Kaggle Challenge - Data Preprocessing

WE'LL COVER THE FOLLOWING

2. Data Preprocessing - Prepare the Data for Machine Learning Algorithms

We took our notes in the exploratory phase, now it's time to act on them and prepare our data for the machine learning algorithms. Instead of just doing this manually, we will also learn how to write functions where possible.

Deal With Missing Values

Let's get a sorted count of the missing values for all the attributes.

```
housing.isnull().sum().sort_values(ascending=False)
```

```
In [34]: housing.isnull().sum().sort_values(ascending=False)
```

```
Out[34]: PoolQC          1453
         MiscFeature     1406
         Alley           1369
         Fence           1179
         FireplaceQu      690
         LotFrontage      259
         GarageType        81
         GarageCond        81
         GarageFinish      81
         GarageQual        81
         GarageYrBlt       81
         BsmtFinType2      38
         BsmtExposure      38
         BsmtQual          37
         BsmtCond          37
         BsmtFinType1      37
         MasVnrArea         8
         MasVnrType         8
         Electrical         1
         RoofMatl           0
         Exterior1st        0
         RoofStyle          0
         ExterQual          0
         Exterior2nd        0
         YearBuilt          0
         ExterCond          0
         Foundation         0
         YearRemodAdd       0
         SalePrice          0
         OverallCond        0
                         ...
         GarageArea         0
         PavedDrive         0
         WoodDeckSF         0
         OpenPorchSF        0
         3SsnPorch          0
         BsmtUnfSF          0
         ScreenPorch        0
         PoolArea           0
         MiscVal            0
         MoSold             0
```

From the results above we can assume that *PoolQC* to *Bsmt* attributes are missing for the houses that do not have these facilities (houses without pools, basements, garage etc.). Therefore, the missing values could be filled in with “None”. *MasVnrType* and *MasVnrArea* both have 8 missing values, likely houses without masonry veneer.

What should we do with all this missing data?

Most machine learning algorithms cannot work with missing features, so we need to take care of them. Essentially, we have three options:

- Get rid of the corresponding houses.
- Get rid of the whole attribute or remove the whole column.

- Set the missing values to some value (zero, the mean, the median, etc.).

We can accomplish these easily using DataFrame's `dropna()`, `drop()`, and `fillna()` methods.

> 📌**Note**: Whenever you choose the third option, say imputing values using the median, you should compute the median value on the training set, and use it to fill the missing values in the training set. But you should also remember to later replace missing values in the test set **using the same median value** when you want to evaluate your system, and also once the model gets deployed to replace missing values in new unseen data.

We are going to apply different approaches to fix our missing values, so that we can various approaches in action:

- We are going to replace values for categorical attributes with *None*.
- For *LotFrontage*, we are going to go a bit fancy and compute the median *LotFrontage* for all the houses in the same neighborhood, instead of the plain median for the entire column, and use that to impute on a neighborhood by neighborhood basis.
- We are going to replace missing values for most of the numerical columns with zero and one with the mode.
- We are going to drop one non-interesting column, *Utilities*.

Right now, we are going to look at how to do these fixes by explicitly writing the name of the column in the code. Later, in the upcoming section on transformation pipelines, we will learn how to handle them in an automated manner as well.

```
# Imputing Missing Values

housing_processed = housing

# Categorical columns:
cat_cols_fill_none = ['PoolQC', 'MiscFeature', 'Alley', 'Fence', 'FireplaceQu',
                      'GarageCond', 'GarageQual', 'GarageFinish', 'GarageType',
                      'BsmtFinType2', 'BsmtExposure', 'BsmtFinType1', 'BsmtQual', 'BsmtCond',
                      'MasVnrType']

# Replace missing values for categorical columns with None
```

```
for cat in cat_cols_fill_none:
    housing_processed[cat] = housing_processed[cat].fillna("None")


# Group by neighborhood and fill in missing value by the median LotFrontage of all the neighb
housing_processed['LotFrontage'] = housing_processed.groupby("Neighborhood")["LotFrontage"].t
    lambda x: x.fillna(x.median()))

# Garage: GarageYrBlt, GarageArea and GarageCars these are numerical columns, replace with ze
for col in ['GarageYrBlt', 'GarageArea', 'GarageCars']:
    housing_processed[col] = housing_processed[col].fillna(int(0))

# MasVnrArea : replace with zero
housing_processed['MasVnrArea'] = housing_processed['MasVnrArea'].fillna(int(0))

# Use the mode value
housing_processed['Electrical'] = housing_processed['Electrical'].fillna(housing_processed['E

# There is no need of Utilities so let's just drop this column
housing_processed = housing_processed.drop(['Utilities'], axis=1)

# Get the count again to verify that we do not have any more missing values
housing_processed.isnull().apply(sum).max()
```

```
In [15]:  # Imputing Missing Values

housing_processed = housing

# Categorical columns:
cat_cols_fill_none = ['PoolQC', 'MiscFeature', 'Alley', 'Fence', 'FireplaceQu',
                      'GarageCond', 'GarageQual', 'GarageFinish', 'GarageType',
                      'BsmtFinType2', 'BsmtExposure', 'BsmtFinType1', 'BsmtQual', 'BsmtCond',
                      'MasVnrType']

# Replace missing values for categorical columns with None
for cat in cat_cols_fill_none:
    housing_processed[cat] = housing_processed[cat].fillna("None")

# Group by neighborhood and fill in missing value by the median LotFrontage of all the neighborhood
housing_processed['LotFrontage'] = housing_processed.groupby("Neighborhood")["LotFrontage"].transform(
    lambda x: x.fillna(x.median()))

# GarageYrBlt, GarageArea and GarageCars these are numerical columns, replace with zero
for col in ['GarageYrBlt', 'GarageArea', 'GarageCars']:
    housing_processed[col] = housing_processed[col].fillna(int(0))

#MasVnrArea : replace with zero
housing_processed['MasVnrArea'] = housing_processed['MasVnrArea'].fillna(int(0))

#Use the mode value
housing_processed['Electrical'] = housing_processed['Electrical'].fillna(housing_processed['Electrical']).mode()[0]

#There is no need of Utilities so let's just drop this column
housing_processed = housing_processed.drop(['Utilities'], axis=1)

In [16]:  # Get the count again to verify that we do not have any more missing values
housing_processed.isnull().apply(sum).max()

Out[16]: 0
```

