02_data_pipeline_refactored

September 23, 2019

1 Notebook 2. Refactored Data Cleaning Pipeline and Train-Test Loop

1.1 ### By Max Pechyonkin

1.2 About Refactored Code

This notebook uses exploratory information from the two previous notebooks and creates a single class `DataCleaner` to process assignment data. I did not explain why I made decisions to handle certain variables in a certain way because it was done in the previous notebooks.

This class is optimized to process this particular dataset, with all its peculiarities and unique values.

I thought about writing a more general class that would automatically detect inconsistencies and problems with data, but that would require having a clearly defined specification of the schema of the data with allowed values described, which I did not have. Another benefit of the `DataCleaner` is that it conforms to `sklearn`'s transformer protocol, which means it can be used as part of a `Pipeline` to simplify data processing and prediction pipelines.

I was also planning to implement training and testing as a custom `Estimator` class, and then chain the `DataCleaner` and `Estimator` together in the `Pipeline` but due to time constraints that was not finished.

Note: please also read the PDF notes acoompanying this submission.

[1]:
```
import json
from functools import partial
from typing import Dict, Union

import numpy as np
import pandas as pd
from pandas import DataFrame, Series, Index
from sklearn.base import TransformerMixin
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.metrics import mean_squared_error
import category_encoders as ce

import matplotlib.pyplot as plt
%matplotlib inline
```

