Preparing data for modeling

Terence Parr
MSDS program
University of San Francisco

UNIVERSITY OF SAN FRANCISCO

Data prep overview

- Data sets must follow two fundamental rules before use in models:
  1. all data must be numeric
  2. there can't be any missing values
- Must delete or derive numeric features from nonnumeric features, such as strings, dates, and categorical variables
- Even with purely numeric data, there is potential cleanup work, such as deleting or replacing erroneous/missing entries or even deleting entire records that are outside our business rules

Data cleaning

Decide what you care about

- View all data cleaning operations through the lens of what exactly we want the model to do, as dictated by business or application
- For apartment data set, we want to predict apartment prices but
  - just for New York City
  - just for the reasonably-priced apartments
  - E.g., $1k < rent < $10k and GPS inside NYC
- Don't make decisions about "reasonable values" after looking at the data because we risk losing generality; *inappropriate data peeking* is a form of overfitting

See https://mlbook.explained.ai/prep.html and https://mlbook.explained.ai/bulldozer-intro.html

Why we care about noise, outliers

- Noise and outliers can lead to inconsistencies
- Zooming in on a small region of New York City there are two apartments with similar features but that are much more expensive:

	bedrooms	bathrooms	street_address	price
39939	1	1.0000	west 54 st & 8 ave	2300
21711	1	1.0000	300 West 55th Street	2400
15352	1	1.0000	300 West 55th Street	3350
48274	1	1.0000	300 West 55th Street	3400
29665	1	1.0000	333 West 57th Street	1070000
30689	1	1.0000	333 West 57th Street	1070000

??

- Could be missing a key feature (view or parking?); sale not rent price?
- Could be errors or simply outliers but such inconsistent data leads to inaccurate predictions
- RFs predict the average price for all apartments in same feature space so predictions for these will be way off

To begin: take a quick sniff of the data

- Identify:
  - column names
  - their datatypes
  - whether target column has numeric values or categories
- Look inside the values of string columns as we might want to break them into multiple columns

bathrooms	1.5000
bedrooms	3
building_id	53a5b119ba8f7b61d4e010512...
created	2016-06-24 07:54:24
description	A Brand New 3 Bedroom 1.5...
display_address	Metropolitan Avenue
features	[]
latitude	40.7145
listing_id	7211212
longitude	-73.9425
manager_id	5ba989232d0489da1b5f2c45f...
photos	['https://photos.renthop....
price	3000
street_address	792 Metropolitan Avenue
interest_level	medium

Look at data ranges with describe()

- 10 bathrooms? 0 bedrooms? Wow.
- Longitude and latitude of 0?
- Apts that are $43 and $4,490,000 / month? Wow

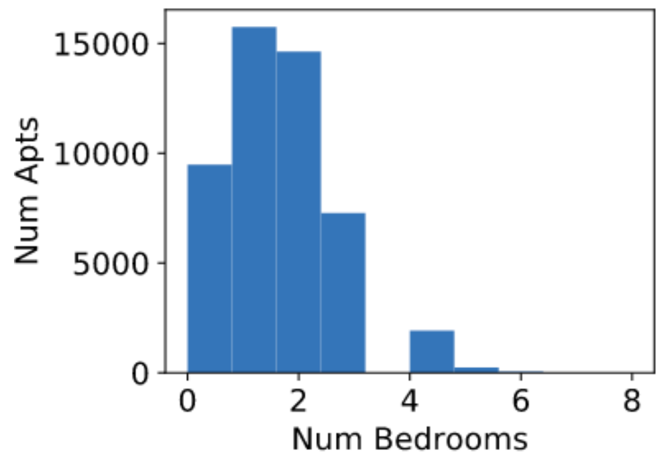
	bathrooms	bedrooms	longitude	latitude	price
count	49352.0000	49352.0000	49352.0000	49352.0000	49352.0000
mean	1.2122	1.5416	-73.9557	40.7415	3830.1740
std	0.5014	1.1150	1.1779	0.6385	22066.8659
min	0.0000	0.0000	-118.2710	0.0000	43.0000
25%	1.0000	1.0000	-73.9917	40.7283	2500.0000
50%	1.0000	1.0000	-73.9779	40.7518	3150.0000
75%	1.0000	2.0000	-73.9548	40.7743	4100.0000
max	10.0000	8.0000	0.0000	44.8835	4490000.0000

