## **Dataset exploration and preparation**

Data: <https://raw.githubusercontent.com/fenago/datasets/main/house_price_prediction.csv>

Source: <https://www.kaggle.com/datasets/shree1992/housedata>

The first step is to import Numpy and Pandas, and then to import the dataset. The following snippet does that and also prints a random couple of rows:

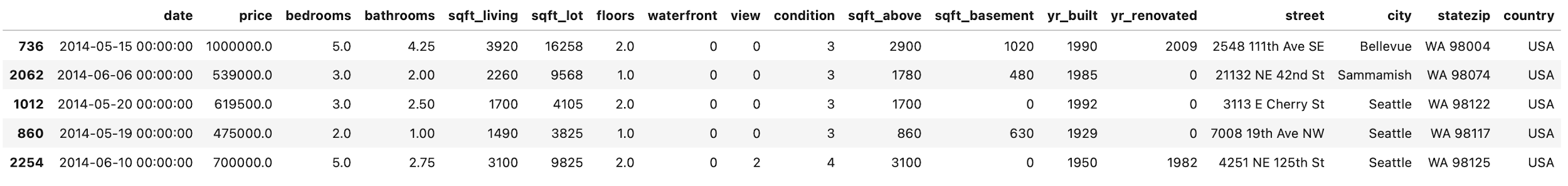
import numpy as np

import pandas as pd

df = pd.read\_csv('data/data.csv')

df.sample(5)

Here’s how the dataset looks like:

Image 2 — Housing prices dataset (image by author)

You definitely can’t pass it to a neural network in this format.

### **Deleting unnecessary columns**

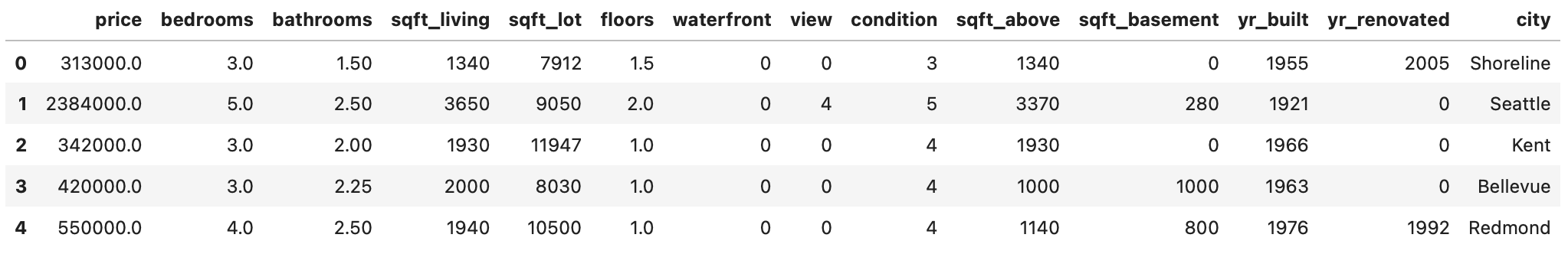
Since we want to avoid spending too much time preparing the data, it’s best to drop most of the non-numeric features. Keep only the city column, as it’s simple enough to encode:

to\_drop = ['date', 'street', 'statezip', 'country']

df = df.drop(to\_drop, axis=1)

df.head()

Here’s how it should look like now:

Image 3 — Dataset after removing most of the string columns (image by author)

You definitely could keep all columns and do some feature engineering with them. It would likely increase the performance of the model. But even this will be enough for what you need today.

### **Feature engineering**

Now you’ll spend some time tweaking the dataset. The yr\_renovated column sometimes has the value of 0. I assume that’s because the house wasn’t renovated. You’ll create a couple of features — house age, was the house renovated or not, was it renovated in the last 10 years, and was it renovated in the last 30 years.

