UNIVERSIDAD AMERICANA
Facultad de Ingeniería y Arquitectura

Inteligencia de negocios

Preprocesamiento y transformación de datos

Estudiante:
Carlos Diego Toruño Espinales

Docente:
Arlen Jeannette Lopez

Fecha:

16 de Septiembre 2024

Introducción

El objetivo del preprocesamiento de datos es limpiar, transformar y mejorar la calidad de los datos, garantizando que estén en un formato adecuado para el análisis y modelado. En este caso el dataset contiene variables necesarias de tratamiento, incluyendo la imputación de valores, detección y eliminación de outliers y normalización de variables para el correcto análisis posterior.

Limpieza de datos

- **Exploración inicial del conjunto de datos**

Como primer paso del análisis para el procesamiento y transformación se realiza una exploración inicial del conjunto de datos, en este se muestran las primeras columnas para darnos una idea de cómo está estructurada la información, el tipo y número de variables y una breve descripción de cada una de ellas.

```
data = pd.read_csv('train.csv')

# Exploración inicial del conjunto de datos
print(data.head())
print(data.shape)
print(data.info())
print(data.describe())
```

- **Identificación de variables relevantes**

Debido a la cantidad de columnas que posee el conjunto de datos, es necesario reducir las opciones para poder realizar un análisis del negocio más acertado. En este caso se han escogido las siguientes columnas para la exploración.

```
# Seleccionar las variables importantes para el análisis
selected_columns = ['SalePrice', 'OverallQual', 'GrLivArea', 'GarageCars',
                    'TotalBsmtSF', 'YearBuilt', 'FullBath', 'LotArea', 'MoSold', 'YrSold']
```

Una vez tenemos nuestras columnas identificadas, procedemos a mostrar las variables que contienen valores vacíos y estos pueden ser mostrados como un porcentaje para mirar de mejor manera la cantidad que hace falta con respecto al total.

```
# Identificar todas las variables que contienen valores faltantes
print(data.columns[data.isnull().any()])
print('\n Proporcion en % \n')
print(data.isnull().sum()/data.shape[0]*100)
```

- **Manejo de valores faltantes**

Para lidiar con los valores faltantes tenemos varias opciones que podemos aplicar, como por ejemplo utilizar la media, mediana, moda o un modelo predictivo. En este caso hemos dividido las variables en dos categorías, la primera de estas son las columnas numéricas a las que les aplicaremos la mediana. En este caso se ha decidido manejar los valores numéricos con esta estrategia debido a que la mediana es menos susceptible a los valores atípicos (outliers). Los otros datos son los que representan una categoría, y estos serán trabajados con la moda, debido a que se reemplazan los valores faltantes con la categoría que más se frecuenta.

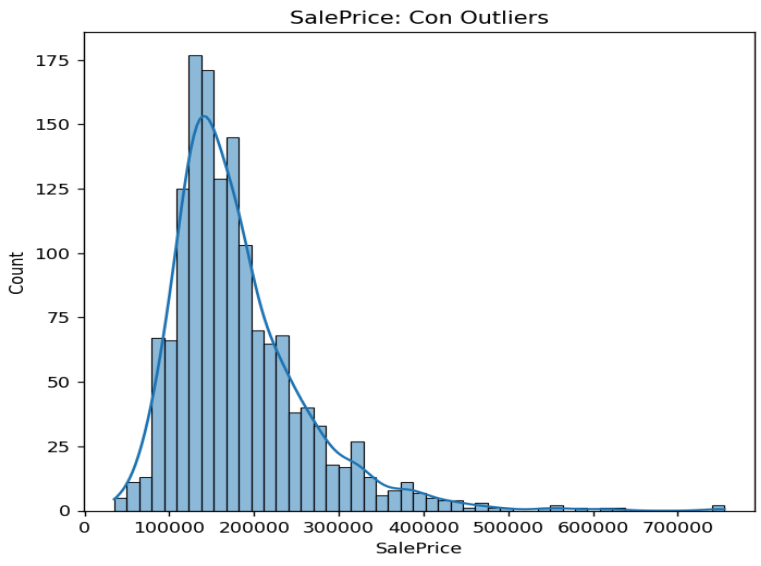
```
# Imputar valores faltantes para las columnas numéricas con la mediana
data['SalePrice'].fillna(data['SalePrice'].median(), inplace=True)
data['OverallQual'].fillna(data['OverallQual'].median(), inplace=True)
data['GrLivArea'].fillna(data['GrLivArea'].median(), inplace=True)
data['GarageCars'].fillna(data['GarageCars'].median(), inplace=True)
data['TotalBsmtSF'].fillna(data['TotalBsmtSF'].median(), inplace=True)
data['YearBuilt'].fillna(data['YearBuilt'].median(), inplace=True)
data['FullBath'].fillna(data['FullBath'].median(), inplace=True)
data['LotArea'].fillna(data['LotArea'].median(), inplace=True)
data['MoSold'].fillna(data['MoSold'].median(), inplace=True)
data['YrSold'].fillna(data['YrSold'].median(), inplace=True)
```

```
# Imputar valores faltantes para las columnas categóricas con la moda
data['BsmtQual'].fillna(data['BsmtQual'].mode()[0], inplace=True)
data['KitchenQual'].fillna(data['KitchenQual'].mode()[0], inplace=True)
data['Neighborhood'].fillna(data['Neighborhood'].mode()[0], inplace=True)
```

