directly determines model accuracy, interpretability, generalization capability, and computational efficiency. This comprehensive guide covers industry-standard techniques used by leading organizations like Google, Amazon, Meta, and Netflix to build production-ready ML systems.

What is Feature Engineering?

Feature engineering involves creating, transforming, and selecting input variables (features) that enable machine learning algorithms to learn patterns effectively. It bridges the gap between raw data and model-ready inputs.

Key objectives:

- Improve model predictive performance
- Reduce training time and computational resources
- Enhance model interpretability
- Capture domain knowledge in quantifiable forms
- Handle non-linear relationships and interactions

The Feature Engineering Process:

1. **Understanding the problem** - Define business objectives and success metrics
2. **Exploratory Data Analysis (EDA)** - Analyze data distributions, correlations, patterns
3. **Feature creation** - Generate new features from existing data
4. **Feature transformation** - Scale, normalize, and encode features
5. **Feature selection** - Identify most informative features
6. **Validation** - Test feature effectiveness on model performance
7. **Iteration** - Refine features based on results

Top 10 Industry-Level Feature Engineering Techniques

1. Scaling and Normalization

What is it?

Scaling and normalization transform numeric features to similar ranges, preventing features with larger magnitudes from dominating the learning process. Different algorithms have different sensitivity to feature scales.

Why is it important?

- Gradient-based algorithms (neural networks, logistic regression) converge faster
- Distance-based algorithms (KNN, SVM, K-Means) require similar scales
- Prevents numerical instability in computations
- Improves model interpretability of coefficients
- Essential for regularization techniques (L1/L2)

Common Techniques:

Standardization (Z-score normalization): Transforms features to have mean=0 and standard deviation=1. Best for normally distributed data.

$$x_{scaled} = \frac{x - \mu}{\sigma}$$

Min-Max Scaling: Transforms features to a fixed range [0, 1]. Preserves zero values and maintains relationships.

$$x_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Robust Scaling: Uses median and IQR, robust to outliers.

$$x_{scaled} = \frac{x - median(x)}{IQR(x)}$$

How to implement:

import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler, MinMaxScaler, RobustScaler

Load data

df = pd.read_csv('data.csv')

Method 1: Standardization (Z-score)

```
scaler = StandardScaler()
df_scaled = pd.DataFrame(
scaler.fit_transform(df[['income', 'age', 'credit_score']]),
columns=['income', 'age', 'credit_score']
)
```

Method 2: Min-Max Scaling (0-1 range)

```
minmax_scaler = MinMaxScaler()
df_minmax = pd.DataFrame(
minmax_scaler.fit_transform(df[['price', 'quantity']]),
columns=['price', 'quantity']
)
```

Method 3: Robust Scaling (for data with outliers)

```
robust_scaler = RobustScaler()
df_robust = pd.DataFrame(
robust_scaler.fit_transform(df[['sales', 'revenue']]),
columns=['sales', 'revenue']
)
```

Method 4: Max Abs Scaling (preserves sparsity)

```
from sklearn.preprocessing import MaxAbsScaler
maxabs_scaler = MaxAbsScaler()
df_maxabs = maxabs_scaler.fit_transform(df[['feature1', 'feature2']])
```

Custom scaling function

```
def custom_scale(series, method='standard'):
if method == 'standard':
return (series - series.mean()) / series.std()
elif method == 'minmax':
return (series - series.min()) / (series.max() - series.min())
elif method == 'log':
return np.log1p(series) # log(1 + x) to handle zeros
```

Industry Best Practices:

- Fit scalers on training data only, then transform test/validation sets

- Save scaler objects for production deployment
- Use StandardScaler for normally distributed data
- Use MinMaxScaler when you need bounded ranges
- Use RobustScaler when data contains outliers
- Document scaling parameters for model reproducibility

When to apply:

Algorithm	Scaling Required?
Linear/Logistic Regression	Yes (StandardScaler)
Neural Networks	Yes (StandardScaler or MinMaxScaler)
SVM	Yes (StandardScaler)
KNN	Yes (MinMaxScaler)
Tree-based (RF, XGBoost)	No
Naive Bayes	No

Table 1: Scaling requirements by algorithm type

---

2. Encoding Categorical Variables

What is it?

Machine learning algorithms require numerical inputs, so categorical variables (text labels, categories) must be converted to numeric representations. The encoding method significantly impacts model performance and interpretability.

Why is it important?

- Enables ML algorithms to process categorical data
- Preserves information content of categories
- Affects model dimensionality and complexity
- Impacts training time and memory usage
- Different encodings capture different relationships

Common Encoding Techniques:

Label Encoding: Assigns integers to categories. Use for ordinal data (low, medium, high).

One-Hot Encoding: Creates binary columns for each category. Use for nominal data with few categories.

Target Encoding: Replaces categories with target variable statistics. Powerful but risk of overfitting.

Frequency Encoding: Encodes by category occurrence frequency.

Binary Encoding: Converts categories to binary digits, reducing dimensionality.

How to implement:

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
from category_encoders import TargetEncoder, BinaryEncoder
import category_encoders as ce
```

Sample data

```
df = pd.DataFrame({
'color': ['red', 'blue', 'green', 'red', 'blue'],
'size': ['small', 'medium', 'large', 'medium', 'small'],
'priority': ['low', 'medium', 'high', 'low', 'high'],
'target': [0, 1, 1, 0, 1]
})
```

Method 1: Label Encoding (for ordinal data)

```
le = LabelEncoder()
df['priority_encoded'] = le.fit_transform(df['priority'])
```

Output: low=2, medium=1, high=0 (alphabetical)

Better: Manual mapping for ordinal data

```
priority_map = {'low': 0, 'medium': 1, 'high': 2}
df['priority_ordinal'] = df['priority'].map(priority_map)
```

Method 2: One-Hot Encoding (for nominal data)

```
df_onehot = pd.get_dummies(df, columns=['color'], prefix='color')
```

Creates: color_red, color_blue, color_green columns

Using sklearn OneHotEncoder

from sklearn.preprocessing import OneHotEncoder
ohe = OneHotEncoder(sparse=False, drop='first') # drop first to avoid multicollinearity
encoded = ohe.fit_transform(df[['color']])
feature_names = ohe.get_feature_names_out(['color'])
df_encoded = pd.DataFrame(encoded, columns=feature_names)

Method 3: Target Encoding (mean encoding)

target_encoder = TargetEncoder()
df['color_target_encoded'] = target_encoder.fit_transform(
df['color'], df['target']
)

Manual target encoding with regularization

def target_encode_with_smoothing(df, column, target, smoothing=10):
# Calculate global mean
global_mean = df[target].mean()

```
# Calculate category statistics
agg = df.groupby(column)[target].agg(['mean', 'count'])

# Apply smoothing
smoothed = (agg['mean'] * agg['count'] + global_mean * smoothing) / (agg['count

return df[column].map(smoothed)
```

df['color_smoothed'] = target_encode_with_smoothing(df, 'color', 'target')

Method 4: Frequency Encoding

freq_map = df['color'].value_counts().to_dict()
df['color_frequency'] = df['color'].map(freq_map)

Method 5: Binary Encoding (reduces dimensionality)

```
binary_encoder = BinaryEncoder(cols=['color'])
df_binary = binary_encoder.fit_transform(df)
```

Method 6: Hash Encoding (for high cardinality)