You can use list comprehension for every mentioned feature. Here’s how:

# How old is the house?

df['house\_age'] = [2021 - yr\_built for yr\_built in df['yr\_built']]

# Was the house renovated and was the renovation recent?

df['was\_renovated'] = [1 if yr\_renovated != 0 else 0

for yr\_renovated in df['yr\_renovated']]

df['was\_renovated\_10\_yrs'] = [1 if (2021 - yr\_renovated) <= 10

else 0 for yr\_renovated in df['yr\_renovated']]

df['was\_renovated\_30\_yrs'] = [1 if 10 < (2021 - yr\_renovated) <= 30

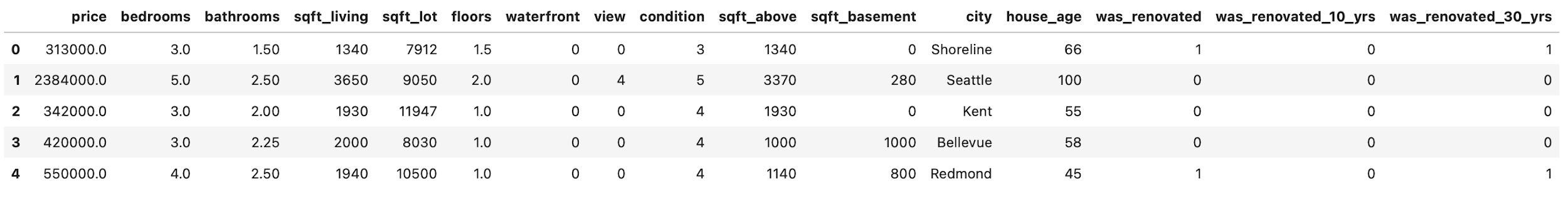
else 0 for yr\_renovated in df['yr\_renovated']]

# Drop original columns

df = df.drop(['yr\_built', 'yr\_renovated'], axis=1)

df.head()

Here’s how the dataset looks now:

Image 4 — Dataset after feature engineering (1) (image by author)

Let’s handle the city column next. Many cities have only a couple of houses listed, so you can declare a function that will get rid of all city values that don’t occur often. That’s what the remap\_location() function will do — if there are less than 50 houses in that city, it’s replaced with something else. It’s just a way to reduce the number of options:

def remap\_location(data: pd.DataFrame,

location: str,

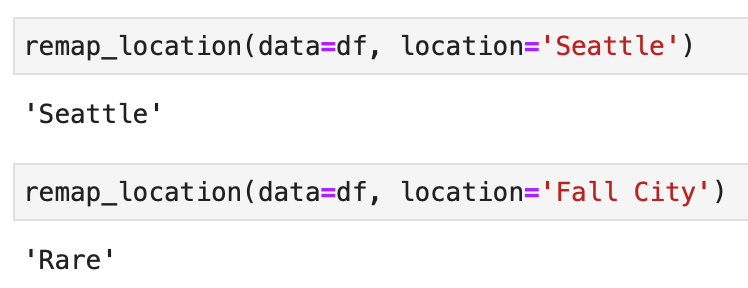
threshold: int = 50) -> str:

if len(data[data['city'] == location]) < threshold:

return 'Rare'

return location

Let’s test the function — the city of *Seattle* has many houses listed, while *Fall City* has only 11:

Image 5 — Remapping city values (image by author)

