data_cleaning

February 18, 2019

1 Data Cleaning

This document describes the process by which raw data (particuarly the features) is cleaned and exceptional entries handled.

1.1 Dependency and data import

Below is the code that imported raw data using `pd.read_csv()`.

```
import pandas as pd
import numpy as np
from mxnet import nd
from preprocessing import *

train = pd.read_csv("./data/raw/kaggle_house_pred_train.csv")
test_features = pd.read_csv("./data/raw/kaggle_house_pred_test.csv")

# Remove response variable from train
train_features = train[list(train)[:-1]]
# Concatenate the feature matrices
all_features = pd.concat([train_features, test_features], axis=0)
all_features = all_features.reset_index(drop=True)
# To help with easier do, replace all NULL value by np.nan
all_features = all_features.fillna(np.nan)
```

Note that the last line unified all NULL values to `np.nan`, whereas previously some NULL values are not.

1.2 MSSubClass

`MSSubClass` is read by `pd.read_csv()` as an integer column. However, by the description on Kaggle, this is actually a numerically coded categorical variable. Therefore, this column is changed to `dtype = str` for future one-hot encoding.

```
update_dtypes = {"MSSubClass": str}
all_features = all_features.astype(update_dtypes)
```

1.3 LotFrontage

LotFrontage describes the length of street connected to the building (?), so it doesn't make sense for a building to have np.nan value. All np.nan values are replaced by 0.

```
all_features.LotFrontage = all_features.LotFrontage.fillna(0)
```

1.4 Masonry Veneer

MasVnrType and MasVnrArea respectively include information regarding the type of Masonry Veneer and its size. Because MasVnrType has a "None" option, it does not make sense to have np.nan values within those two columns. All NULL values in MasVnrType are replaced by "None" and all NULL values in MasVnrArea are replaced by 0.

```
all_features.MasVnrType = all_features.MasVnrType.fillna("None")
all_features.MasVnrArea = all_features.MasVnrArea.fillna(0)
```

1.5 External Quality and Condition

(This is true for all quality and condition columns and will not be repeated below)
ExterQual and ExterCond are read as categorical columns. However, there exists a column OverallQual that establishes a map between a verbal rating and a numerical rating. Therefore, these two columns are converted into numerical columns to express the "ordering" information. Note that the lowest possible score on OverallQual is 1, so I use 0 to encode a "NA" if it is present.

```
verbal_rating_dict = {"Ex": np.float64(9),
                      "Gd": np.float64(7),
                      "TA": np.float64(5),
                      "Fa": np.float64(3),
                      "Po": np.float64(2),
                      "NA": np.float64(0)}
def vr_map(verbal_rating):
    # If verbal rating is string, return appropriate value
    if isinstance(verbal_rating, str):
        return verbal_rating_dict[verbal_rating]
    else:
        return np.nan
ext_qual_num = [verbal_rating_dict[token] for
                token in all_features.ExterQual.values]
ext_cond_num = [verbal_rating_dict[token] for
                token in all_features.ExterCond.values]
all_features.ExterQual = ext_qual_num
all_features.ExterCond = ext_cond_num
```

1.6 Basement

There are a number of columns related to basement:
BsmtCond, BsmtQual, BsmtExposure, BsmtFinType1, BsmtFinType2, BsmtFinSF1, BsmtFinSF2, BsmtUnfSF, TotalBsmtSF.

There are rows in which BsmtQual is not NULL, but BsmtCond is NULL. This is inconsistent because any one of those two being not NULL indicates the existence of a basement. This is resolved by replacing the NULL value by the mode value of the missing-value column for all buildings from the same year built. Similar discrepancy between other pairs of variables is resolved in the same way

```
# There are rows in which BsmtQual is not NULL, but BsmtCond is NULL
# such NULL values will be replaced by the mode of BsmtCond of buildings
# from the same year built
row_indices = all_features.query("not BsmtQual.isnull() and BsmtCond.isnull()").index
for row_index in row_indices:
    # Get the year built
    YB = all_features.iloc[row_index].YearBuilt
    # Find the mode of BsmtCond
    BC_mode = all_features.query("YearBuilt == @YB").BsmtCond.mode()[0]
    # replace!
    all_features.iloc[row_index, list(all_features).index("BsmtCond")] = BC_mode

#   There are tows in which BsmtQual is not NULL but BsmtCond is NULL
#   such NULL values will be replaced by the mode of BsmtQual of buildings
#   from the same year built
row_indices = all_features.query("BsmtQual.isnull() and not BsmtCond.isnull()").index
for row_index in row_indices:
    # Get the year built
    YB = all_features.iloc[row_index].YearBuilt
    # Find the mode of BsmtCond
    BQ_mode = all_features.query("YearBuilt == @YB").BsmtQual.mode()[0]
    # replace!
    all_features.iloc[row_index, list(all_features).index("BsmtQual")] = BQ_mode
```