```
from sklearn.feature_extraction import FeatureHasher
hasher = FeatureHasher(n_features=8, input_type='string')
hashed = hasher.transform(df['color'].astype(str).values.reshape(-1, 1))
```

Method 7: Leave-One-Out Encoding (prevents leakage)

```
from category_encoders import LeaveOneOutEncoder
loo_encoder = LeaveOneOutEncoder()
df['color_loo'] = loo_encoder.fit_transform(df['color'], df['target'])
```

Industry Best Practices:

- Use one-hot encoding for low-cardinality nominal features (<10 categories)
- Use target encoding for high-cardinality features, with cross-validation to prevent overfitting
- Apply label encoding only for truly ordinal data
- Consider frequency encoding for categories with natural ordering by occurrence
- Use hash encoding for very high cardinality features (>1000 categories)
- Always encode training and test sets consistently
- Handle unknown categories in test set (use 'handle_unknown' parameter)

Comparison table:

Encoding Type	Pros	Cons	Best Use Case
Label	Simple, low memory	Implies ordering	Ordinal data
One-Hot	No false ordering	High dimensionality	Low cardinality
Target	Captures target relationship	Overfitting risk	High cardinality
Frequency	Simple, informative	May lose information	Ranked categories
Binary	Reduces dimensions	Less interpretable	Medium cardinality
Hash	Handles unseen values	Collision possible	Very high cardinality

Table 2: Comparison of encoding techniques

---

3. Handling Missing Data with Advanced Imputation

What is it?

Beyond simple deletion or mean imputation, advanced techniques use statistical and machine learning methods to predict missing values based on other features, preserving data integrity and relationships.

Why is it important?

- Maximizes use of available data
- Preserves sample size and statistical power
- Maintains relationships between features
- Reduces bias compared to simple deletion
- Can improve model performance significantly

Advanced Imputation Techniques:

How to implement:

```
import pandas as pd
import numpy as np
from sklearn.impute import SimpleImputer, KNNImputer, IterativeImputer
from sklearn.experimental import enable_iterative_imputer
```

Sample data with missing values

```
df = pd.DataFrame({
'age': [25, 30, np.nan, 45, np.nan, 35],
'income': [50000, 60000, 55000, np.nan, 70000, 65000],
'score': [85, np.nan, 78, 92, 88, np.nan]
})
```

Method 1: Simple Statistical Imputation

```
simple_imputer = SimpleImputer(strategy='mean')
df_simple = pd.DataFrame(
simple_imputer.fit_transform(df),
columns=df.columns
)
```

Different strategies

```
median_imputer = SimpleImputer(strategy='median')
mode_imputer = SimpleImputer(strategy='most_frequent')
constant_imputer = SimpleImputer(strategy='constant', fill_value=0)
```

Method 2: KNN Imputation (considers feature similarity)

```
knn_imputer = KNNImputer(n_neighbors=3, weights='distance')
df_knn = pd.DataFrame(
knn_imputer.fit_transform(df),
columns=df.columns
)
```

Method 3: Iterative Imputation (MICE - Multiple Imputation by Chained Equations)

```
iterative_imputer = IterativeImputer(max_iter=10, random_state=42)
df_iterative = pd.DataFrame(
iterative_imputer.fit_transform(df),
columns=df.columns
)
```

Method 4: Forward Fill / Backward Fill (for time series)

df_ffill = df.fillna(method='ffill') # Forward fill
df_bfill = df.fillna(method='bfill') # Backward fill

Method 5: Interpolation (for time series)

df_interpolated = df.interpolate(method='linear')
df_interpolated_time = df.interpolate(method='time') # if datetime index

Method 6: Custom ML-based Imputation

from sklearn.ensemble import RandomForestRegressor

def ml_impute(df, target_col):
"""Use ML model to predict missing values"""
# Separate complete and incomplete rows
complete_rows = df[df[target_col].notna()]
incomplete_rows = df[df[target_col].isna()]

```
if len(incomplete_rows) == 0:
    return df

# Train model on complete data
X_train = complete_rows.drop(columns=[target_col])
y_train = complete_rows[target_col]

model = RandomForestRegressor(n_estimators=100, random_state=42)
model.fit(X_train, y_train)

# Predict missing values
X_predict = incomplete_rows.drop(columns=[target_col])
predictions = model.predict(X_predict)

# Fill missing values
df_filled = df.copy()
df_filled.loc[df[target_col].isna(), target_col] = predictions

return df_filled
```

Method 7: Multiple Imputation (statistical approach)

```
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
```

Create multiple imputed datasets

```
n_imputations = 5
imputed_datasets = []

for i in range(n_imputations):
imputer = IterativeImputer(random_state=i)
imputed_data = pd.DataFrame(
imputer.fit_transform(df),
columns=df.columns
)
imputed_datasets.append(imputed_data)
```

Method 8: Creating Missing Indicator Features

```
from sklearn.impute import MissingIndicator
```

Add binary columns indicating missingness

```
missing_indicator = MissingIndicator()
missing_mask = missing_indicator.fit_transform(df)
missing_cols = [f'{col}_missing' for col in df.columns]
df_with_indicators = pd.concat([
df.fillna(df.mean()),
pd.DataFrame(missing_mask, columns=missing_cols)
], axis=1)
```

Industry Best Practices:

- Analyze missing data patterns first (MCAR, MAR, MNAR)
- Use domain knowledge to guide imputation strategy
- Create missing indicator features to preserve information about missingness
- For MCAR/MAR: use KNN or iterative imputation
- For time series: use forward fill or interpolation
- Validate imputation quality by comparing distributions
- Consider multiple imputation for statistical inference
- Document imputation methods for reproducibility

---

4. Binning and Discretization

What is it?

Binning converts continuous numeric variables into discrete categories or bins. This technique can capture non-linear relationships, reduce noise, and make features more robust to outliers.

Why is it important?

- Captures non-linear patterns that linear models can learn
- Reduces impact of outliers and measurement errors
- Creates interpretable categories from continuous data
- Can improve model performance for certain algorithms
- Enables interaction with categorical features

How to implement:

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import KBinsDiscretizer
```

Sample data

```
df = pd.DataFrame({
'age': [22, 35, 48, 19, 67, 34, 52, 28],
'income': [35000, 55000, 72000, 28000, 95000, 48000, 68000, 42000],
'credit_score': [650, 720, 680, 590, 780, 710, 740, 670]
})
```

Method 1: Equal-width binning (using pd.cut)

```
df['age_bin_equal'] = pd.cut(
df['age'],
bins=4, # Create 4 equal-width bins
labels=['Young', 'Adult', 'Middle', 'Senior']
)
```

With specific bin edges

```
df['age_bin_custom'] = pd.cut(
df['age'],
bins=[0, 25, 40, 60, 100],
labels=['18-25', '26-40', '41-60', '60+']
)
```

Method 2: Equal-frequency binning (using pd.qcut)

df['income_bin_quantile'] = pd.qcut(
df['income'],
q=4, # Create 4 quartiles
labels=['Low', 'Medium-Low', 'Medium-High', 'High']
)