[2]:
```
class DataCleaner(TransformerMixin):
    """
```

```
    This class performs all necessary cleaning required
    to proceed to the modeling step:
    1. remove all tows with NaNs
    2. remove row with NaNs percentages > 99%
    3. remove unnecessary columns:
       - id-like columns to avoid overfitting
       - columns with URLs
       - columns with very messy unstructured data
       - columns requiring too much work featurizing (like NLP)
    4. perform cleaning on the remaining columns
       - parse values with inconsistent formatting
    """

    def __init__(self):
        """
        This data cleaner is stateless, but the method is
        required by superclass.
        """
        pass

    def fit(self, X, y=None):
        """
        The data cleaner is stateless, but this method is required by␣
↪superclass.
        It does nothing in this case.
        :param X:
        :param y:
        :return:
        """

        print("You don't need to fit for cleaning the data!")
        return self

    @staticmethod
    def _remove_all_nans_columns(X: DataFrame) -> DataFrame:
        """
        Remove columns that have all NaNs

        :param X: input DataFrame
        :return: DataFrame with removed NaNs columns
        """

        all_nan_cols = X.isnull().all()
        all_nan_col_names = all_nan_cols[all_nan_cols].index
        X = X.drop(all_nan_col_names, axis=1)
        return X
```

```
    @staticmethod
    def _get_nans_percentages(X: DataFrame) -> DataFrame:
        """
        Get percentages of NaNs for each column.

        Returns Series with index of column names and float values.

        :param X: input DataFrame
        :return: DataFrame with percentage of NaNs for each column of input␣
↪DataFrame
        """

        nrows, _ = X.shape
        return (X.isnull().sum(axis=0) / nrows).sort_values(ascending=False)

    def _get_colnames_with_nans_above_thresh(self, X: DataFrame, thresh: float)␣
↪-> Index:
        """
        Return column names with NaNs proportions above the given level.

        :param X: input DataFrame
        :param thresh: threshold above which to remove columns
        :return: Index with column names
        """

        percentages = self._get_nans_percentages(X)
        result = percentages[percentages > thresh]
        return result.index

    def _remove_cols_with_nans_above_thresh(self, X: DataFrame, thresh: float)␣
↪-> DataFrame:
        """
        Remove columns that have NaNs percentage above a given threshold.
        :param X: input DataFrame
        :param thresh: threshold above which to remove columns
        :return: DataFrame with removed columns
        """

        colnames_with_too_many_nans = self.
↪_get_colnames_with_nans_above_thresh(X, thresh=thresh)
        X = X.drop(colnames_with_too_many_nans, axis=1)
        return X

    @staticmethod
    def _remove_unnecessary_columns(X: DataFrame) -> DataFrame:
        """
        Remove columns that I deemed unnecessary for one of three
```

```
        reasons:
          1. id-like data
          2. too much effort needed to process column
          3. data is too messy, noisy, unstructured
          4. too many NaNs, and remaining non NaNs have too many unique values

        :param X: input DataFrame
        :return: DataFrame with columns removed
        """

        cols_to_remove = [
            'id',
            'asins',
            'dateAdded',
            'dateUpdated',
            'descriptions',
            'ean',
            'features',
            'imageURLs',
            'keys',
            'manufacturerNumber',
            'merchants',
            'name',
            'prices.amountMin',
            'prices.dateAdded',
            'prices.dateSeen',
            'prices.sourceURLs',
            'skus',
            'sourceURLs',
            'upc',
            'weight',
            'categories',
            'prices.size',
            'prices.color',
        ]

        X = X.drop(cols_to_remove, axis=1)
        return X

    @staticmethod
    def _process_currency_column(X: DataFrame) -> DataFrame:
        """
        Remove rows from df corresponding to currency column values that are␣
↪not currencies.
        Supported currencies are defined in the list below.

        :param X: DataFrame with currencies column
```

```
        :return: DataFrame with removed rows corresponding to illegal␣
↪currencies.
        """

        legal_currencies = [
            'USD',
            'AUD',
            'CAD',
            'EUR',
            'GBP',
        ]

        rows_illegal_currency = X['prices.currency'].apply(lambda x: x not in␣
↪legal_currencies)
        rows_illegal_fx_idxs = rows_illegal_currency[rows_illegal_currency ==␣
↪True].index
        X = X.drop(rows_illegal_fx_idxs, axis=0)

        return X

    @staticmethod
    def _is_invalid_price_string(s: str) -> bool:
        """
        Return true if string represents a valid float number.

        :param s: input string
        :return: whether string can be converted to float
        """

        # TODO: use try-except when casting to float to decide.
        s = str(s)
        illegal_chars = '-T:Z'
        for c in illegal_chars:
            if c in s:
                return True
        return False

    def _remove_illegal_price_values(self, X: DataFrame) -> DataFrame:
        """
        Remove rows with illegal price values.

        Note: this function doesn't perform conversion to float!

        :param X: input DataFrame
        :return: DataFrame with illegal rows removed
        """
```

```
        illegal_price_mask = X['prices.amountMax'].apply(self.
↪_is_invalid_price_string)
        illegal_price_idxs = illegal_price_mask[illegal_price_mask].index
        X = X.drop(illegal_price_idxs)
        return X

    @staticmethod
    def _convert_to_usd(row: Series, fx_rates: Dict[str, float]) -> float:
        """
        Convert price from other currency to USD.

        :param row: row from DataFrame to process
        :param fx_rates: dictionary of fx rates
        :return: price converted to USD
        """

        price = float(row['prices.amountMax'])
        currency = row['prices.currency']
        if not currency:
            # if currency is NaN assume USD and return price
            return price
        fx_rate = fx_rates[currency]
        return price * fx_rate

    def _convert_price_to_usd(self, X):
        """
        Convert all prices to USD by using exchange rates.