- **Detección de valores atípicos**

Una vez nos encargamos de los valores vacíos, existe otra preocupación de la cual debemos encargarnos, estos son los valores atípicos, los cuales representan un valor numérico distante del resto del conjunto y pueden ser críticos en el análisis de datos. Para detectar valores atípicos utilizaremos gráficos para observar las inconsistencias con respecto a los datos.

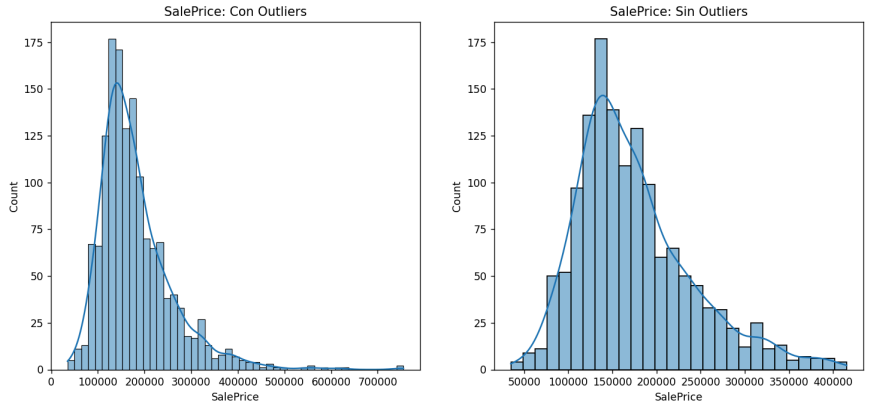
De ejemplo utilizaremos la columna **SalePrice**, la cual representa el valor con el cual se venden las casas en este conjunto de datos.



Como podemos observar en la gráfica la columna **SalePrice** muestra precios de pocas casas las cuales están muy alejados de los valores principales de los demás datos. En este caso para eliminar los valores atípicos de las columnas utilizaremos Z-Core ya que este mide cuántas desviaciones estándar están un valor por encima o por debajo de la media de los valores numéricos. En este código definimos un umbral de 3 donde definimos que una desviación estándar superior a 3 o inferior a -3 mostraría un valor muy alejado con respecto a la media de los demás valores.

```
# Identificar y eliminar los outliers usando Z-score
threshold = 3
outliers_condition = (np.abs(stats.zscore(data[numerical_columns])) < threshold).all(axis=1)
```

- **Comparación de la columna SalePrice antes y después de los outliers**



Como podemos observar se ha reducido el rango de los precios dejando una media más cercana con respecto a la media del conjunto de datos.