Check distributions

- Only a few outlier apartments with > 6 bedrooms/bathrooms

```
print(df_num.bathrooms.value_counts())
```

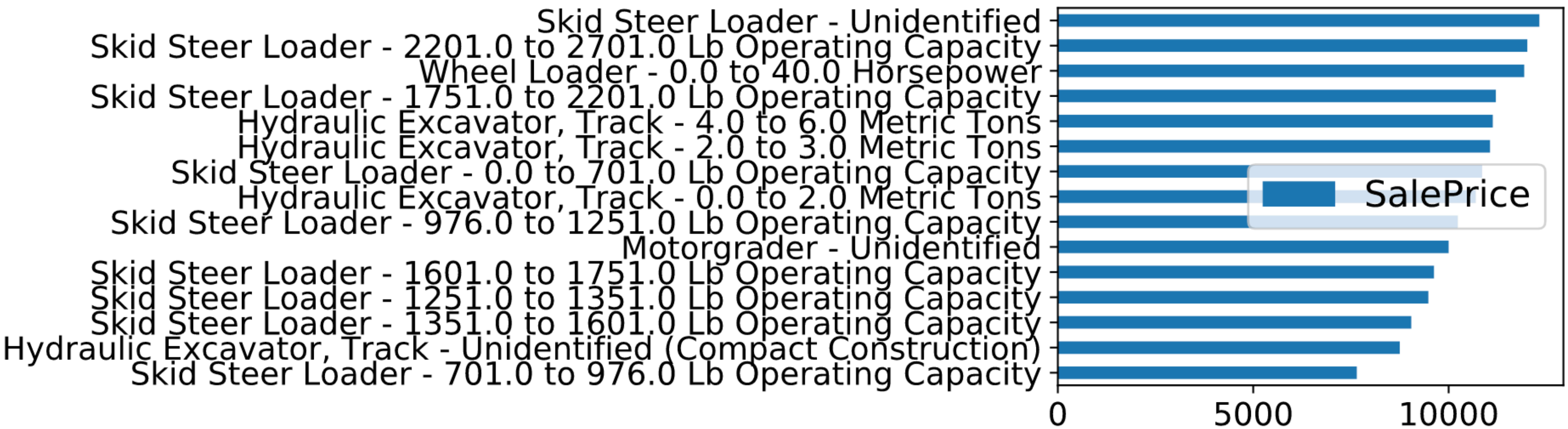
```
1.0      39422
2.0       7660
3.0        745
1.5        645
0.0        313
2.5        277
4.0        159
3.5         70
4.5         29
5.0         20
5.5          5
6.0          4
6.5          1
10.0         1
7.0          1
Name: bathrooms, dtype: int64
```



Not many outliers:
len(df[df.price>10_000]) = 878

Check variable-target relationships too

- Sometimes checking the relationship between each variable and the target can be illuminating; e.g., here is a categorical variable versus bulldozer sale price:



We should try extracting useful info from feature as it is predictive

Let’s clean up

- Filter data per business goals
- In NY only:

```
df_clean = df[(df['latitude']>40.55) & (df['latitude']<40.94) &
              (df['longitude']>-74.1) & (df['longitude']<-73.67)]
```

- Reasonable prices:

```
df_clean = df[(df.price>1_000) & (df.price<10_000)]
```

- If column known to be corrupted or useless, can just delete it; e.g., from bulldozer data set:

```
del df[‘MachineID’]
```

More clean up

- Sold before manufactured? (ask stakeholders) Can adjust date or delete if there few enough of those records
- Some columns are read in as numbers but are really categorical; e.g., bulldozer **auctioneerID**; we can set to strings (affects missing data handling):

```
df['auctioneerID’] = df['auctioneerID'].astype(str)
```

	SalePrice	YearMade	saledate
36156	27000	1996.0	1995-03-31
36417	11500	1996.0	1995-04-08
34303	70000	1996.0	1995-01-25

?

	auctioneerID
0	6.0
1	2.0
2	3.0
3	1.0
4	NaN

Don’t replace with median (to impute value)

Normalization

- Some columns are shown as strings but are numbers; e.g., bulldozer **Tire_Size**; delete double-quote and then convert column to numbers
- Bulldozer **Stick_length** is more complicated but could still be normalized to inches rather than string
- Bulldozer **Enclosure** has "EROPS w AC" and "EROPS AC"; normalize to one or other: `df['Enclosure'].replace('EROPS w AC','EROPS AC')`

	Tire_Size
0	None
1	23.5
2	14"
3	None or Unspecified
4	17.5"

	Stick_Length
0	None
1	None or Unspecified
2	10' 2"
3	9' 6"

Find missing data indicators

- Missing values are np.NaN after loading with pandas
- BUT, some are physically-present numbers or strings that actually represent missing values:
  - Rent dataset: Some **longitude/latitude** values are 0 (off the west coast of Africa?)
  - Bulldozer dataset: strings like **Tire_Size** have “None or Unspecified”
  - Bulldozer **fiModelSeries** has “#Name?”
- Replace those with NaN; for example:

```
df.loc[df['Tire_Size']=='None or Unspecified’,
       'Tire_Size'] = np.nan
```

	fiModelSeries
0	5
1	SeriesII
2	#NAME?
3	ZTS

	Tire_Size
0	None
1	23.5
2	14"
3	None or Unspecified
4	17.5"

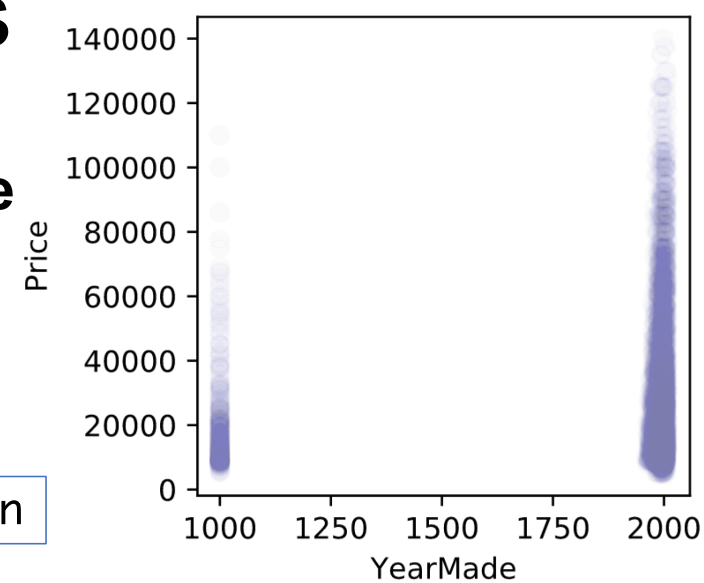
More missing data indicators

- Something fishing with Bulldozer **YearMade**
- **YearMade**=1000 must mean unknown (or don't ask it's age! haha)
- Replace weird dates with NaN (missing):

```
df.loc[df.YearMade<1950, 'YearMade'] = np.nan
```

- Bulldozer **Backhoe_Mounting** should be boolean; normalize, convert to true/false, set type

	Backhoe_Mounting
0	None or Unspecified
1	None
2	Yes



Encoding non-numeric variables

Encoding date variables

- Date columns in datasets are often predictive of target variables
- E.g., in bulldozer data set, the date of sale and the year of manufacture together are strongly predictive of the sale price
- **General procedure**:
  - Shatter date columns into constituent components such as: year, month, day, day of week (1..7), day of year (1..365), and even things like “end of quarter” and “end of month”
  - After extracting the components, convert datetime64 column to integer with number of seconds since 1970 (unix time)
- Can add business holidays, big snowstorm days, …

See https://mlbook.explained.ai/bulldozer-feateng.html

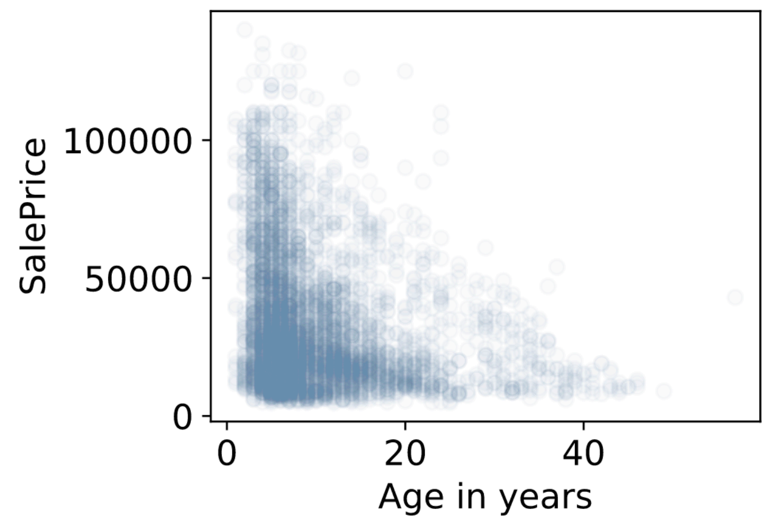
Date-related computations also useful

- E.g., bulldozer should add age:

```
df['age'] = df['saleyear'] - df['YearMade’]
```

  Makes life easier on the RF model

- Can try introducing variables like “days since event E” (e.g., “days since we had a big sale”) or other cumulative counts, averages, sums, etc…



Sample date conversion code

```
def df_split_dates(df,colname):
    df["saleyear"] = df[colname].dt.year
    df["salemonth"] = df[colname].dt.month
    df["saleday"] = df[colname].dt.day
    df["saledayofweek"] = df[colname].dt.dayofweek
    df["saledayofyear"] = df[colname].dt.dayofyear
    df[colname] = df[colname].astype(np.int64) # convert to seconds since 1970
```

	0
saledate	1232668800000000000
saleyear	2009
salemonth	1
saleday	23
saledayofweek	4
saledayofyear	23

Encoding categorical vars

- Categorical variables are named elements like US states or arbitrary strings like addresses; pandas calls them objects
- We distinguish between *ordinal* (low/high) and *nominal* (zip code) categoricals
- First, convert ordinals to appropriate ordered ints
- Then, make a choice about nominals:
  - One-hot encode (dummy variables)
  - Label encode (category → unique integer)
  - Frequency encode
  - Break up string into more useful columns
  - Advanced: embeddings, target encoding, …

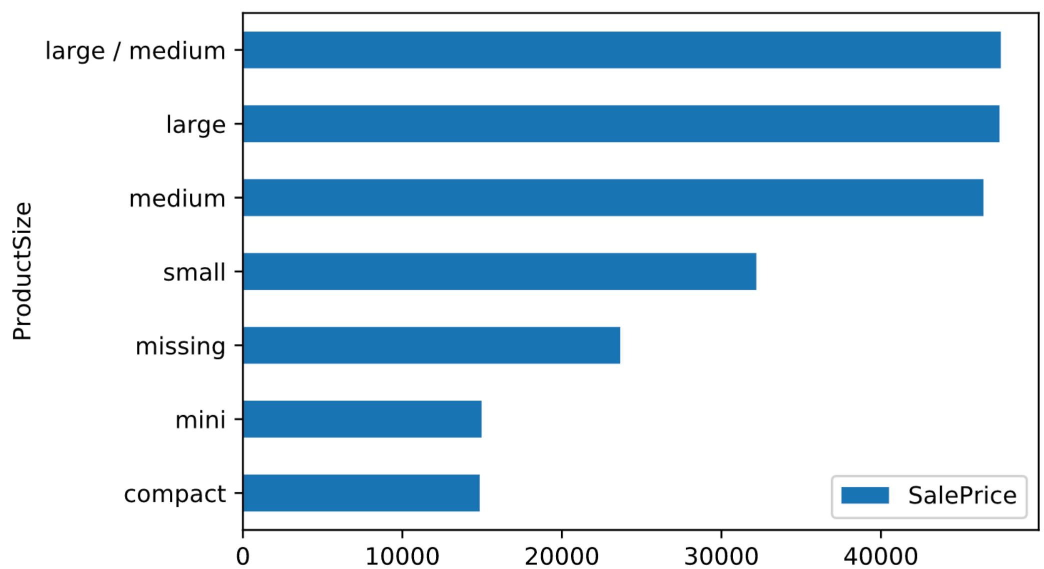
MachineHoursCurrentMeter	float64
saledate	datetime64[ns]
Coupler	object
Tire_Size	object
Tip_Control	object
Hydraulics	object

> The easy way to remember the difference between ordinal and nominal variables is that ordinal variables have order and nominal comes from the word for "name" in Latin (*nomen*) or French (*nom*).