        :param X: input DataFrame
        :return: DataFrame with processed float price, converted to USD
        """

        rates = {
            'USD': 1.00,
            'AUD': 0.68,
            'CAD': 0.75,
            'EUR': 1.10,
            'GBP': 1.25,
        }

        X['price'] = X.apply(partial(self._convert_to_usd, fx_rates=rates),␣
↪axis=1)
        X = X.drop(['prices.amountMax'], axis=1)
        return X

    def _process_price_column(self, X: DataFrame) -> DataFrame:
        """
```

```
        Three-step process of the price column:
            1. process currency column
            2. remove illegal price values
            3. convert prices to USD

        :param X: input DataFrame
        :return: processed DataFrame
        """

        X = self._process_currency_column(X)
        X = self._remove_illegal_price_values(X)
        X = self._convert_price_to_usd(X)
        return X

    @staticmethod
    def _process_issale_column(X: DataFrame) -> DataFrame:
        """
        Process issale column, make sure it only has three values:
          - True
          - False
          - NaN

        :param X: input DataFrame
        :return: processed DataFrame
        """

        def process_is_sale(x: Union[str, float]) -> Union[bool, float]:
            """ Process individual entry in issale column. """
            if isinstance(x, bool):
                return x
            elif x.capitalize() == 'True':
                return True
            elif x.capitalize() == 'False':
                return False
            else:
                print(x)
                print(type(x))
                raise ValueError("Something went wrong!")

        X['prices.isSale'] = X['prices.isSale'].apply(process_is_sale)
        return X

    @staticmethod
    def _process_colors_column(X: DataFrame) -> DataFrame:
        """
        Process colors: make them lowercase, remove spaces and dashes.
```

```
        :param X: input DataFrame
        :return: processed DataFrame
        """
        X['colors'] = X['colors'].apply(
            lambda x: x.lower().replace(' ', '').replace('-', '') if not pd.
↪isnull(x) else x)
        return X

    @staticmethod
    def _process_shipping_column(X: DataFrame) -> DataFrame:
        """
        Make shipping into two categories: free and not free.

        :param X: input DataFrame
        :return: processed DataFrame
        """

        def process_shipping(x: Union[str, float]) -> Union[str, float]:
            """ Process individual entry in shipping column. """
            if pd.isnull(x):
                return x

            x = x.lower()
            if 'free' in x:
                return 'free'
            else:
                return 'not free'

        X['prices.shipping'] = X['prices.shipping'].apply(process_shipping)
        return X

    @staticmethod
    def _process_sizes_column(X: DataFrame) -> DataFrame:
        """
        Calculate the number of sizes available.

        :param X: input DataFrame
        :return: processed DataFrame
        """

        def process_sizes(x: Union[str, float]) -> Union[str, float]:
            """ Process individual entry in sizes column. """
            if pd.isnull(x):
                return x
            return len(x.split(','))

        X['sizes'] = X['sizes'].apply(process_sizes)
```

```
        return X

    @staticmethod
    def _process_dimension_column(X):
        """
        Calculate total dimension: height + width + depth where available.

        Also deals with 1 edge case of unique value.

        :param X: input DataFrame
        :return: processed DataFrame
        """

        def process_dimensions(x: float) -> float:
            """ Process individual entry in dimensions column. """
            if pd.isnull(x):
                return x

            # deal with edge case
            if x == '32 inches':
                return 32.0

            x = x.lower()
            dims = x.split('x')
            dims = [d.replace(' ', '').replace('in', '') for d in dims]
            dims = map(float, dims)
            return sum(dims)

        X['dimension'] = X['dimension'].apply(process_dimensions)
        return X

    @staticmethod
    def _calculate_review_ratings(X: DataFrame) -> DataFrame:
        """
        Calculate average review rating, where available, otherwise set to zero.

        :param X: input DataFrame
        :return: processed DataFrame
        """

        def parse_review(x: Union[str, float]) -> Union[str, float]:
            """ Process individual entry in reviews column. """
            if pd.isnull(x):
                return x

            rating = 0.0
            try:
```

```
                reviews = json.loads(x)
                for r in reviews:
                    if 'rating' in r.keys():
                        rating += float(r['rating'])
                return rating / len(reviews)
            except:
                return rating

        X['reviews'] = X['reviews'].apply(parse_review)
        return X

    @staticmethod
    def _remove_price_outliers(X: DataFrame, thresh: float) -> DataFrame:

        X = X.drop(X[X.price > thresh].index)
        return X

    @staticmethod
    def _convert_numerical_nans(X: DataFrame, value: float) -> DataFrame:
        """
        Convert NaNs of a numerical column to a given value,
        this is required for categorical encoder later.

        :param X: input DataFrame
        :return: processed DataFrame
        """
        cols_to_fill_nas = ['dimension', 'reviews', 'sizes']
        for c in cols_to_fill_nas:
            X[c] = X[c].fillna(value)
        return X

    def transform(self, X: DataFrame):
        """
        Process all the steps.

        :param X: input uncleaned DataFrame
        :return: cleaned and processed DataFrame, ready for modeling step
        """

        X = self._remove_all_nans_columns(X)
        X = self._remove_cols_with_nans_above_thresh(X, thresh=0.99)
        X = self._remove_unnecessary_columns(X)
        X = self._process_price_column(X)
        X = self._process_issale_column(X)
        X = self._process_colors_column(X)
        X = self._process_shipping_column(X)
        X = self._process_sizes_column(X)
```

```
        X = self._process_dimension_column(X)
        X = self._calculate_review_ratings(X)
        X = self._remove_price_outliers(X, thresh=2000)
        X = self._convert_numerical_nans(X, value=-999.9)
        return X
```