Deal With Outliers

To remove noisy data, we are going to remove houses where we have some attribute that is above the 0.999 quantile, highly abnormal datapoint. We can do this by invoking the `quantile()` method on the DataFrame and then filtering based on the knowledge of the quantiles for each attribute, like so:

```
num_attributes = housing_processed.select_dtypes(exclude='object')

high_quant = housing_processed.quantile(.999)
```

```
for i in num_attributes.columns:
    housing_processed = housing_processed.drop(housing_processed[i][housing_processed[i]>high

housing_processed.info()
```

```
In [37]:  num_attributes = housing_processed.select_dtypes(exclude='object')

          high_quant = housing_processed.quantile(.999)

          for i in num_attributes.columns:
              housing_processed = housing_processed.drop(housing_processed[i][housing_processed[i]>high_quant[i]].index)

          housing_processed.info()

          <class 'pandas.core.frame.DataFrame'>
          Int64Index: 1422 entries, 0 to 1458
          Data columns (total 79 columns):
          MSSubClass       1422 non-null int64
          MSZoning         1422 non-null object
          LotFrontage      1422 non-null float64
          LotArea          1422 non-null int64
          Street           1422 non-null object
          Alley            1422 non-null object
          LotShape         1422 non-null object
          LandContour      1422 non-null object
          LotConfig        1422 non-null object
          LandSlope        1422 non-null object
          Neighborhood     1422 non-null object
          Condition1       1422 non-null object
          Condition2       1422 non-null object
          BldgType         1422 non-null object
          HouseStyle       1422 non-null object
          OverallQual      1422 non-null int64
```

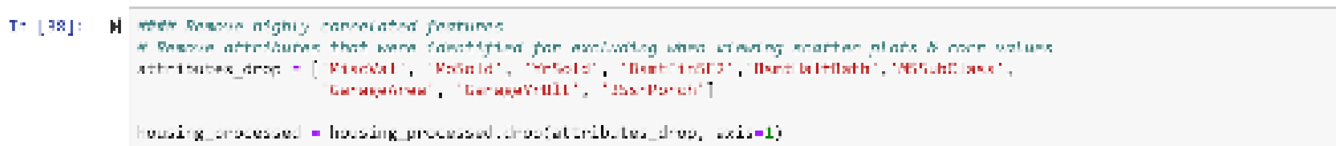
Invoking the *info()* method on the updated DataFrame tells us that we are left with 1422 rows now.

Deal With Correlated Attributes

Using highly-correlated features when creating machine learning models can impact performance negatively. As we saw in the numerical analysis section, we have quite a few correlated attributes. For example, we concluded that we can drop *GarageArea* because it is highly correlated with *GarageCars* and the reason for preferring *GarageCars* is because it is more correlated with price than area. (Pull out your notes from exploratory analysis at this step.)

```
#### Remove highly correlated features
# Remove attributes that were identified for excluding when viewing scatter plots & corr valu
attributes_drop = ['MiscVal', 'MoSold', 'YrSold', 'BsmtFinSF2','BsmtHalfBath','MSSubClass',
                   'GarageArea', 'GarageYrBlt', '3SsnPorch']

housing_processed = housing_processed.drop(attributes_drop, axis=1)
```



```
In [38]:  #### Remove highly correlated features
          # Remove attributes that were identified for excluding when viewing scatter plots & corr values
          attributes_drop = ['MiscVal', 'MoSold', 'YrSold', 'BsmtFinSF2','BsmtHalfBath','MSSubClass',
                             'GarageArea', 'GarageYrBlt', '3SsnPorch']

          housing_processed = housing_processed.drop(attributes_drop, axis=1)
```