Method 3: Custom business logic binning

def categorize_credit_score(score):
if score < 580:
return 'Poor'
elif score < 670:
return 'Fair'
elif score < 740:
return 'Good'
elif score < 800:
return 'Very Good'
else:
return 'Exceptional'

df['credit_category'] = df['credit_score'].apply(categorize_credit_score)

Method 4: Using sklearn KBinsDiscretizer

Strategy options: 'uniform' (equal width), 'quantile' (equal frequency), 'kmeans'

binner = KBinsDiscretizer(n_bins=5, encode='ordinal', strategy='quantile')
df['income_binned'] = binner.fit_transform(df[['income']])

One-hot encode the bins

binner_onehot = KBinsDiscretizer(n_bins=5, encode='onehot-dense', strategy='quantile')
income_bins_onehot = binner_onehot.fit_transform(df[['income']])

Method 5: Adaptive binning based on target variable

def optimal_binning(df, feature, target, n_bins=5):
"""Create bins that maximize separation of target variable"""
from sklearn.tree import DecisionTreeClassifier

```
# Use decision tree to find optimal splits
dt = DecisionTreeClassifier(max_leaf_nodes=n_bins, random_state=42)
dt.fit(df[[feature]], df[target])

# Get split points from tree
thresholds = dt.tree_.threshold[dt.tree_.threshold != -2]
thresholds = np.sort(np.unique(thresholds))

# Create bins
bins = [-np.inf] + list(thresholds) + [np.inf]
df[f'{feature}_optimal_bin'] = pd.cut(df[feature], bins=bins)

return df
```

Method 6: Log-scale binning for skewed distributions

df['income_log'] = np.log1p(df['income'])
df['income_log_bin'] = pd.qcut(df['income_log'], q=5, labels=False)

Method 7: Interaction binning (binning combinations)

df['age_income_interaction'] = (
df['age_bin_custom'].astype(str) + '_' +
df['income_bin_quantile'].astype(str)
)

Industry Best Practices:

- Use domain knowledge to define meaningful bin boundaries
- Avoid too many bins (overfitting) or too few bins (underfitting)
- Use equal-frequency binning to ensure balanced categories

- Consider target-based binning for classification problems
- Validate that binning improves model performance
- Document bin definitions for production deployment
- Handle edge cases and boundary values explicitly

---

5. Dimensionality Reduction and Feature Extraction

What is it?

Dimensionality reduction transforms high-dimensional data into lower-dimensional representations while preserving important information. This addresses the curse of dimensionality and improves computational efficiency.

Why is it important?

- Reduces computational cost and memory usage
- Mitigates curse of dimensionality
- Removes multicollinearity
- Enables visualization of high-dimensional data
- Can improve model generalization
- Reduces overfitting risk

How to implement:

```
import pandas as pd
import numpy as np
from sklearn.decomposition import PCA, TruncatedSVD, FastICA, NMF
from sklearn.manifold import TSNE
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
import umap
```

Sample high-dimensional data

```
np.random.seed(42)
n_samples = 1000
n_features = 50
X = np.random.randn(n_samples, n_features)
y = np.random.randint(0, 2, n_samples)
```

Method 1: Principal Component Analysis (PCA)

Linear dimensionality reduction

```
pca = PCA(n_components=10) # Reduce to 10 dimensions
X_pca = pca.fit_transform(X)
```

Explained variance ratio

print(f"Explained variance: {pca.explained_variance_ratio_}")
print(f"Cumulative variance: {np.cumsum(pca.explained_variance_ratio_)}")

Automatically select components explaining 95% variance

pca_auto = PCA(n_components=0.95)
X_pca_auto = pca_auto.fit_transform(X)
print(f"Components selected: {pca_auto.n_components_}")

Method 2: Incremental PCA (for large datasets)

from sklearn.decomposition import IncrementalPCA

ipca = IncrementalPCA(n_components=10, batch_size=100)
X_ipca = ipca.fit_transform(X)

Method 3: Kernel PCA (for non-linear relationships)

from sklearn.decomposition import KernelPCA

kpca = KernelPCA(n_components=10, kernel='rbf', gamma=0.1)
X_kpca = kpca.fit_transform(X)

Method 4: t-SNE (for visualization, non-linear)

tsne = TSNE(n_components=2, random_state=42, perplexity=30)
X_tsne = tsne.fit_transform(X)

Method 5: UMAP (faster alternative to t-SNE)

import umap
umap_reducer = umap.UMAP(n_components=2, random_state=42)
X_umap = umap_reducer.fit_transform(X)

Method 6: Linear Discriminant Analysis (supervised)

lda = LinearDiscriminantAnalysis(n_components=1)
X_lda = lda.fit_transform(X, y)

Method 7: Truncated SVD (works with sparse matrices)

svd = TruncatedSVD(n_components=10, random_state=42)
X_svd = svd.fit_transform(X)

Method 8: Independent Component Analysis

ica = FastICA(n_components=10, random_state=42)
X_ica = ica.fit_transform(X)

Method 9: Non-negative Matrix Factorization (for non-negative data)

nmf = NMF(n_components=10, random_state=42)
X_nmf = nmf.fit_transform(np.abs(X)) # NMF requires non-negative values

Method 10: Autoencoder (deep learning approach)

from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model

Define autoencoder architecture

input_dim = X.shape[1]
encoding_dim = 10

input_layer = Input(shape=(input_dim,))
encoded = Dense(encoding_dim, activation='relu')(input_layer)
decoded = Dense(input_dim, activation='linear')(encoded)

autoencoder = Model(input_layer, decoded)
encoder = Model(input_layer, encoded)

```
autoencoder.compile(optimizer='adam', loss='mse')
autoencoder.fit(X, X, epochs=50, batch_size=32, verbose=0)

X_autoencoder = encoder.predict(X)
```

Visualizing explained variance (for PCA)

```
import matplotlib.pyplot as plt

pca_full = PCA()
pca_full.fit(X)

plt.figure(figsize=(10, 6))
plt.plot(np.cumsum(pca_full.explained_variance_ratio_))
plt.xlabel('Number of Components')
plt.ylabel('Cumulative Explained Variance')
plt.title('PCA Explained Variance')
plt.grid(True)
plt.show()
```

Industry Best Practices:

- Use PCA for linear dimensionality reduction with interpretable components
- Apply t-SNE or UMAP only for visualization, not as features for models
- Use LDA when you have labeled data and want supervised reduction
- Scale features before applying PCA (it's sensitive to scale)
- Choose number of components based on explained variance threshold (typically 85-95%)
- Use Kernel PCA or autoencoders for complex non-linear relationships
- Save fitted transformers for consistent production deployment
- Monitor reconstruction error to validate reduction quality

---

6. Feature Selection

What is it?

Feature selection identifies and retains only the most informative features for modeling, removing redundant, irrelevant, or noisy features. Unlike dimensionality reduction, it preserves original features.

Why is it important?