- **Transformación de variables**

Otra forma de manejar variables es la normalización (Min-Max Scaling), la cual es una técnica de normalización donde se escalan los datos dentro de un rango especificado con el propósito de hacer un conjunto de datos comparable. Esta se puede utilizar para modelos de machine learning y redes neuronales. La principal diferencia que existe con Z-Core es que esta es menos intuitiva, ya que pueden existir valores negativos y no están dentro de un rango definido. Para utilizar **Min-Max Scaling** necesitamos la librería **sklearn.preprocessing**

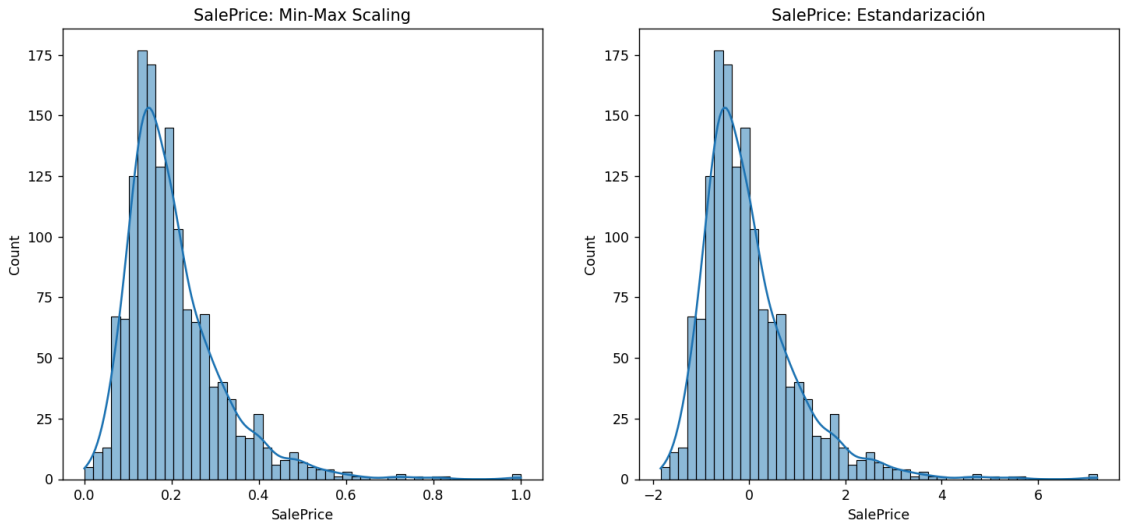
```
# Aplicar Min-Max Scaling y Z-score Estandarización
scaler_minmax = MinMaxScaler()
scaler_standard = StandardScaler()
```

En esta línea de código se inicia una instancia MinMaxScaler que reescala los valores de las columnas numéricas dentro de un rango especificado, por defecto este es entre 0 y 1.

Después de inicializar la instancia se transforman los datos según los parámetros y se guardan en un nuevo dataframe de pandas para facilitar su visualización en un gráfico posteriormente.

```
# Aplicar Min-Max Scaling a las columnas numéricas
minmax_scaled_data = scaler_minmax.fit_transform(data[numerical_columns])
minmax_scaled_df = pd.DataFrame(minmax_scaled_data, columns=numerical_columns)
```

A continuación se muestra un ejemplo de la diferencia entre estas dos técnicas en el conjunto de datos con la columna **SalePrice**.



- **Transformación Logarítmica**

Luego de normalizar y estandarizar las variables numéricas buscaremos por variables que presenten una asimetría o una distribución sesgada. Existen dos tipos de distribuciones, positivas y negativas. Cada una de estas indica valores extremos en cada extremo, ya sea positivo o negativo. A las variables que presentan asimetría se les aplicará una transformación logarítmica para que se ajusten a una mejor distribución. Esto lo haremos con la siguiente función:

En este código para cada columna seleccionada se define una función para calcular la asimetría con el paquete SciPy y se obtiene el sesgo por cada columna.

```
# Evaluar el sesgo de las columnas numéricas
skewness = data[selected_columns].apply(lambda x: stats.skew(x))
print(f'\nSesgo de las columnas seleccionadas:\n{skewness}')
```

```
Sesgo de las columnas seleccionadas:
SalePrice       1.880941
OverallQual     0.216721
GrLivArea       1.365156
GarageCars     -0.342197
TotalBsmtSF     1.522688
YearBuilt      -0.612831
FullBath        0.036524
LotArea        12.195142
MoSold          0.211835
YrSold          0.096170
```