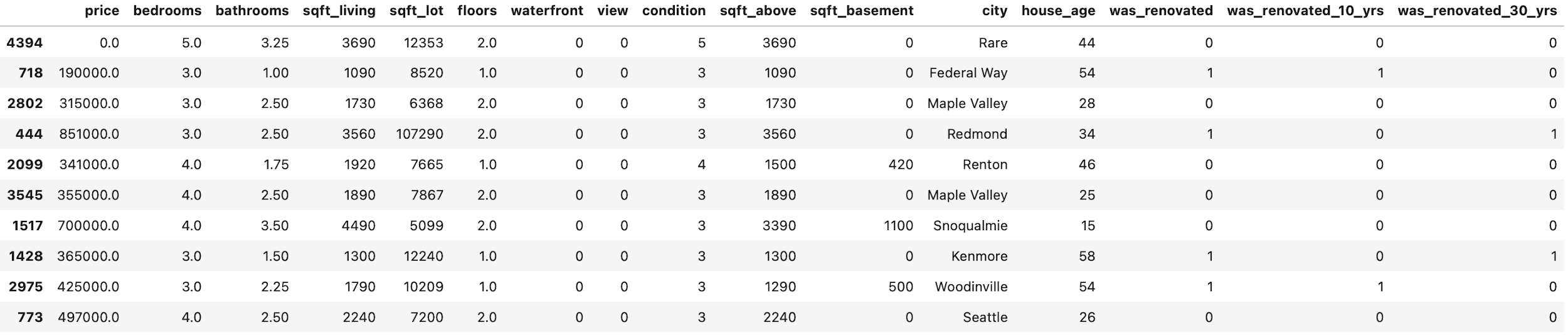
Let’s apply this function to all cities and print a sample of 10 rows:

df['city'] = df['city'].apply(

lambda x: remap\_location(data=df, location=x)

)

df.sample(10)

Image 6 — Dataset after feature engineering (2) (image by author)

Everything looks as it should, so let’s continue.

### **Target variable visualization**

Anytime you’re dealing with prices, it’s unlikely the target variable will be distributed normally. And this housing dataset is no exception. Let’s verify it by importing Matplotlib and visualizing the distribution with a Histogram:

import matplotlib.pyplot as plt

from matplotlib import rcParams

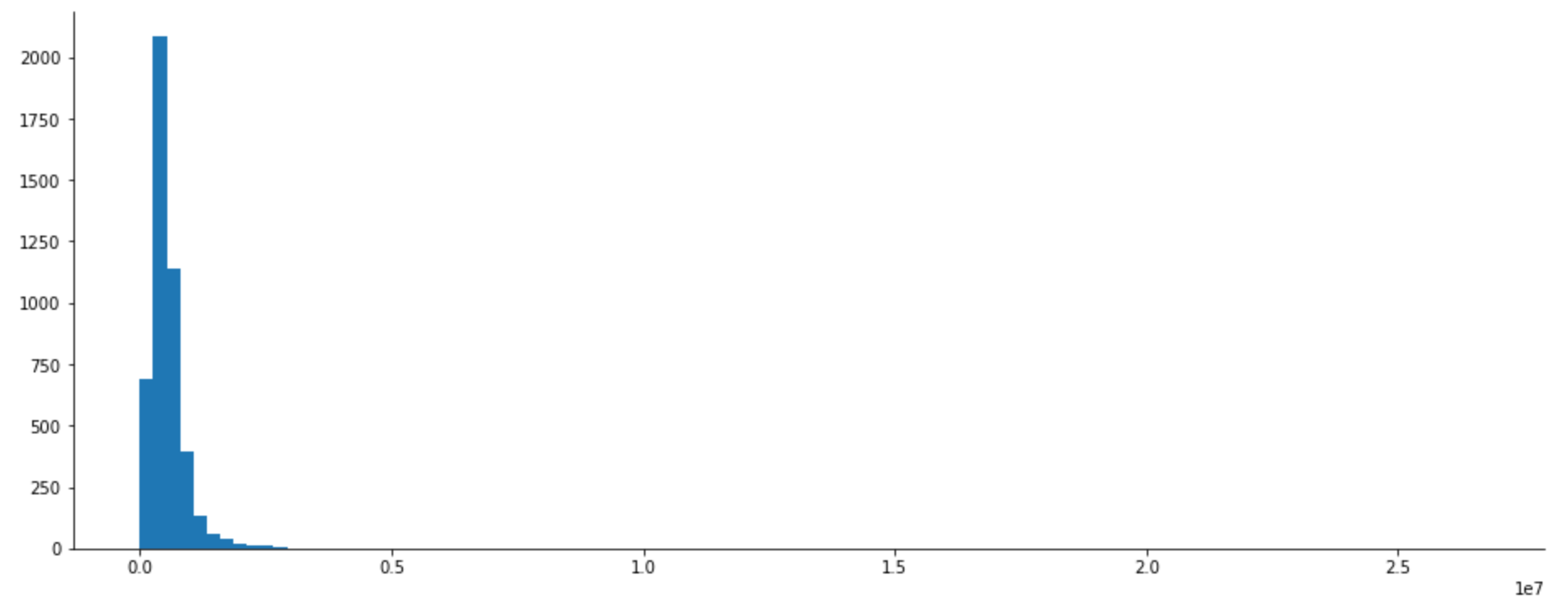
rcParams['figure.figsize'] = (16, 6)