Then, it does not make sense that BsmtFinSF1, BsmtFinSF2, BsmtUnfSF, and TotalBsmtSF have NULL values since not having a basement correspond to any square foortage being 0. Therefore, all NULL values are replaced by 0

```
all_features.BsmtFinSF1 = all_features.BsmtFinSF1.fillna(0)
all_features.BsmtFinSF2 = all_features.BsmtFinSF2.fillna(0)
all_features.BsmtUnfSF = all_features.BsmtUnfSF.fillna(0)
all_features.TotalBsmtSF = all_features.TotalBsmtSF.fillna(0)
```

Finally, convert BsmtQual and BsmtCond to the appropriate numerical scale.

```
all_features.BsmtQual = all_features.BsmtQual.apply(vr_map)
all_features.BsmtCond = all_features.BsmtCond.apply(vr_map)
```

1.7 Electrical

Although the Kaggle page does not specify an nan possibility for this column, there exists 1 row for which Electrical is NULL. Such NULL value is replaced by the mode.

```
drop_rows = list(all_features.query("Electrical.isnull()").index)
E_mode = all_features.Electrical.mode()[0]
all_features.iloc[drop_rows, list(all_features).index("Electrical")] = E_mode
```

1.8 Kitchen

KitchenQual contains NULL values despite KitchenAbvGr containing no NULL values. Although a far-stretch, I assume that customers should assume the worst when kitchen information is missing. Therefore, all NULL values in KitchenQual are replaced by "Po", the lowest quality rating possible. Then KitchenQual is converted into numerical scale.

```
all_features.KitchenQual = all_features.KitchenQual.fillna("Po")
all_features.KitchenQual = all_features.KitchenQual.apply(vr_map)
```

1.9 Fireplace

Fireplace does not contain NULL values. FireplaceQu is converted into numerical scale.

```
all_features.FireplaceQu = all_features.FireplaceQu.apply(vr_map)
```

1.10 Garage

There are a number of columns related to garage: * GarageType * GaragteYrBlt * GarageFinish * GarageCars * GarageArea * GarageCond * GarageQual

There exist rows in which GarageType is not NULL but GarageYrBlt is NULL. Such NULL value is replaced by the value from YearBuilt since the year a house is built is most likely the year a garage is built.

```
indices = list(all_features.query("not GarageType.isnull() and GarageYrBlt.isnull()").index)
GYB_index = list(all_features).index('GarageYrBlt')
YB_index = list(all_features).index('YearBuilt')
for row_index in indices:
    all_features.iloc[row_index, GYB_index] = all_features.iloc[row_index, YB_index]
```

There exist rows in which GarageType is not NULL but GarageFinish is NULL. Such NULL value is replaced by the mode value of GarageFinish of all rows with the same GarageYrBlt.

```
indices = list(all_features.query("not GarageType.isnull() and GarageFinish.isnull()").index)
target_col = list(all_features).index('GarageFinish')
for row_index in indices:
    #   Get GarageYrBlt
    GYB = all_features.iloc[row_index, GYB_index]
    GF_mode = all_features.query("GarageYrBlt == @GYB").iloc[:, target_col].mode()[0]
    all_features.iloc[row_index, target_col] = GF_mode
```

There exist rows in which GarageType is not NULL but GarageCars is NULL. Such NULL value is replaced by the mode of GarageCars of all rows with the same GarageYrBlt

```
indices = list(all_features.query("not GarageType.isnull() and GarageCars.isnull()").index)
target_col = list(all_features).index('GarageCars')
for row_index in indices:
    #   Get GarageYrBlt
    GYB = all_features.iloc[row_index, GYB_index]
    GC_mode = all_features.query("GarageYrBlt == @GYB").iloc[:, target_col].mode()[0]
    all_features.iloc[row_index, target_col] = GC_mode
```

A similar treatment is applied to GarageArea

```
indices = list(all_features.query("not GarageType.isnull() and GarageArea.isnull()").index)
target_col = list(all_features).index('GarageArea')
for row_index in indices:
    #   Get GarageYrBlt
    GYB = all_features.iloc[row_index, GYB_index]
    GA_mean = all_features.query("GarageYrBlt == @GYB").iloc[:, target_col].mean()
    all_features.iloc[row_index, target_col] = GA_mean
```

Finally, convert GarageCond and GarageQual into numerical scale

```
all_features.GarageCond = all_features.GarageCond.apply(vr_map)
all_features.GarageQual = all_features.GarageQual.apply(vr_map)
```

1.11 PoolArea

There are rows in which PoolArea is greater than 0 but PoolQC is NULL. This is resolved by replacing such NULL values by the mean PoolQC numerical value after converting the verbal rating into numerical rating.

```
all_features.PoolQC = all_features.PoolQC.apply(vr_map)
sub_val = all_features.PoolQC.mean()
row_indices = list(all_features.query("PoolArea > 0 and PoolQC.isnull()").index)
all_features.iloc[row_indices, list(all_features).index("PoolQC")] = sub_val
```