Después se evalúa el sesgo de cada columna, y se seleccionan las que poseen uno mayor a 0.5 o menor a -0.5. Esto lo podemos hacer con la siguiente función:

```
# Identificar las columnas con sesgo alto (skewness > 0.5 o skewness < -0.5)
skewed_columns = skewness[skewness.abs() > 0.5].index
print(f'\nColumnas con sesgo alto:\n{skewed_columns}')
```

Luego de identificar las columnas con un sesgo alto, aplicamos la transformación logarítmica de la siguiente manera:

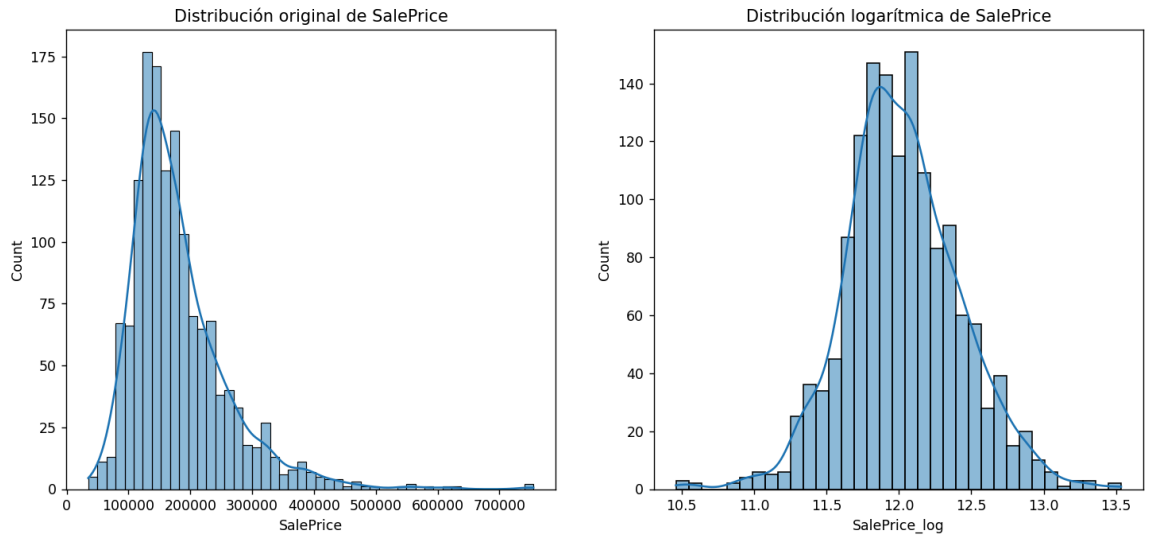
```
# Aplicar la transformación logarítmica a las columnas sesgadas
for col in skewed_columns:
    data[col + '_log'] = np.log1p(data[col])  # log1p maneja ceros
```

En este bucle que recorre cada columna con un sesgo significativo y por cada columna se crea su versión _log, la cual es una copia que representa la transformación de la variable, y con la función log1p, equivalente a **log (1+x)**, se aplica la transformación logarítmica. Se utiliza esta función en lugar de **log(x)** para manejar de manera más eficiente los valores cercanos a 0.

Podemos ver el resultado de las columnas con sesgo alto luego de la transformación:

```
Sesgo de las columnas seleccionadas después de la transformación:
SalePrice_log      0.121222
GrLivArea_log     -0.006134
TotalBsmtSF_log   -5.149373
YearBuilt_log     -0.640470
LotArea_log       -0.137263
```

- **Gráfico comparativo entre distribución normal y logarítmica con la columna SalePrice**



En el gráfico podemos observar que al aplicar la transformación logarítmica la distribución de datos se vuelve mucho más simétrica y más parecida a una distribución normal.