- Reduces overfitting by removing noise
- Improves model interpretability
- Decreases training time
- Reduces computational and memory costs
- Can improve model generalization
- Simplifies model deployment

Feature Selection Methods:

1. **Filter Methods** - Statistical tests independent of ML models

2. **Wrapper Methods** - Use model performance to select features
3. **Embedded Methods** - Feature selection during model training

How to implement:

```
import pandas as pd
import numpy as np
from sklearn.feature_selection import (
SelectKBest, chi2, f_classif, mutual_info_classif,
RFE, SelectFromModel, VarianceThreshold
)
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.linear_model import LassoCV, LogisticRegression
import matplotlib.pyplot as plt
```

Sample data

```
np.random.seed(42)
n_samples = 1000
n_features = 20
X = np.random.randn(n_samples, n_features)
y = (X[:, 0] + X[:, 1] + np.random.randn(n_samples) * 0.5 > 0).astype(int)

feature_names = [f'feature_{i}' for i in range(n_features)]
X_df = pd.DataFrame(X, columns=feature_names)
```

===== FILTER METHODS =====

Method 1: Variance Threshold (remove low-variance features)

```
variance_selector = VarianceThreshold(threshold=0.1)
X_variance = variance_selector.fit_transform(X)
selected_features = X_df.columns[variance_selector.get_support()]
print(f"Features selected by variance: {len(selected_features)}")
```

Method 2: Chi-squared test (for classification with non-negative features)

```
X_positive = np.abs(X) # Chi2 requires non-negative values
chi2_selector = SelectKBest(chi2, k=10)
X_chi2 = chi2_selector.fit_transform(X_positive, y)
chi2_scores = chi2_selector.scores_
chi2_features = X_df.columns[chi2_selector.get_support()]
```

Method 3: ANOVA F-test

```
f_selector = SelectKBest(f_classif, k=10)
X_f = f_selector.fit_transform(X, y)
f_scores = f_selector.scores_
f_features = X_df.columns[f_selector.get_support()]
```

Method 4: Mutual Information

```
mi_selector = SelectKBest(mutual_info_classif, k=10)
X_mi = mi_selector.fit_transform(X, y)
mi_scores = mi_selector.scores_
mi_features = X_df.columns[mi_selector.get_support()]
```

Method 5: Correlation with target

```
correlations = X_df.corrwith(pd.Series(y)).abs().sort_values(ascending=False)
top_corr_features = correlations.head(10).index
```

===== WRAPPER METHODS =====

Method 6: Recursive Feature Elimination (RFE)

```
estimator = RandomForestClassifier(n_estimators=100, random_state=42)
rfe_selector = RFE(estimator, n_features_to_select=10, step=1)
X_rfe = rfe_selector.fit_transform(X, y)
rfe_features = X_df.columns[rfe_selector.get_support()]
rfe_ranking = rfe_selector.ranking_
```

Method 7: Sequential Feature Selection

```
from sklearn.feature_selection import SequentialFeatureSelector

sfs_selector = SequentialFeatureSelector(
estimator,
n_features_to_select=10,
direction='forward', # or 'backward'
cv=5
)
X_sfs = sfs_selector.fit_transform(X, y)
sfs_features = X_df.columns[sfs_selector.get_support()]
```

===== EMBEDDED METHODS =====

Method 8: L1-based feature selection (Lasso)

```
lasso = LassoCV(cv=5, random_state=42)
lasso.fit(X, y)
lasso_selector = SelectFromModel(lasso, prefit=True)
X_lasso = lasso_selector.transform(X)
lasso_features = X_df.columns[lasso_selector.get_support()]
```

Method 9: Tree-based feature importance

```
rf = RandomForestClassifier(n_estimators=100, random_state=42)
rf.fit(X, y)
feature_importances = pd.Series(rf.feature_importances_, index=feature_names)
top_rf_features = feature_importances.sort_values(ascending=False).head(10).index
```

Using SelectFromModel with threshold

```
rf_selector = SelectFromModel(rf, threshold='median')
X_rf = rf_selector.fit_transform(X, y)
rf_selected_features = X_df.columns[rf_selector.get_support()]
```

Method 10: Gradient Boosting feature importance

```
gbm = GradientBoostingClassifier(n_estimators=100, random_state=42)
gbm.fit(X, y)
gbm_importances = pd.Series(gbm.feature_importances_, index=feature_names)
top_gbm_features = gbm_importances.sort_values(ascending=False).head(10).index
```

===== CUSTOM METHODS =====

Method 11: Permutation Importance

```
from sklearn.inspection import permutation_importance

perm_importance = permutation_importance(
rf, X, y,
n_repeats=10,
random_state=42
)
```

```
perm_importances = pd.Series(
perm_importance.importances_mean,
index=feature_names
)
top_perm_features = perm_importances.sort_values(ascending=False).head(10).index
```

Method 12: Boruta (wrapper around Random Forest)

pip install boruta

```
from boruta import BorutaPy

rf_boruta = RandomForestClassifier(n_jobs=-1, max_depth=5, random_state=42)
boruta_selector = BorutaPy(rf_boruta, n_estimators='auto', random_state=42)
boruta_selector.fit(X, y)
boruta_features = X_df.columns[boruta_selector.support_]
```

Visualize feature importances

```
plt.figure(figsize=(12, 6))
feature_importances.sort_values().plot(kind='barh')
plt.xlabel('Feature Importance')
plt.title('Random Forest Feature Importances')
plt.tight_layout()
plt.show()
```

Compare different selection methods

```
comparison = pd.DataFrame({
'Chi2': chi2_features.isin(f_features),
'F-test': f_features.isin(f_features),
'MI': mi_features.isin(f_features),
'RFE': rfe_features.isin(f_features),
'Lasso': lasso_features.isin(f_features),
'RF_Importance': top_rf_features.isin(f_features),
}, index=f_features)
```

Industry Best Practices:

- Start with filter methods for quick initial selection
- Use wrapper methods for optimal performance but higher computational cost
- Prefer embedded methods for efficiency in production
- Combine multiple selection methods and choose features selected by majority
- Use cross-validation to avoid overfitting during selection
- Re-evaluate feature importance as data distributions change
- Document feature selection rationale for model governance

- Monitor selected features' performance over time

---

7. Creating Interaction and Polynomial Features

What is it?

Interaction features capture relationships between multiple features by combining them mathematically (multiplication, division, etc.). Polynomial features create non-linear transformations of existing features.

Why is it important?

- Captures non-linear relationships in linear models
- Reveals hidden patterns through feature combinations
- Improves model expressiveness without changing algorithm
- Can significantly boost performance for linear models
- Enables detection of conditional relationships

How to implement:

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

Sample data

```
df = pd.DataFrame({
'area': [1200, 1500, 1800, 2000, 2200],
'bedrooms': [2, 3, 3, 4, 4],
'age': [5, 10, 15, 8, 3],
'distance_to_city': [10, 15, 20, 12, 8],
'income': [50000, 60000, 75000, 80000, 90000],
'household_size': [2, 3, 4, 3, 2]
})
```

===== INTERACTION FEATURES =====

Method 1: Simple pairwise multiplication

```
df['area_bedrooms'] = df['area'] * df['bedrooms']
df['income_per_person'] = df['income'] / df['household_size']
df['area_per_bedroom'] = df['area'] / df['bedrooms']
```

Method 2: Ratio features

df['bedroom_to_area_ratio'] = df['bedrooms'] / df['area']
df['age_to_distance_ratio'] = df['age'] / (df['distance_to_city'] + 1)

Method 3: Addition/subtraction features

df['total_space_metric'] = df['area'] + df['bedrooms'] * 100

Method 4: Custom domain-specific interactions

df['commute_affordability'] = df['income'] / df['distance_to_city']
df['space_per_person'] = df['area'] / df['household_size']