rcParams['axes.spines.top'] = False

rcParams['axes.spines.right'] = False

plt.hist(df['price'], bins=100);

Here’s how it looks like:

Image 7 — Target variable histogram (1) (image by author)

Outliers are definitely present, so let’s handle them next. The pretty common thing to do is to calculate Z-scores. They let you know how many standard deviations a value is located from the mean. In the case of a normal distribution, anything below or above 3 standard deviations is classified as an outlier. The distribution of prices isn’t normal, but let’s still do the Z-test to remove the houses on the far right.

You can calculate the Z-score with Scipy. You’ll assign them as a new dataset column — price\_z, and then keep only the rows in which the absolute value of Z is less than or equal to three.

There are also around 50 houses listed for $0, so you’ll delete those as well:

from scipy import stats

# Calculate Z-values

df['price\_z'] = np.abs(stats.zscore(df['price']))

# Filter out outliers

df = df[df['price\_z'] <= 3]

# Remove houses listed for $0

df = df[df['price'] != 0]

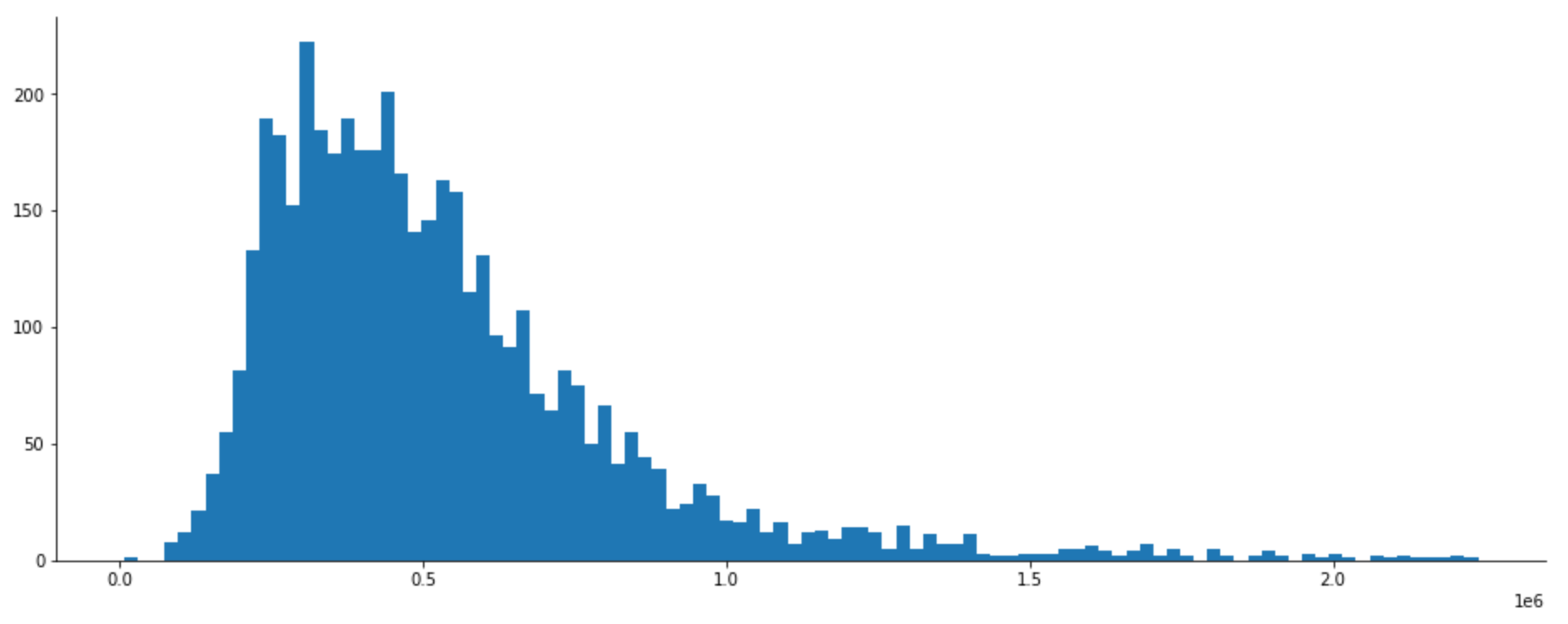
# Drop the column

df = df.drop('price\_z', axis=1)

# Draw a histogram

plt.hist(df['price'], bins=100);

Here’s how the distribution looks like now:

Image 8 — Target variable histogram (2) (image by author)

There’s still a bit of skew present, but let’s declare it *good enough*.

As the last step, let’s convert the data into a format ready for machine learning.