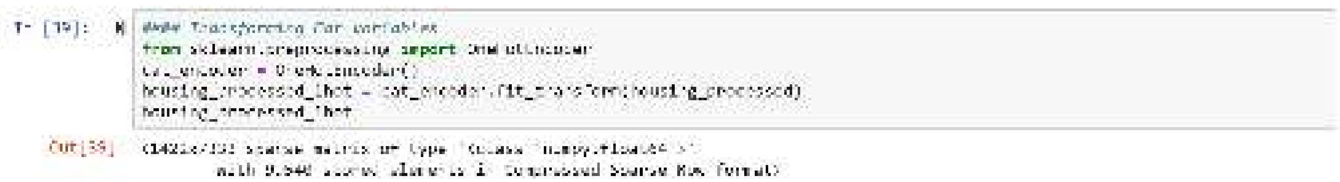
Handle Text And Categorical Attributes

Most Machine Learning algorithms need numbers as input, so let's convert all the categories from text to numbers.

A common approach to deal with textual data is to create one binary attribute for each category of the feature: for example, for type of houses, we would have one attribute equal to 1 when the category is *1Story* (and 0 otherwise), another attribute equal to 1 when the category is *2Story* (and 0 otherwise), and so on. This is called **one-hot encoding**, because only one attribute will be equal to 1 (hot), while the others will be 0 (cold). The new attributes are also known as ***dummy attributes***. Scikit-Learn provides a `OneHotEncoder` class to convert categorical values into one-hot vectors:

Scikit-Learn *is the most widely used library for working on machine learning/data science projects. It is simple, easy to use and it provides many efficient tools for data mining, data analysis and modeling. In short, it is awesome!*

```
#### Transforming Cat variables
from sklearn.preprocessing import OneHotEncoder
cat_encoder = OneHotEncoder()
housing_processed_1hot = cat_encoder.fit_transform(housing_processed)
housing_processed_1hot
```



Notice that as a result of creating new one-hot attributes our total number of attributes has jumped to 7333! We have a 1422x7333 matrix which is mostly sparse (zeros).

Feature Scaling

Feature Scaling is one of the most important transformations we need to apply to our data. As we said earlier, machine learning algorithms mostly do not perform well if they are fed numerical attributes with very different scales as input. This is the case for the housing data. If you go back and look at the distribution plots that we created in the very beginning, we notice that

LotArea ranges from 0 to 200000, while *GarageCars* ranges only from 0 to 4.

There are two common ways to get all attributes to have the same scale: min-max scaling and standardization.

- **Min-max scaling** (also known as normalization): this is a simple technique. Values are shifted and rescaled so that they end up ranging from 0 to 1. This can be done by subtracting the min value and dividing by the max minus the min, but fortunately Scikit-Learn provides a transformer (we will talk about transformers in a bit) called `MinMaxScaler` to do this in a hassle-free manner. This transformer also provides the *feature_range* hyperparameter so that we can change the range if for some reason we don't want the 0 to 1 scale.

$$X_{sc} = \frac{(X - X_{min})}{X_{max} - X_{min}}$$

- **Standardization**: this is a more sophisticated approach. Remember the lessons from statistics? Standardization is done by first subtracting the mean value (so standardized values always have a 0 mean), and then dividing by the standard deviation so that the resulting distribution has unit variance. Since it only cares about "fixing" the mean and variance, standardization does not limit values to a specific range, which may be problematic for some algorithms (e.g., neural networks often expect an input value ranging from 0 to 1). However, standardization is much less affected by outliers. Say Bill Gates walks into a bar, suddenly the median income for people in the bar would shoot up to the moon, so min-max scaling would be a poor choice for scaling here. On the other hand, standardization would not be much affected. Scikit-Learn provides a transformer called `StandardScaler` for standardization.

$$Z = \frac{x - \mu}{\sigma}$$

Instead of applying these scaling transformations on a column-by-column basis like we have been handling data preparation so far, in the next lesson, we are going to understand how to use transformation pipelines in order to do all this work in a more automated and cleaner fashion.

Jupyter Notebook

Jupyter Notebook

You can see the instructions running in the Jupyter Notebook below:

> **How to Use a Jupyter NoteBook?**
>
> - Click on **"Click to Launch"** button to work and see the code running live in the notebook.
> - You can click to open the **Jupyter Notebook in a new tab**.
> - Go to files and click *Download as* and then choose the format of the file to **download** . You can choose Notebook(.ipynb) to download the file and work locally or on your personal Jupyter Notebook.
> - The notebook **session expires after 30 minutes of inactivity**. It will reset if there is no interaction with the notebook for 30 consecutive minutes.

Your app can be found at: https://1dgnrwwoynk5m-live-app.educative.run/notebooks/DataPreprocessing.ipynb