Method 5: Conditional features

df['is_new_and_close'] = ((df['age'] < 5) & (df['distance_to_city'] < 10)).astype(int)
df['is_large_family'] = (df['household_size'] > 3).astype(int)

Method 6: Systematic interaction generation

def create_interactions(df, features, max_degree=2):
"""Create all pairwise interactions"""
df_interactions = df.copy()

```
for i, feat1 in enumerate(features):
    for feat2 in features[i+1:]:
        # Multiplication
        df_interactions[f'{feat1}_x_{feat2}'] = df[feat1] * df[feat2]

        # Division (avoid division by zero)
        df_interactions[f'{feat1}_div_{feat2}'] = df[feat1] / (df[feat2] + 1e-5)
        df_interactions[f'{feat2}_div_{feat1}'] = df[feat2] / (df[feat1] + 1e-5)

        # Addition
        df_interactions[f'{feat1}_plus_{feat2}'] = df[feat1] + df[feat2]

        # Difference
```

```
        df_interactions[f'{feat1}_minus_{feat2}'] = df[feat1] - df[feat2]

    return df_interactions
```

numeric_features = ['area', 'bedrooms', 'age', 'distance_to_city']
df_with_interactions = create_interactions(df, numeric_features)

===== POLYNOMIAL FEATURES =====

Method 7: Sklearn PolynomialFeatures

poly = PolynomialFeatures(degree=2, include_bias=False, interaction_only=False)
poly_features = poly.fit_transform(df[numeric_features])
poly_feature_names = poly.get_feature_names_out(numeric_features)
df_poly = pd.DataFrame(poly_features, columns=poly_feature_names)

Degree 3 polynomial

poly_3 = PolynomialFeatures(degree=3, include_bias=False)
poly_3_features = poly_3.fit_transform(df[['area', 'bedrooms']])

Interaction only (no squared terms)

poly_interact = PolynomialFeatures(degree=2, interaction_only=True, include_bias=False)
interact_features = poly_interact.fit_transform(df[numeric_features])

Method 8: Custom polynomial features

df['area_squared'] = df['area'] ** 2
df['age_squared'] = df['age'] ** 2
df['income_log'] = np.log1p(df['income'])
df['area_log'] = np.log1p(df['area'])

Method 9: Root and reciprocal transformations

df['area_sqrt'] = np.sqrt(df['area'])
df['distance_reciprocal'] = 1 / (df['distance_to_city'] + 1)

Method 10: Three-way interactions (for specific domain knowledge)

df['area_bed_age'] = df['area'] * df['bedrooms'] * df['age']
df['density_affordability'] = (df['area'] / df['bedrooms']) * (df['income'] / 1000)

===== STATISTICAL INTERACTIONS =====

Method 11: Group statistics

df['area_vs_mean'] = df['area'] / df.groupby('bedrooms')['area'].transform('mean')
df['income_vs_median'] = df['income'] / df.groupby('household_size')
['income'].transform('median')

Method 12: Binned interactions

df['age_bin'] = pd.cut(df['age'], bins=[0, 5, 10, 100], labels=['new', 'medium', 'old'])
df['distance_bin'] = pd.cut(df['distance_to_city'], bins=[0, 10, 20, 100], labels=['close',
'medium', 'far'])
df['age_distance_interaction'] = df['age_bin'].astype(str) + '_' + df['distance_bin'].astype(str)

Industry Best Practices:

- Start with domain-knowledge-based interactions
- Use automated polynomial features for quick experimentation (degree 2-3)
- Be cautious of exponential feature growth with high-degree polynomials
- Apply feature selection after creating interactions to remove redundant ones
- Normalize features before creating interactions to prevent magnitude issues
- Use interaction terms especially with linear models (they benefit most)
- Tree-based models automatically capture interactions, so manual creation less critical
- Document interaction logic for interpretability

---

8. Text Feature Engineering

What is it?

Text feature engineering transforms unstructured text data into numerical representations that machine learning algorithms can process. This includes techniques from basic counting to advanced embeddings.

Why is it important?

- Enables ML models to process natural language data
- Captures semantic meaning and relationships
- Critical for NLP tasks (classification, sentiment analysis, etc.)
- Can significantly impact model performance

- Balances expressiveness with computational efficiency

How to implement:

```
import pandas as pd
import numpy as np
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer,
HashingVectorizer
from sklearn.decomposition import LatentDirichletAllocation, NMF
import re
import nltk
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer, WordNetLemmatizer
```

Download required NLTK data

nltk.download('stopwords')

nltk.download('wordnet')

nltk.download('punkt')

Sample text data

```
documents = [
"Machine learning is a subset of artificial intelligence",
"Deep learning uses neural networks with multiple layers",
"Natural language processing helps computers understand text",
"Computer vision enables machines to interpret visual data",
"Reinforcement learning trains agents through rewards"
]
```

===== TEXT PREPROCESSING =====

```
def preprocess_text(text):
"""Comprehensive text preprocessing"""
# Lowercase
text = text.lower()

    # Remove URLs
    text = re.sub(r'http\S+|www\S+|https\S+', '', text)

    # Remove emails
```

```
    text = re.sub(r'\S+@\S+', '', text)

    # Remove special characters and digits
    text = re.sub(r'[^a-zA-Z\s]', '', text)

    # Remove extra whitespace
    text = re.sub(r'\s+', ' ', text).strip()

    return text
```

documents_clean = [preprocess_text(doc) for doc in documents]

===== BASIC TEXT FEATURES =====

Method 1: Character-level features

df = pd.DataFrame({'text': documents})
df['char_count'] = df['text'].str.len()
df['word_count'] = df['text'].str.split().str.len()
df['avg_word_length'] = df['char_count'] / df['word_count']
df['uppercase_count'] = df['text'].str.count(r'[A-Z]')
df['digit_count'] = df['text'].str.count(r'\d')
df['special_char_count'] = df['text'].str.count(r'[^a-zA-Z0-9\s]')

Method 2: Word-level features

df['unique_word_count'] = df['text'].apply(lambda x: len(set(x.lower().split())))
df['stop_word_count'] = df['text'].apply(
lambda x: len([w for w in x.lower().split() if w in stopwords.words('english')])
)

===== BAG OF WORDS =====

Method 3: Count Vectorizer (Bag of Words)

count_vectorizer = CountVectorizer(
max_features=100, # Top 100 most frequent words
min_df=1, # Minimum document frequency
max_df=0.8, # Maximum document frequency (remove very common words)
ngram_range=(1, 2) # Unigrams and bigrams
)
bow_features = count_vectorizer.fit_transform(documents_clean)

```
bow_feature_names = count_vectorizer.get_feature_names_out()
bow_df = pd.DataFrame(bow_features.toarray(), columns=bow_feature_names)
```

===== TF-IDF =====

Method 4: TF-IDF Vectorizer

```
tfidf_vectorizer = TfidfVectorizer(
max_features=100,
min_df=1,
max_df=0.8,
ngram_range=(1, 2),
sublinear_tf=True # Apply sublinear tf scaling
)
tfidf_features = tfidf_vectorizer.fit_transform(documents_clean)
tfidf_feature_names = tfidf_vectorizer.get_feature_names_out()
tfidf_df = pd.DataFrame(tfidf_features.toarray(), columns=tfidf_feature_names)
```