See https://mlbook.explained.ai/catvars.html and https://mlbook.explained.ai/bulldozer-feateng.html

Start by converting ordinals

- Bulldozer **ProductSize** categorical is ordinal not nominal so convert it to integers with appropriate order
- Marginal plot makes it look very predictive



encoding
5
4
3
2
0
1
1

Articles on web say mini/compact are same

	ProductSize
0	NaN
1	small
2	large / medium
3	medium
4	compact
5	mini
6	large

Ordinal encoding mechanics

	interest_level
0	medium
1	low
2	high

- Apply a dictionary, mapping name to ordered value
- E.g., rent data set:

```
df['interest_level'] = \
    df['interest_level'].map({'low':1,'medium':2,'high':3})
```

- For RFs, only the order matters not the scale so {'low':10,'medium':20,'high':30} would also work

One-hot encoding (dummy variables)

Note: RFs don't require dummy variables but sometimes dummies are useful

- Instead of a number, the “hot” position indicates the category
- Notice how the missing value ends up with none hot (all 0s)

	Dept
0	Math
1	CS
2	Physics
3	

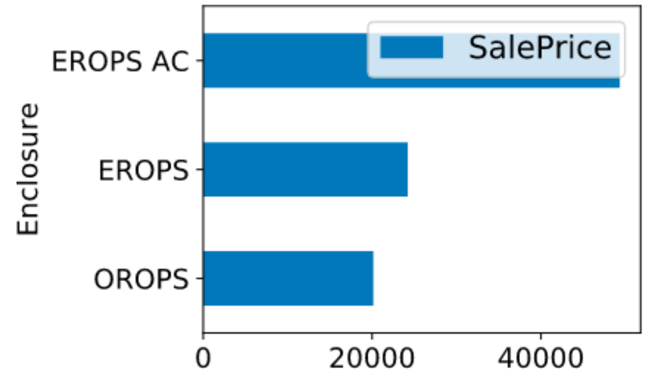
	Dept	CS	Math	Physics
0	Math	0	1	0
1	CS	1	0	0
2	Physics	0	0	1
3		0	0	0

```
onehot = pd.get_dummies(df['Dept’])
df_encoded = pd.concat([df, onehot], axis=1)
del df_encoded[‘Dept’]
```

(Some people differentiate between one-hot and dummy vars.)

When to one-hot encode

- Don't one-hot encode when there are many cat levels otherwise you will end up with thousands of columns in your data set
- That slows down training speed and usually doesn't help (for RFs)
- One-hot encoding is worth it for cat vars that are strongly predictive (if there are few levels)
- E.g., "EROPS AC" gets, on average, twice the price of the other bulldozers meaning air-conditioning is important



Frequency encoding

- Sometimes we can extract some meaning from the nominals
- Convert categories to the frequencies with which they appear in the training
- E.g., rent data: might be predictive power in the number of apartments managed by a particular manager

manager_id	count
e6472c7237327dd3903b3d6f6a94515a	2509
6e5c10246156ae5bdcd9b487ca99d96a	695
8f5a9c893f6d602f4953fcc0b8e6e9b4	404
62b685cc0d876c3a1a51d63a0d6a8082	396
cb87dadbca78fad02b388dc9e8f25a5b	370

```
managers_count = df['manager_id'].value_counts()
df['mgr_apt_count'] = df['manager_id'].map(managers_count)
```

Label encoding categoricals

- If you can't extract more useful information from a nominal variable, label encode it
- There are more advanced techniques such as embeddings, target encoding but we'll leave those to another class
- **Result**: each category becomes a unique numeric value where missing becomes 0 and other categories are 1..n
- We ignore the fact that the categories are not really ordered