1.2.1 Loading, cleaning and splitting the data into features and target variable

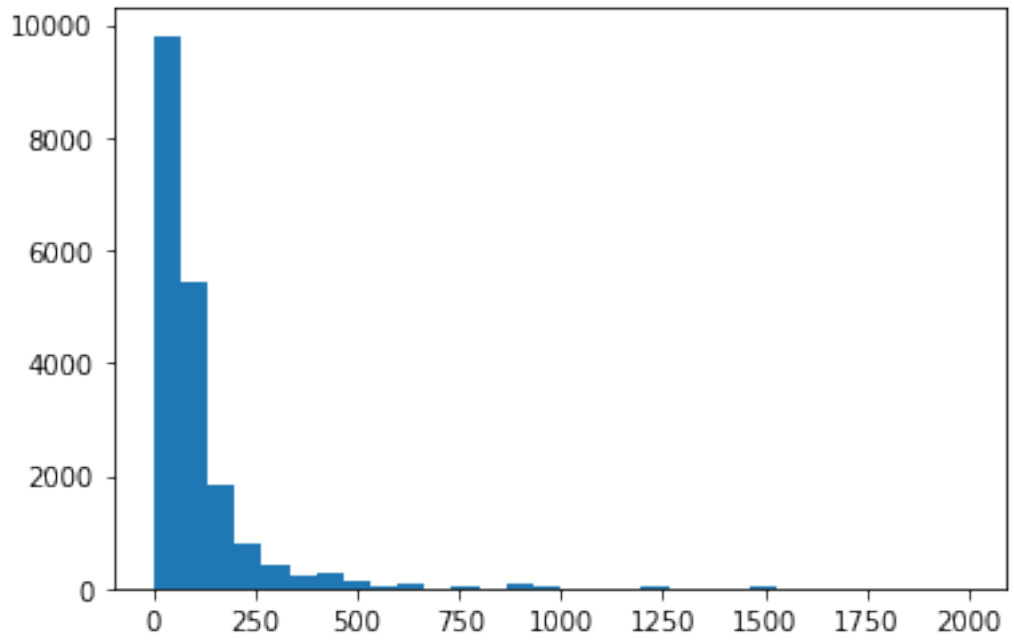
[3]:
```
DATA_PATH = '7004_1.csv'

cleaner = DataCleaner()

data = pd.read_csv(DATA_PATH, error_bad_lines=False, warn_bad_lines=False)
data = cleaner.transform(data)

X, y = data.drop('price', axis=1), data['price']
```

```
/Users/fazzl/miniconda3/envs/fastai_dev/lib/python3.7/site-
packages/IPython/core/interactiveshell.py:3058: DtypeWarning: Columns
(20,21,25,29,30,36) have mixed types. Specify dtype option on import or set
low_memory=False.
  interactivity=interactivity, compiler=compiler, result=result)
```