===== ADVANCED VECTORIZATION =====

Method 5: Hashing Vectorizer (for large vocabularies)

```
hash_vectorizer = HashingVectorizer(
n_features=2**10, # 1024 features
ngram_range=(1, 2)
)
hash_features = hash_vectorizer.transform(documents_clean)
```

Method 6: Word embeddings (using pre-trained models)

Using Gensim Word2Vec

```
from gensim.models import Word2Vec
```

Tokenize documents

```
tokenized_docs = [doc.split() for doc in documents_clean]
```

Train Word2Vec model

w2v_model = Word2Vec(
sentences=tokenized_docs,
vector_size=100,
window=5,
min_count=1,
workers=4
)

Get document vectors (average of word vectors)

def get_document_vector(doc, model):
words = doc.split()
word_vectors = [model.wv[word] for word in words if word in model.wv]
if word_vectors:
return np.mean(word_vectors, axis=0)
else:
return np.zeros(model.vector_size)

w2v_vectors = np.array([get_document_vector(doc, w2v_model) for doc in documents_clean])

Method 7: Using pre-trained embeddings (GloVe, fastText)

Example with sentence-transformers (BERT-based)

from sentence_transformers import SentenceTransformer

sbert_model = SentenceTransformer('all-MiniLM-L6-v2')
sbert_embeddings = sbert_model.encode(documents)

===== TOPIC MODELING =====

Method 8: Latent Dirichlet Allocation (LDA)

lda = LatentDirichletAllocation(n_components=3, random_state=42)
lda_features = lda.fit_transform(bow_features)

Method 9: Non-negative Matrix Factorization (NMF)

nmf = NMF(n_components=3, random_state=42)
nmf_features = nmf.fit_transform(tfidf_features)

===== LINGUISTIC FEATURES =====

Method 10: Part-of-speech tagging features

from nltk import pos_tag, word_tokenize

def pos_features(text):
tokens = word_tokenize(text)
pos_tags = pos_tag(tokens)

```
noun_count = sum(1 for word, tag in pos_tags if tag.startswith('NN'))
verb_count = sum(1 for word, tag in pos_tags if tag.startswith('VB'))
adj_count = sum(1 for word, tag in pos_tags if tag.startswith('JJ'))

return {
    'noun_count': noun_count,
    'verb_count': verb_count,
    'adj_count': adj_count,
    'noun_ratio': noun_count / len(tokens) if tokens else 0
}
```

pos_features_df = pd.DataFrame([pos_features(doc) for doc in documents])

Method 11: Sentiment features

from textblob import TextBlob

df['polarity'] = df['text'].apply(lambda x: TextBlob(x).sentiment.polarity)
df['subjectivity'] = df['text'].apply(lambda x: TextBlob(x).sentiment.subjectivity)

Method 12: Named Entity Recognition features

Using spaCy

```
import spacy

nlp = spacy.load('en_core_web_sm')

def ner_features(text):
doc = nlp(text)
entities = {
'person_count': sum(1 for ent in doc.ents if ent.label_ == 'PERSON'),
'org_count': sum(1 for ent in doc.ents if ent.label_ == 'ORG'),
'gpe_count': sum(1 for ent in doc.ents if ent.label_ == 'GPE'),
'total_entities': len(doc.ents)
}
return entities

ner_df = pd.DataFrame([ner_features(doc) for doc in documents])
```

Industry Best Practices:

- Always preprocess text (lowercase, remove special characters, etc.)
- Use TF-IDF over raw counts for most applications
- Leverage pre-trained embeddings (BERT, GPT) for state-of-the-art results
- Remove stop words unless doing sentiment analysis (negation words matter)
- Use n-grams (bigrams, trigrams) to capture context
- Apply dimensionality reduction after vectorization to reduce features
- Consider domain-specific features (e.g., medical terminology, legal jargon)
- Save fitted vectorizers for consistent production deployment

---

9. Time Series Feature Engineering

What is it?

Time series feature engineering creates features that capture temporal patterns, trends, seasonality, and autocorrelations in sequential data ordered by time.

Why is it important?

- Captures temporal dependencies and patterns
- Enables forecasting and trend analysis
- Reveals seasonal and cyclic behaviors
- Critical for time-dependent predictions
- Improves model accuracy for sequential data

How to implement:

```
import pandas as pd
import numpy as np
from datetime import datetime, timedelta
```

Sample time series data

```
dates = pd.date_range(start='2023-01-01', periods=365, freq='D')
df = pd.DataFrame({
'date': dates,
'sales': np.random.randint(100, 500, 365) + np.sin(np.arange(365) * 2 * np.pi / 365) * 50
})
df['date'] = pd.to_datetime(df['date'])
df = df.set_index('date')
```

===== DATE/TIME FEATURES =====

Method 1: Extract datetime components

```
df['year'] = df.index.year
df['month'] = df.index.month
df['day'] = df.index.day
df['dayofweek'] = df.index.dayofweek # Monday=0, Sunday=6
df['dayofyear'] = df.index.dayofyear
df['quarter'] = df.index.quarter
df['week'] = df.index.isocalendar().week
df['is_weekend'] = (df['dayofweek'] >= 5).astype(int)
df['is_month_start'] = df.index.is_month_start.astype(int)
df['is_month_end'] = df.index.is_month_end.astype(int)
df['is_quarter_start'] = df.index.is_quarter_start.astype(int)
```

Method 2: Cyclical encoding (for periodic features)

```
df['month_sin'] = np.sin(2 * np.pi * df['month'] / 12)
df['month_cos'] = np.cos(2 * np.pi * df['month'] / 12)
df['dayofweek_sin'] = np.sin(2 * np.pi * df['dayofweek'] / 7)
df['dayofweek_cos'] = np.cos(2 * np.pi * df['dayofweek'] / 7)
```

===== LAG FEATURES =====

Method 3: Lagged values

```
df['sales_lag_1'] = df['sales'].shift(1) # Previous day
df['sales_lag_7'] = df['sales'].shift(7) # Same day last week
df['sales_lag_30'] = df['sales'].shift(30) # Same day last month
df['sales_lag_365'] = df['sales'].shift(365) # Same day last year
```

Multiple lags

```
for lag in [1, 2, 3, 7, 14, 30]:
df[f'sales_lag_{lag}'] = df['sales'].shift(lag)
```

===== ROLLING WINDOW FEATURES =====

Method 4: Rolling statistics

```
df['sales_rolling_mean_7'] = df['sales'].rolling(window=7).mean()
df['sales_rolling_mean_30'] = df['sales'].rolling(window=30).mean()
df['sales_rolling_std_7'] = df['sales'].rolling(window=7).std()
df['sales_rolling_median_7'] = df['sales'].rolling(window=7).median()
df['sales_rolling_min_7'] = df['sales'].rolling(window=7).min()
df['sales_rolling_max_7'] = df['sales'].rolling(window=7).max()
```

Method 5: Rolling sum and count

```
df['sales_rolling_sum_7'] = df['sales'].rolling(window=7).sum()
df['sales_rolling_count_7'] = df['sales'].rolling(window=7).count()
```