1.12 MiscFeature and MiscVal

There exist rows in which MiscFeature is NULL but MiscVal is greater than 0. Such NULL values are replaced by "Othr".

```
row_indices = all_features.query("MiscFeature.isnull() and MiscVal > 0").index
all_features.iloc[row_indices,list(all_features).index("MiscFeature")] = "Othr"
```

1.13 YrSold and MoSold

YrSold and MoSold is consolidated into a single numerical variable that indicates how many months have passed since January of 2006. The new feature is named SaleTimestamp and the two original columns are dropped.

```
all_features["SaleTimestamp"] = (all_features.YrSold - 2006)*12 + all_features.MoSold
all_features = all_features.drop(["YrSold", "MoSold"], axis=1)
```

1.14 One-hot and normalization

Convert all categorical variables at this time into one-hot encoding and drop the original column

```
cat_colnames = list(all_features.dtypes[all_features.dtypes == 'object'].index)
num_colnames = list(all_features.dtypes[all_features.dtypes != 'object'].index)
for cat_colname in cat_colnames:
    all_features = make_one_hot(all_features,
        list(all_features).index(cat_colname))
```

At this moment, collect all numerical, non-one-hot columns that still have NULL values. If it is a column indicating quality/condition, replace the NULL values by 0, the lowest rating possible; otherwise, replace NULL by column mean.

```
nullc = [c_name for c_name in list(all_features) if all_features[c_name].isnull().values.any()]
for colname in nullc:
    # If you are one of the columns below, replace NaN by 0
    if colname in ["BsmtQual", "BsmtCond",
                   "FireplaceQu",
                   "GarageQual", "GarageCond",
                   "PoolQC"]:
        all_features[colname] = all_features[colname].fillna(0)
    # Otherwise, replace by mean of the column
    else:
        col_mean = all_features[colname].mean()
        all_features[colname] = all_features[colname].fillna(col_mean)
```

Finally, normalize all numerical columns:

```
for num_colname in num_colnames:
    col_mean = all_features[num_colname].mean()
    col_se = all_features[num_colname].std()
    all_features[num_colname] = (all_features[num_colname] - col_mean) / col_se
```

1.15 Export

Convert `pandas.DataFrame` object into numpy array; this is valid because all columns are either one-hot numerical or continuous numerical. Split the composite `all_features` back into training and testing sets, then export the numpy array to binary save files for future loading.

```
train_X = all_features.iloc[:train.shape[0], 1:].values
test_X = all_features.iloc[train.shape[0]:, 1:].values
train_Y = train.SalePrice.values.reshape((-1, 1))

np.save("./data/clean/train_X.npy", train_X)
np.save("./data/clean/test_X.npy", test_X)
np.save("./data/clean/train_Y.npy", train_Y)
```

1.16 Appendix

There are 2 methods frequently used throughout the cleaning process. Their code is posted here:

```
def normalize(df, col_idx, scale=True):
    # Given a DataFrame object and a column index that points to a
    # numerical column, return a NEW copy of the data frame
    # with that column normalized (if scale is True) or
    # mean centered (if scale is false)
    # If there are NA and NaNs, replace them with the mean; this means
    # computing the mean without taking into consideration these values
    # then assigning the appropriate rows with 0, since the output is
    # either mean centered or normalized

    # Get a copy of the data frame object and the column out of the
    # copies output DF. From now all operations will be done on the
    # copy
    output_df = df.copy()
    target_col = output_df.iloc[:, col_idx]

    # Compute the mean and the sample error (divided by n-1)
    col_mean = tagret_col.mean()
    col_se = target_col.std()
    # If scale is set to true, normalize the column to make
    # mean 0 and sample error 1. Otherwise, only do mean centering
    if scale:
        target_col = (target_col - col_mean) / col_se
    else:
        target_col = (target_col - col_mean)

    # Fill the NaNs and NAs by 0 because we have at least mean centered
    # the column
    target_col.fillna(0)
    # Fill the copy with the new column
    output_df.iloc[:, col_idx] = target_col
    return output_df

def make_one_hot(df, col_idx):
    # Given a DataFrame object and a column index that points to
    # a categorical variables. Return a NEW copy of the data frame
    # in which the categorical column is removed, and one-hot columns
    # are created and appended to the end

    # Get a copy of the data frame for output, get the column
    # then drop this column from the output DF
    output_df = df.copy()
    # When extracting the target column, enforce the dtype "str"
    target_col = output_df.iloc[:, col_idx].astype(str)
    # Extract the column name so that NaNs from different columns
```

```
# will not get confused; this means dummy_na should be False
target_colname = list(output_df)[col_idx]
# Replace values within this column by colname_val
target_col[:] = target_colname + "_" + target_col
# Drop the categorical column
output_df = output_df.drop(labels=list(output_df)[col_idx],
                           axis=1)

# Use pd method to create one-hot, including using NAN as a distinct
# label
one_hot = pd.get_dummies(target_col, dummy_na = False)

# Concatenate the two dfs and return it
return pd.concat([output_df, one_hot], axis=1)
```