1.2.2 Regadring disctribution of the target

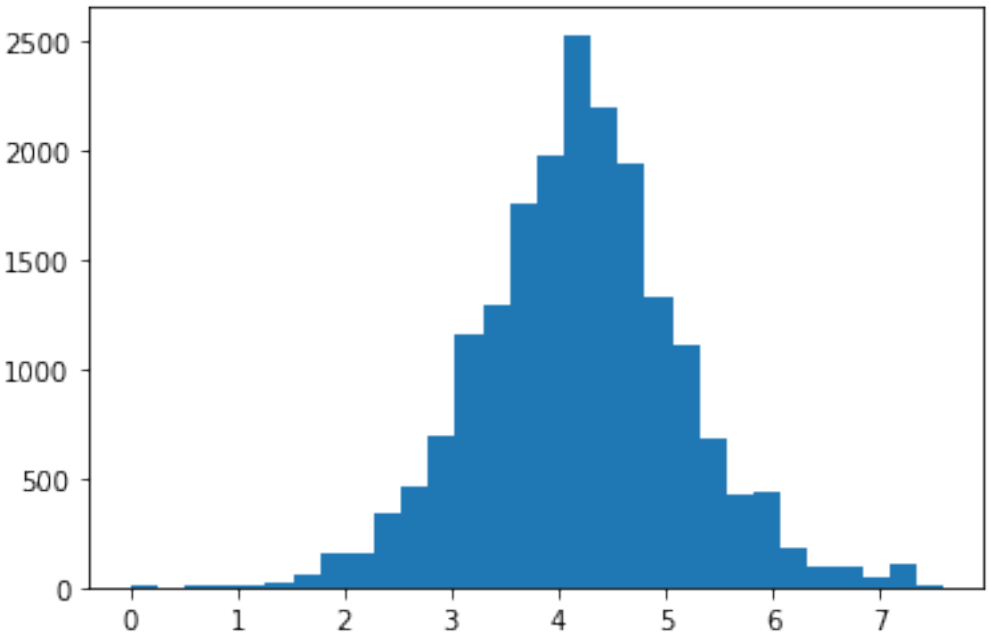
Let's examine the distribution of the target. As we see, there is a long tail.

[4]:
```
plt.hist(y, bins=30);
```



[5]:
```
def logify(data):
    """
    Applies log transform to make distribution more symmetrical.
    Goes from price to log-price.
    Note: 1 added for numerical stability (hadles zero input)
    """
    return np.log(1 + data)

def unlogify(data):
    """
    Inverse transform to go from log-price to price.
    """
    return np.exp(data) - 1
```

[6]:
```
plt.hist(logify(y), bins=30);
```



This is much better.

1.3 We will proceed with single-validation training

1.3.1 Training Stage

1. we split data into train (5/6) and test (1/6) sets.
2. train set will be split into 5 equal folds, and for each fold we will:
    - fit a CatBoost categorical encoder

- encode test fold data using the encoder from step above
- train a model on the encoded data, optionally using other folds for hyperparameter tuning

1.3.2 Testing Stage

1. we put test data through encoders fit in the training stage
2. for each encoded test data, we retrieve predictions for that model
3. we combine predictions (average) to arrive at the final prediction
4. evaluate error rate of the final prediction

This can be summarized in the image below:

1.3.3 On the choise of categorical feature encoders

I chose CatBoost encoder because it encodes categorical features in one vector, as opposed to one-hot encoding, for example. This is particulary convenient when having multiple distinct features. If using one-hot encoding, this would inflate the number of variables and make it harder for the model to learn due to the curse of dimensionality.

CatBoost uses information about relative frequencies of veatures and values of the target variable for the train data set to encode the features numerically in just one vector, which keeps the number of features unchanged.