- **Ingeniería de características**

Con variables existentes de nuestro conjunto de datos podemos hacer la creación de nuevas variables, combinando previas o transformando las que ya existen. Por ejemplo con nuestro conjunto de datos podemos crear la variable **House Age**, la cual es el resultado de la resta de las columnas **YrSold** y **YearBuilt** y representa el tiempo que pasó hasta que una casa se vendió.

```
# Creacion de la variable House_Age
data['House_Age'] = data['YrSold'] - data['YearBuilt']
print("\n Variable House Age \n")
print(data[['House_Age']].head())
```

Otra variable que puede ser creada es **Total_Outside_Area** la cual es la suma de **OpenPorchSF + EnclosedPorch + 3SsnPorch + ScreenPorch** y representa el área total de la parte exterior de la casa.

```
# Creacion de la variable TotalOusideArea
data['Total_Outside_Area'] = data['OpenPorchSF'] + data['EnclosedPorch'] + data['3SsnPorch'] + data['ScreenPorch']
print("\n Variable Total Outside Area \n")
print(data[['Total_Outside_Area']].head())
```

- **Codificación de variables categóricas**

La codificación de variables es el proceso de transformar variables categóricas (es decir, variables que contienen categorías o etiquetas, como "bajo", "medio", "alto") en un formato numérico que pueda ser interpretado por los algoritmos de aprendizaje.

Para el ejemplo de este conjunto de datos usaremos las dos variables que hemos creado: **House Age** y **Total_Outside_Area**. Para estas dos variables utilizaremos Label Encoding, la cual es una técnica simple de codificación donde cada valor único es asignado a un número entero.

Para codificar las variables utilizaremos este código:

```
# Crear el codificador
label_encoder = LabelEncoder()

# Codificar 'House_Age_Category'
data['House_Age_Category'] = pd.cut(data['House_Age'], bins=[0, 10, 30, 100],
                                    labels=['Nuevo', 'Moderno', 'Antiguo'], include_lowest=True)
data['House_Age_Category_Encoded'] = label_encoder.fit_transform(data['House_Age_Category'])
```

En este código inicializamos una instancia LabelEncoder, este es un codificador de etiquetas y es parte de la librería **sklearn.** Creamos la columna **House_Age_Category** a partir de **House_Age** que contiene los años de antigüedad de la casa y se crean tres rangos para clasificar las casas:

0 a 10 años → ‘Nuevo’
10 a 30 años → ’Moderno’
30 a 100 años → ’Antiguo’

Podemos ver la salida de estos imprimiendo un .**head()** de la columna y la versión codificada

	House_Age_Category
0	Nuevo
1	Antiguo
2	Nuevo
3	Antiguo
4	Nuevo

	House_Age_Category_Encoded
0	2
1	0
2	2
3	0
4	2

De igual manera lo realizamos con la variable **Total_Outside_Area_Category**.

```
# Codificar 'Total_Outside_Area_Category'
data['Total_Outside_Area_Category'] = pd.cut(data['Total_Outside_Area'], bins=[0, 100, 300, 1000],
                                             labels=['Pequeño', 'Mediano', 'Grande'], include_lowest=True)
data['Total_Outside_Area_Category_Encoded'] = label_encoder.fit_transform(data['Total_Outside_Area_Category'])
```

En esta variable creamos categorías para el área total de la parte exterior de la casa, estas etiquetas están distribuidas de la siguiente manera:

0 a 100 pies cuadrados→ ‘Pequeño’
100 a 300 pies cuadrados→ ‘Mediano’
300 a 1000 pies cuadrados→ ‘Grande’

Podemos observar la salida de de la variable de igual manera con un .**head()**:

	Total_Outside_Area_Category
0	Pequeño
1	Pequeño
2	Pequeño
3	Grande
4	Pequeño

	Total_Outside_Area_Category_Encoded
0	2
1	2
2	2
3	0
4	2

- **Comparación antes y después del conjunto de datos**

Estado del conjunto de datos antes:
Antes del preprocesamiento, el conjunto de datos presentaba varias irregularidades que dificultan su análisis y modelado. Las distribuciones de muchas variables presentaban asimetrías , con una alta presencia de outliers en las variables seleccionadas que distorsionan la representación central de los datos. Además, existían valores faltantes en diferentes columnas, lo que incrementa la posibilidad de obtener resultados erróneos o inconsistentes.

Estado del conjunto de datos después:
Después del preprocesamiento, se observó una mejora considerable en la calidad y estructura del conjunto de datos. La eliminación de outliers y la transformación logarítmica redujeron la asimetría y normalizaron las distribuciones, lo que facilitó un análisis más preciso. Asimismo, la imputación de valores faltantes aseguró que la información crítica no se perdiera y creó una base sólida para su análisis en profundidad.