Label encoding mechanics

- Convert string column to ordered categorical
- Replace categories with cat code + 1
- NaN gets cat code -1 so +1 means missing = 0

```
def df_string_to_cat(df):
    for col in df.columns:
        if is_string_dtype(df[col]):
            df[col] = df[col].astype('category')
            df[col] = df[col].cat.as_ordered()

def df_cat_to_catcode(df):
    for col in df.columns:
        if is_categorical_dtype(df[col]):
            df[col] = df[col].cat.codes + 1
```

	Name
0	Xue
1	
2	Tom

	Name	catcodes
0	Xue	1
1		-1
2	Tom	0

	Name	catcodes
0	2	1
1	0	-1
2	1	0

The unreasonable effectiveness of label encoding categorical variables

- Why is it “legal” to convert all of those unordered (nominal) categorical variables to ordered integers?
- RF models can still partition such converted categorical features in a way that is predictive
- Might require more complex / bigger tree
- Definitely not appropriate for models doing math on variables, such as linear models (which require one-hot encoding)
- In practice, label encoding categorical variables is surprisingly effective
- Some RF models do subset comparisons not int comparisons

Dealing with missing data

UNIVERSITY OF SAN FRANCISCO

Real data sets are often full of holes

- Here are some stats on Bulldozer data set

	percent missing
SalesID	0.0000
SalePrice	0.0000
MachineID	0.0000
ModelID	0.0000
datasource	0.0000
YearMade	0.0000
auctioneerID	5.1747
MachineHoursCurrentMeter	64.7178
saledate	0.0000
Coupler	46.8269
Tire_Size	76.3297
Tip_Control	93.6982
Hydraulics	20.1663
Ripper	73.9670

Missing categorical data

- Missing categorical values are dealt with automatically because of the label-encoding process
- We convert categories to unique integer values and missing values, np.nan, become category code 0 and all other categories are codes 1 and above
- In other words, “missing” is just another category hardcoded to 0

Missing numeric data

- Don’t delete columns/rows with missing values; destroys info!
- Don’t *just* replace missing values; destroys fact they were missing
- E.g., missing **YearMade** could mean “ancient”
- E.g., missing **Employer** on loan app could mean ”unemployed” (or missing **YearsOfEducation** might mean ”no college degree”)
- We still must fill in values in order to train a model, however, and we don’t want to skew the column distribution by replacing with 0 or 999999 or some other anomalous value

Imputing missing numeric values

- Dealing with missing numeric values requires a new column and replacement of np.nans:
  1. For column *x*, create a new boolean column *x*_na where *x_na*[i] is true if *x*[i] is missing.
  2. Replace missing values in column *x* with the median of all *x* values in that column.

```
def fix_missing_num(df, colname):
    df[colname+'_na'] = pd.isnull(df[colname])
    df[colname].fillna(df[colname].median(), inplace=True)
```

	YearMade
0	1995.0000
1	2001.0000
2	

→

	YearMade	YearMade_na
0	1995.0000	False
1	2001.0000	False
2	1998.0000	True

Supporting academic work

- See "**On the consistency of supervised learning with missing values**" https://hal.archives-ouvertes.fr/hal-02024202v2:

  "*A striking result is that the widely-used method of imputing with the mean prior to learning is consistent when missing values are not informative*."

  "*When missingness is related to the prediction target, imputation does not suffice and it is useful to add indicator variables of missing entries as features.*"

Rectifying training and validation sets

- Replacing missing values, encoding categorical variables, etc… introduces synchronization issues between training and validation/test sets
- Key rules:
  1. Transformations must be applied to features consistently across data subsets
  2. Transformations of validation/test sets can only use data derived from training set
- To follow those rules, we have to remember all transformations done to the training set for later application to the validation and test sets.
- That means tracking the median of all numeric columns, all category-to-code mappings, frequency encodings, and one-hot'd categories
- Special care is required to ensure that one-hot encoded variables use the same name and number of columns in the training and testing sets.
- Beware: it's easy to screw up the synchronization!

For details, see https://mlbook.explained.ai/bulldozer-testing.html