1.3.4 Prepare test indices and validation fold indices

[7]:
```
np.random.seed(42)  # for reproducibility
random_idxs = np.random.permutation(len(data))
split_idxs = np.array_split(random_idxs, 6)
test_idxs, folds_idxs = split_idxs[0], split_idxs[1:]
```

1.3.5 Train Stage

[8]:
```
# categorical columns to encode using CatBoost
categorical=[
    'brand',
    'colors',
    'manufacturer',
    'prices.condition',
    'prices.currency',
    'prices.isSale',
    'prices.merchant',
    'prices.offer',
    'prices.returnPolicy',
    'prices.shipping',
]

# make folds of data
y_folds = []
```

```
y_logified = []
X_folds = []
for fold_idxs in folds_idxs:
    X_fold, y_fold = X.iloc[fold_idxs], y.iloc[fold_idxs]
    X_folds.append(X_fold)
    y_folds.append(y_fold)
    y_logified.append(logify(y_fold))
```

[9]:
```
# make encoders and encoded data folds
encoders = []
X_folds_encoded = []
for X_fold, y_fold in zip(X_folds, y_logified):
    cat_encoder = ce.CatBoostEncoder(cols=categorical)
    X_encoded = cat_encoder.fit_transform(X_fold, y_fold)
    X_folds_encoded.append(X_encoded)
    encoders.append(cat_encoder)
```

[10]:
```
# create and train models (no hyperparameter optimization)
models = []
for X_train, y_train in zip(X_folds_encoded, y_logified):
    model = GradientBoostingRegressor()
    model.fit(X_train, y_train)
    models.append(model)
```

1.3.6 Test Stage

[11]:
```
X_test, y_test = X.iloc[test_idxs], y.iloc[test_idxs]

# put test data through encoders
X_test_encoded_folds = []
for e in encoders:
    X_test_encoded = e.transform(X_test)
    X_test_encoded_folds.append(X_test_encoded)

# make predictions through each of the corresponding models
preds = []
for X_test, model in zip(X_test_encoded_folds, models):
    pred = model.predict(X_test)
    preds.append(pred)
```

1.3.7 Analysis of predictions

[12]:
```
preds_aggregated = unlogify(np.array(preds).mean(axis=0))
```

[13]:
```
mean_squared_error(y_test, preds_aggregated)
```

[13]:
```
17856.343197421767
```

As we can see below, the models were able to learn some relationship, but it by far not perfect.

If we convert to log-space then we see that relationship looks much better.

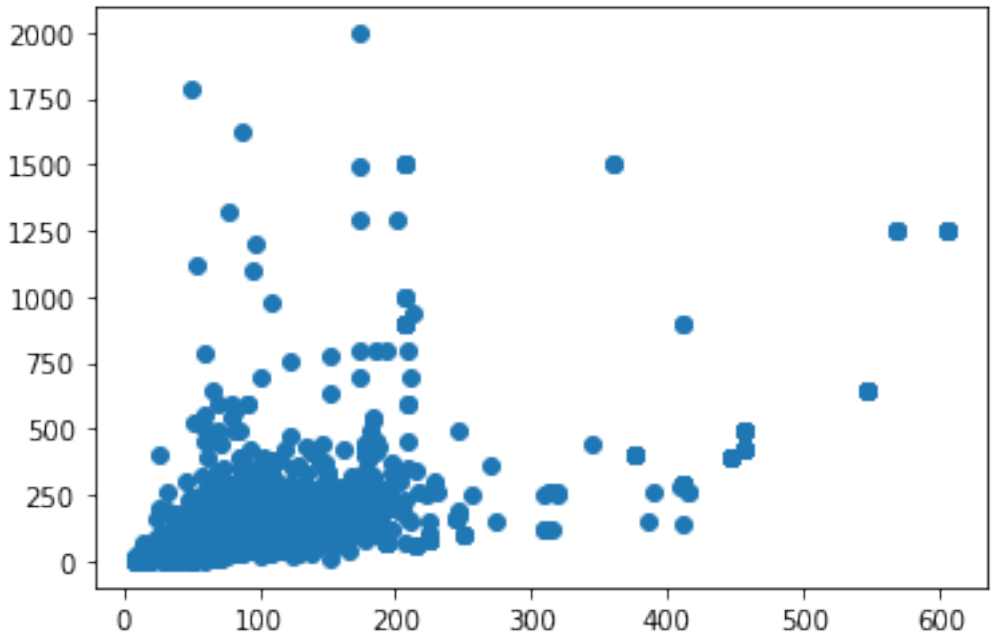
One reason for this is that I did not perform hyperparameter optimization due to time constraints.