### **Data preparation for ML**

A neural network likes to see only numerical data on the same scale. Our dataset isn’t, and we also have some non-numerical data. That’s where data scaling and one-hot encoding come into play.

You could now transform each feature individually, but there’s a better way. You can use the make\_column\_transformer() function from Scikit-Learn to apply scaling and encoding in one go.

You can ignore features like waterfront, was\_renovated, was\_renovated\_10\_yrs, and was\_renovated\_30\_yrs, as they already are in the format you need:

from sklearn.compose import make\_column\_transformer

from sklearn.preprocessing import MinMaxScaler, OneHotEncoder

transformer = make\_column\_transformer(

(MinMaxScaler(),

['sqft\_living', 'sqft\_lot','sqft\_above',

'sqft\_basement', 'house\_age']),

(OneHotEncoder(handle\_unknown='ignore'),

['bedrooms', 'bathrooms', 'floors',

'view', 'condition'])

)

Next, let’s separate features from the target variable, and split the dataset into training and testing parts. The train set will account for 80% of the data, and we’ll use everything else for testing:

from sklearn.model\_selection import train\_test\_split

X = df.drop('price', axis=1)

y = df['price']

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X, y, test\_size=0.2, random\_state=42

)

And finally, you can apply the transformations declared a minute ago. You’ll fit and transform the training features, and only apply the transformations to the testing set:

# Fit

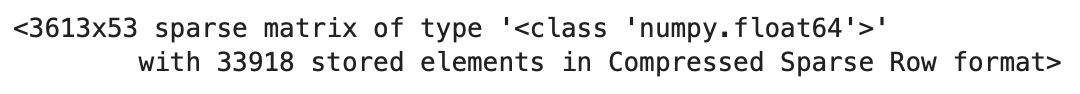
transformer.fit(X\_train)

# Apply the transformation

X\_train = transformer.transform(X\_train)

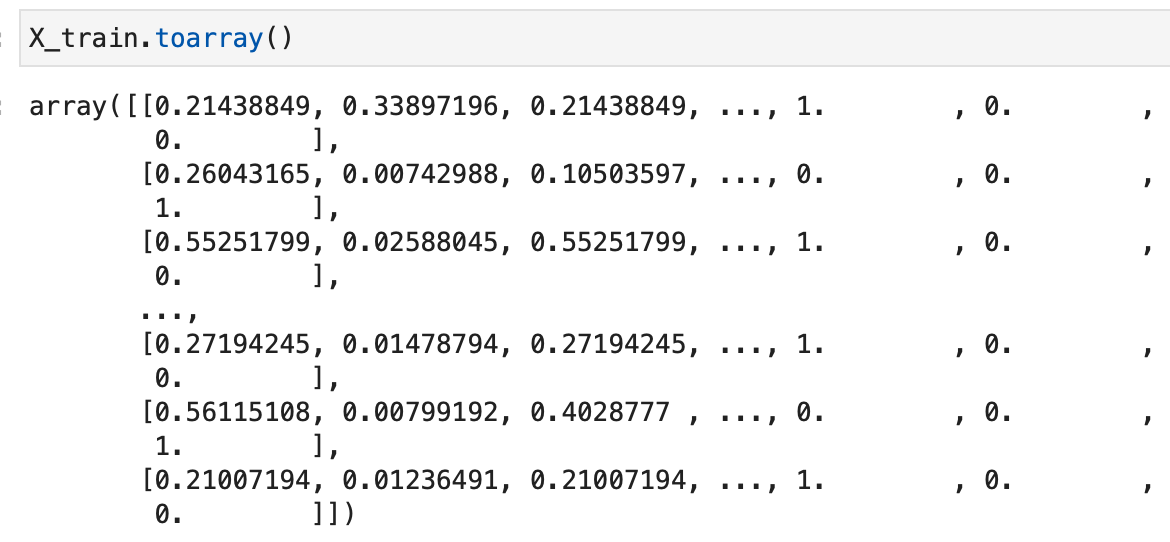
X\_test = transformer.transform(X\_test)

You won’t be able to inspect X\_train and X\_test directly, as they’re now stored as a sparse matrix:

Image 9 — Sparse matrix (image by author)

TensorFlow won’t be able to read that format, so you’ll have to convert it to a multidimensional Numpy array. You can use the toarray() function. Here’s an example:

X\_train.toarray()

Image 10 — Sparse matrix to Numpy array (image by author)

Convert both feature sets to a Numpy array, and you’re good to go:

X\_train = X\_train.toarray()

X\_test = X\_test.toarray()

Let’s finally train the model.

## **Training a regression model with TensorFlow**

You’ll now build a sequential model made of fully connected layers. There are many imports to do, so let’s get that out of the way:

import tensorflow as tf

from tensorflow.keras import Sequential

from tensorflow.keras.layers import Dense

from tensorflow.keras.optimizers import Adam

from tensorflow.keras import backend as K

### **Loss tracking**

You’re dealing with housing prices here, so the loss could be quite huge if you track it through, let’s say, *mean squared error*. That metric also isn’t very useful to you, as it basically tells you how wrong your model is in units squared.

You can calculate the square root of the MSE to go back to the original units. That metric isn’t supported by default, but we can declare it manually. Keep in mind that you’ll have to use functions from Keras backend to make it work:

def rmse(y\_true, y\_pred):

return K.sqrt(K.mean(K.square(y\_pred - y\_true)))

#### **Building a model**

And now you can finally declare a model. It will be a simple one, having just three hidden layers of 256, 256, and 128 units. Feel free to experiment with these, as there’s no right or wrong way to set up a neural network. These layers are then followed by an output layer of one node, since you’re predicting a numerical value.

You’ll then compile a model using the RMSE as a way to keep track of the loss and as an evaluation metric, and you’ll optimize the model using the Adam optimizer.

Finally, you’ll train the model on the training data for 100 epochs:

tf.random.set\_seed(42)

model = Sequential([

Dense(256, activation='relu'),

Dense(256, activation='relu'),

Dense(128, activation='relu'),

Dense(1)

])

model.compile(

loss=rmse,

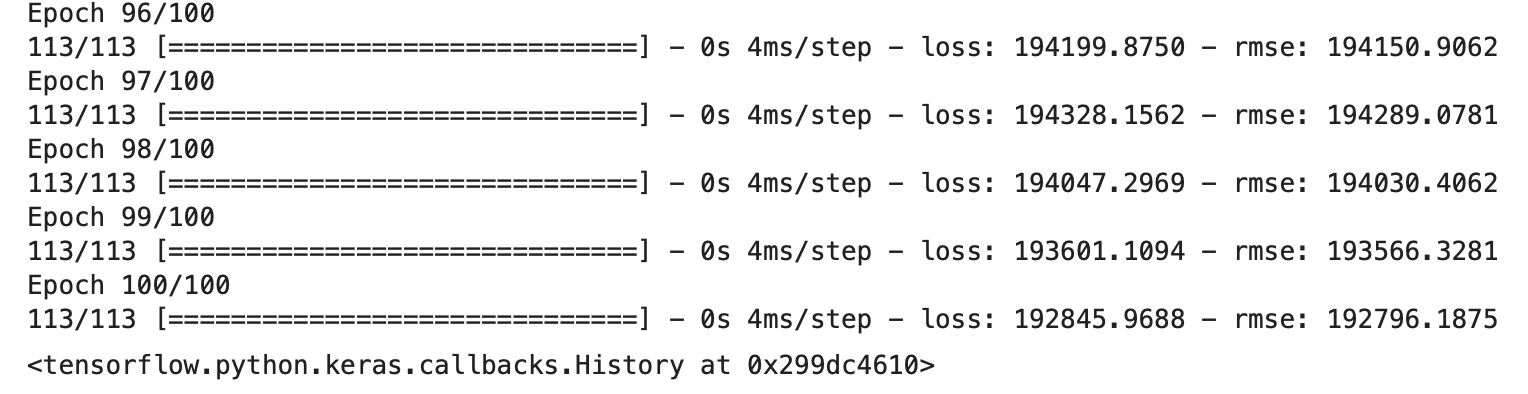
optimizer=Adam(),

metrics=[rmse]

)

model.fit(X\_train, y\_train, epochs=100)

The training should finish in a minute or so, depending on the hardware behind:

Image 11 — Regression model training with TensorFlow (image by author)

The final RMSE value on the training set is just above 192000, which means that for an average house, the model is wrong in the price estimate by $192000.

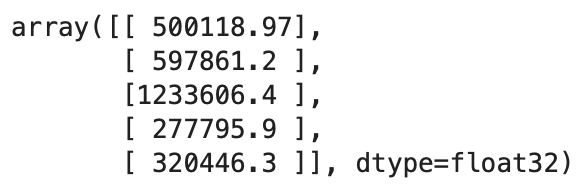
### **Making predictions**

You can make predictions on the test set:

predictions = model.predict(X\_test)

predictions[:5]

Here’s how the first five predictions look like:

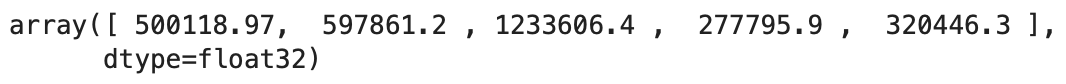
Image 12 — First 5 predictions (image by author)

You’ll have to convert these to a 1-dimensional array if you want to calculate any metrics. You can use the ravel() function from Numpy to do so:

predictions = np.ravel(predictions)

predictions[:5]

Here are the results:

Image 13 — First 5 predictions as a 1D array (image by author)

### **Model evaluation**

And now let’s evaluate the predictions on the test set by using RMSE:

rmse(y\_test, predictions).numpy()

You’ll get 191000 as an error value, which indicates the model hasn’t overfitted on the training data. You’d likely get better results with a more complex model training for more epochs. That’s something you can experiment with on your own.