===== EXPANDING WINDOW FEATURES =====

Method 6: Cumulative statistics

```
df['sales_cumsum'] = df['sales'].cumsum()
df['sales_cummean'] = df['sales'].expanding().mean()
df['sales_cummax'] = df['sales'].cummax()
df['sales_cummin'] = df['sales'].cummin()
```

===== EXPONENTIALLY WEIGHTED FEATURES =====

Method 7: Exponentially weighted moving average

df['sales_ewm_7'] = df['sales'].ewm(span=7, adjust=False).mean()
df['sales_ewm_30'] = df['sales'].ewm(span=30, adjust=False).mean()
df['sales_ewm_std_7'] = df['sales'].ewm(span=7, adjust=False).std()

===== DIFFERENCE FEATURES =====

Method 8: Differencing (for stationarity)

df['sales_diff_1'] = df['sales'].diff(1) # First difference
df['sales_diff_7'] = df['sales'].diff(7) # Weekly difference
df['sales_pct_change'] = df['sales'].pct_change() # Percentage change

===== COMPARISON FEATURES =====

Method 9: Comparing with historical values

df['sales_vs_lag7'] = df['sales'] - df['sales_lag_7']
df['sales_vs_rolling_mean'] = df['sales'] - df['sales_rolling_mean_7']
df['sales_ratio_lag7'] = df['sales'] / (df['sales_lag_7'] + 1)

===== SEASONAL FEATURES =====

Method 10: Season indicators

def get_season(month):
if month in [12, 1, 2]:
return 'winter'
elif month in [3, 4, 5]:
return 'spring'
elif month in [6, 7, 8]:
return 'summer'
else:
return 'fall'

```
df['season'] = df['month'].apply(get_season)
```

===== TREND FEATURES =====

Method 11: Time-based trend

```
df['time_index'] = np.arange(len(df))
df['time_index_squared'] = df['time_index'] ** 2
```

===== FOURIER FEATURES =====

Method 12: Fourier terms for seasonality

```
def create_fourier_features(df, period, order):
"""Create Fourier features for seasonality"""
for i in range(1, order + 1):
df[f'sin_{period}{i}'] = np.sin(2 * np.pi * i * df['dayofyear'] / period)
df[f'cos{period}_{i}'] = np.cos(2 * np.pi * i * df['dayofyear'] / period)
return df

df = create_fourier_features(df, period=365.25, order=3)
```

===== AUTOCORRELATION FEATURES =====

Method 13: Autocorrelation at different lags

```
from statsmodels.tsa.stattools import acf, pacf

acf_values = acf(df['sales'].dropna(), nlags=30)
pacf_values = pacf(df['sales'].dropna(), nlags=30)
```

Add significant lags as features

```
for lag in [7, 14, 30]:
df[f'acf_lag_{lag}'] = acf_values[lag]
```

===== HOLIDAY AND EVENT FEATURES =====

Method 14: Holiday indicators

from pandas.tseries.holiday import USFederalHolidayCalendar

cal = USFederalHolidayCalendar()
holidays = cal.holidays(start=df.index.min(), end=df.index.max())
df['is_holiday'] = df.index.isin(holidays).astype(int)

Days to/from holiday

df['days_to_holiday'] = (holidays.searchsorted(df.index) - np.arange(len(df))).clip(lower=0)
df['days_from_holiday'] = (np.arange(len(df)) - holidays.searchsorted(df.index, side='right')).clip(lower=0)

===== COMPLEX TIME FEATURES =====

Method 15: Time since last event

def time_since_event(series, condition):
"""Calculate time steps since last event"""
event_indices = series[condition].index
time_since = []

```
for idx in series.index:
    past_events = event_indices[event_indices < idx]
    if len(past_events) > 0:
        time_since.append((idx - past_events[-1]).days)
    else:
        time_since.append(-1)

return pd.Series(time_since, index=series.index)
```

df['days_since_peak'] = time_since_event(df['sales'], df['sales'] > df['sales'].quantile(0.9))

Clean up NaN values created by rolling operations

df = df.fillna(method='bfill').fillna(0)

Industry Best Practices:

- Use lag features appropriate to the data frequency (daily, hourly, etc.)
- Include multiple lag values to capture different temporal patterns
- Apply rolling window features to smooth noise
- Use cyclical encoding for periodic features (hour, day, month)
- Create difference features to achieve stationarity
- Add domain-specific temporal features (holidays, events, business cycles)
- Validate on future time periods, never shuffle time series data
- Handle missing values appropriately (forward fill, interpolation)

---

10. Domain-Specific Feature Engineering

What is it?

Domain-specific feature engineering leverages industry knowledge, business understanding, and subject matter expertise to create highly relevant features tailored to specific problem domains.

Why is it important?

- Incorporates valuable domain expertise
- Creates features directly aligned with business objectives
- Often provides the biggest performance gains
- Improves model interpretability for stakeholders
- Captures nuances that generic methods miss

Examples by Domain:

Finance/Banking:

Credit risk features

df['debt_to_income_ratio'] = df['total_debt'] / df['annual_income']
df['credit_utilization'] = df['credit_card_balance'] / df['credit_limit']
df['payment_to_income_ratio'] = df['monthly_payment'] / (df['monthly_income'] + 1)
df['has_bankruptcy'] = (df['bankruptcy_count'] > 0).astype(int)
df['account_age_years'] = (pd.to_datetime('today') - df['account_open_date']).dt.days / 365

Investment features

```
df['sharpe_ratio'] = (df['returns'] - df['risk_free_rate']) / df['volatility']
df['max_drawdown'] = df['portfolio_value'].cummax() - df['portfolio_value']
df['portfolio_turnover'] = df['trades_value'] / df['portfolio_value']
```

E-commerce/Retail:

Customer behavior features

```
df['avg_order_value'] = df['total_revenue'] / df['order_count']
df['days_since_last_purchase'] = (pd.to_datetime('today') - df['last_purchase_date']).dt.days
df['purchase_frequency'] = df['order_count'] / df['customer_lifetime_days']
df['cart_abandonment_rate'] = df['abandoned_carts'] / df['carts_created']
df['customer_lifetime_value'] = df['avg_order_value'] * df['purchase_frequency'] *
df['expected_lifetime']
```

Product features

```
df['discount_depth'] = (df['original_price'] - df['sale_price']) / df['original_price']
df['price_per_unit'] = df['price'] / df['quantity']
df['inventory_turnover'] = df['units_sold'] / df['avg_inventory']
```

Healthcare:

Patient risk features

```
df['bmi'] = df['weight_kg'] / (df['height_m'] ** 2)
df['bmi_category'] = pd.cut(df['bmi'], bins=[0, 18.5, 25, 30, 100],
labels=['underweight', 'normal', 'overweight', 'obese'])
df['age_bmi_interaction'] = df['age'] * df['bmi']
df['heart_rate_recovery'] = df['heart_rate_peak'] - df['heart_rate_resting']
df['blood_pressure_ratio'] = df['systolic_bp'] / df['diastolic_bp']
```

Treatment features

```
df['days_on_medication'] = (pd.to_datetime('today') - df['medication_start_date']).dt.days
df['readmission_within_30days'] = (df['days_to_readmission'] <= 30).astype(int)
```

Real Estate:

Property value features

```
df['price_per_sqft'] = df['price'] / df['square_feet']
df['bedroom_to_bathroom_ratio'] = df['bedrooms'] / (df['bathrooms'] + 0.5)
df['lot_coverage'] = df['building_area'] / df['lot_area']
df['age_of_property'] = 2025 - df['year_built']
df['renovated_recently'] = (df['years_since_renovation'] < 5).astype(int)
```

Location features

```
df['school_quality_weighted'] = df['avg_school_rating'] * (1 / (df['distance_to_school'] + 1))
df['walkability_score'] = (df['restaurants_nearby'] + df['shops_nearby'] +
df['parks_nearby']) / df['distance_to_city_center']
```

Marketing/Advertising:

Campaign performance features

```
df['click_through_rate'] = df['clicks'] / df['impressions']
df['conversion_rate'] = df['conversions'] / df['clicks']
df['cost_per_click'] = df['spend'] / df['clicks']
df['cost_per_acquisition'] = df['spend'] / df['conversions']
df['return_on_ad_spend'] = df['revenue'] / df['spend']
```

Engagement features

```
df['engagement_rate'] = (df['likes'] + df['comments'] + df['shares']) / df['reach']
df['avg_session_duration'] = df['total_time_on_site'] / df['sessions']
df['bounce_rate'] = df['single_page_sessions'] / df['total_sessions']
```

Manufacturing/IoT:

Equipment health features

```
df['vibration_change_rate'] = df['vibration'].diff() / df['time'].diff()
df['temperature_deviation'] = df['temperature'] - df['normal_temperature']
df['cycles_until_maintenance'] = df['maintenance_interval'] -
df['cycles_since_maintenance']
df['efficiency_ratio'] = df['actual_output'] / df['expected_output']
df['downtime_percentage'] = df['downtime_hours'] / df['total_hours']
```

Quality features

df['defect_rate'] = df['defective_units'] / df['total_units']
df['mean_time_between_failures'] = df['operating_hours'] / df['failure_count']

Industry Best Practices:

- Collaborate closely with domain experts and stakeholders
- Validate features against business logic and real-world constraints
- Create features that align with known causal relationships
- Document the business rationale for each feature
- Test features against regulatory and compliance requirements
- Use industry-standard metrics and KPIs as features
- Consider temporal context (features may need updating)
- Balance sophistication with interpretability for stakeholders

---

Building a Complete Feature Engineering Pipeline

Here's an end-to-end pipeline combining multiple techniques:

import pandas as pd
import numpy as np
from sklearn.pipeline import Pipeline, FeatureUnion
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer
from sklearn.base import BaseEstimator, TransformerMixin

Custom transformer for feature creation

class FeatureCreator(BaseEstimator, TransformerMixin):
def **init**(self):
pass

```
def fit(self, X, y=None):
    return self

def transform(self, X):
    X = X.copy()

    # Create interaction features
    X['area_per_bedroom'] = X['area'] / (X['bedrooms'] + 1)
    X['income_per_person'] = X['income'] / (X['household_size'] + 1)

    # Create polynomial features
```

```
    X['area_squared'] = X['area'] ** 2

    # Create binned features
    X['age_bin'] = pd.cut(X['age'], bins=[0, 30, 50, 100], labels=[0, 1, 2])

    return X
```

Define column types

numeric_features = ['area', 'bedrooms', 'age', 'income', 'household_size']
categorical_features = ['city', 'property_type']

Create preprocessing pipelines

numeric_transformer = Pipeline(steps=[
('imputer', SimpleImputer(strategy='median')),
('scaler', StandardScaler())
])

categorical_transformer = Pipeline(steps=[
('imputer', SimpleImputer(strategy='constant', fill_value='missing')),
('onehot', OneHotEncoder(drop='first', handle_unknown='ignore'))
])

Combine preprocessing steps

preprocessor = ColumnTransformer(
transformers=[
('num', numeric_transformer, numeric_features),
('cat', categorical_transformer, categorical_features)
])

Complete pipeline

feature_pipeline = Pipeline(steps=[
('feature_creator', FeatureCreator()),
('preprocessor', preprocessor)
])

Use the pipeline

X_transformed = feature_pipeline.fit_transform(df)

Save pipeline for production

import joblib
joblib.dump(feature_pipeline, 'feature_pipeline.pkl')

Load and use in production

loaded_pipeline = joblib.load('feature_pipeline.pkl')
new_data_transformed = loaded_pipeline.transform(new_data)

Feature Engineering Checklist

1. **Understand the problem** - Define clear objectives and success metrics
2. **Perform EDA** - Analyze distributions, correlations, missing patterns
3. **Handle missing data** - Use appropriate imputation strategies
4. **Encode categorical variables** - Choose encoding based on cardinality and model
5. **Scale numeric features** - Standardize or normalize based on algorithm requirements
6. **Create domain features** - Leverage business knowledge
7. **Engineer interactions** - Capture feature relationships
8. **Add temporal features** - For time-series data
9. **Transform skewed features** - Apply log, square root, or Box-Cox
10. **Select features** - Remove redundant and irrelevant features
11. **Validate** - Test feature impact on model performance
12. **Document** - Maintain clear documentation for reproducibility
13. **Automate** - Build reusable pipelines for production

Industry Tools and Libraries

Tool/Library	Purpose	Use Case
Pandas	Data manipulation	Data transformation, aggregation
Scikit-learn	ML preprocessing	Scaling, encoding, pipelines
Feature-engine	Feature engineering	Automated transformations
Category Encoders	Categorical encoding	Advanced encoding methods
TPOT / AutoML	Automated FE	Discovering optimal features
Featuretools	Automated FE	Deep feature synthesis
tsfresh	Time series FE	Automated time series features
Boruta	Feature selection	Wrapper-based selection

Table 3: Popular tools for feature engineering

Key Takeaways for Client Meetings

1. **Feature engineering is often more impactful than algorithm choice** - invest time here first
2. **Domain knowledge is invaluable** - collaborate with subject matter experts
3. **Automate and document** - build reproducible, maintainable pipelines
4. **Iterate based on model performance** - feature engineering is not one-time
5. **Balance complexity with interpretability** - stakeholders need to understand features
6. **Scale matters** - ensure pipelines work efficiently on production data volumes
7. **Monitor feature distributions** - data drift affects feature relevance over time
8. **Version control features** - track feature definitions and transformations

---

Conclusion

Feature engineering is both an art and a science—combining statistical techniques, domain expertise, and creative problem-solving. Mastering these 10 industry-level techniques will significantly improve your machine learning models' performance, interpretability, and business impact.

The most successful ML practitioners spend significant time understanding their data, creating meaningful features, and iterating based on results. Automated feature engineering tools can accelerate this process, but human insight and domain knowledge remain irreplaceable for creating truly powerful features.

By systematically applying these techniques and building robust, reproducible pipelines, you position your ML projects for long-term success in production environments.

Further Resources and References

- Scikit-learn Documentation: https://scikit-learn.org/stable/modules/preprocessing.html
- Feature Engineering for Machine Learning (Domino Data Lab, 2024)
- Top 10 Feature Engineering Methods (Academic Nest Hub, 2024)
- Feature Engineering Tutorial (DataCamp, 2025)
- 8 Feature Engineering Techniques (ProjectPro, 2024)
- Feature Selection Techniques (GeeksforGeeks, 2021)
- What is Feature Engineering? (IBM Think, 2024)