Another reason is the data set was very noisy, with many variables having a lot of NaNs. If I had more time, I could perform some ablation studies and examine whether removing some of the variables would improve the situation.

Another way to make the model perform better is to put more work into some of the variables that I discarded, or try to exctract more information from existing variables. For example, an NLP model could be applied to product descriptions, or available sizes information could be parsed in a way that gives more information that just the number of sizes available.

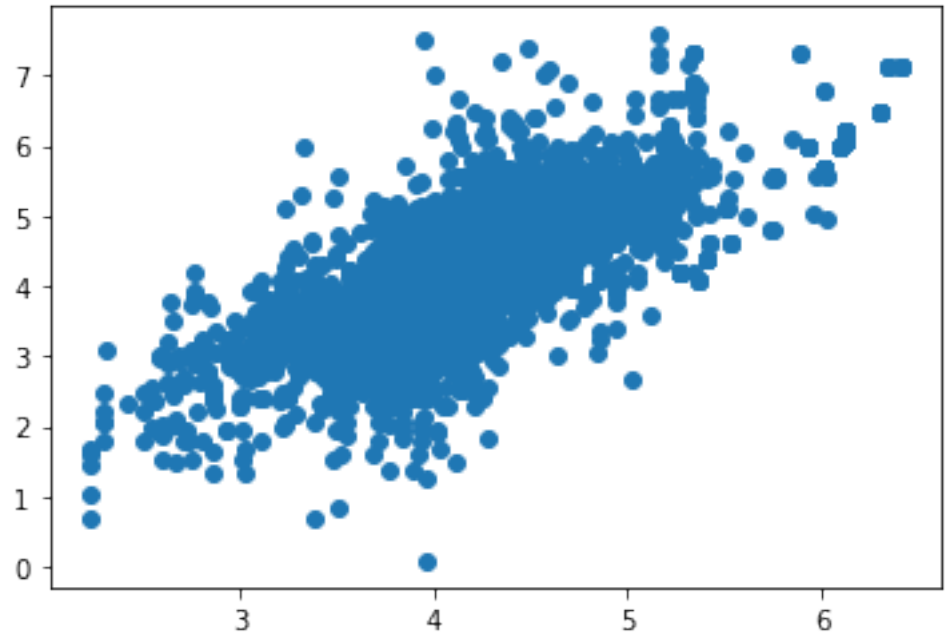
[14]:
```
plt.scatter(preds_aggregated, y_test)
```

[14]:
```
<matplotlib.collections.PathCollection at 0x1a229c55d0>
```



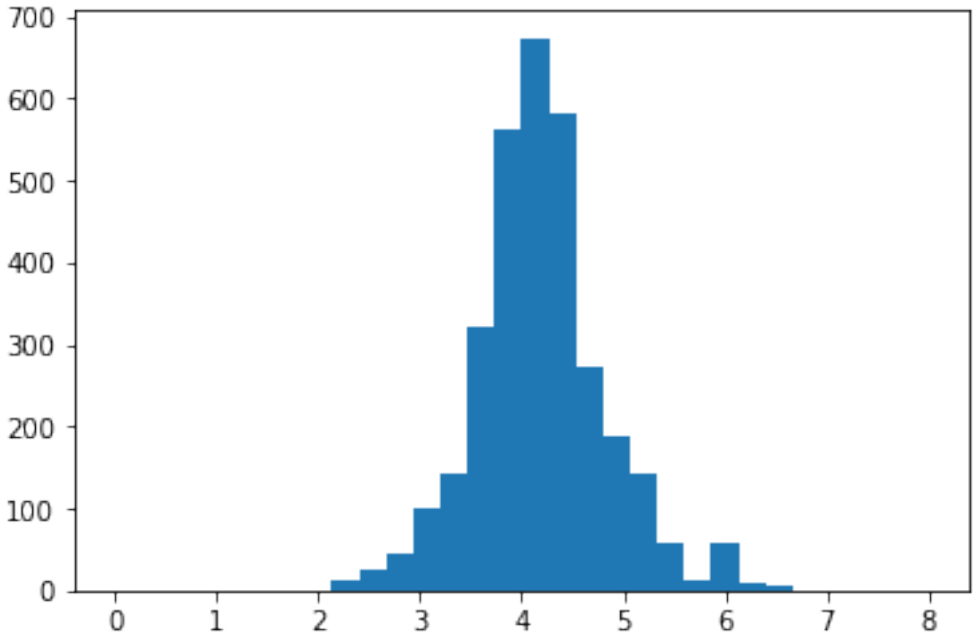
[15]:
```
plt.scatter(logify(preds_aggregated), logify(y_test))
```

[15]:
```
<matplotlib.collections.PathCollection at 0x1a2275aa10>
```

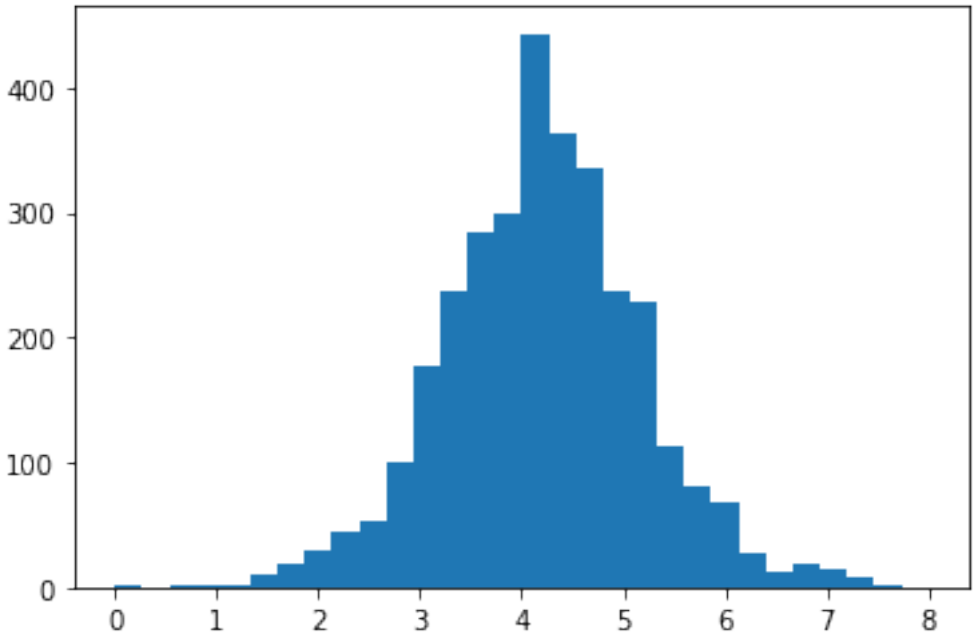


Below you can see distributions of the log-price.

[16]:
```
plt.hist(logify(preds_aggregated), bins=30, range=(0,8));
```



[17]:
```
plt.hist(logify(y_test), bins=30, range=(0,8));
```


400
300
200
100
0
0
1
2
3
4
5
6
7
8

[ ]: