[*Book contents*](http://mlbook.explained.ai/) *Work in progress*  
Book version 0.4

**7 Exploring and Cleaning the Bulldozer Dataset**

[Terence Parr](http://parrt.cs.usfca.edu/) and [Jeremy Howard](http://www.fast.ai/about/#jeremy)

Copyright © 2018-2019 Terence Parr. All rights reserved.  
*Please don't replicate on web or redistribute in any way.*  
This book generated from markup+markdown+python+latex source with [Bookish](https://github.com/parrt/bookish).

You can make **comments or annotate** this page by going to the [annotated version of this page](https://via.hypothes.is/https:/mlbook.explained.ai/bulldozer-intro.html). You'll see existing annotated bits highlighted in yellow. They are *PUBLICLY VISIBLE*. Or, you can send comments, suggestions, or fixes directly to [Terence](mailto:parrt@cs.usfca.edu).

**Contents**

* [Loading the bulldozer data](https://mlbook.explained.ai/bulldozer-intro.html#sec:7.1)
* [Taking an initial look at the data](https://mlbook.explained.ai/bulldozer-intro.html#sec:7.2)
* [Baseline model](https://mlbook.explained.ai/bulldozer-intro.html#sec:7.3)
* [Cleaning up](https://mlbook.explained.ai/bulldozer-intro.html#sec:7.4)
* [Dealing with missing data](https://mlbook.explained.ai/bulldozer-intro.html#sec:missing)
  + [Replacing missing categorical values](https://mlbook.explained.ai/bulldozer-intro.html#sec:7.5.1)
  + [Replacing missing numeric values](https://mlbook.explained.ai/bulldozer-intro.html#sec:7.5.2)
* [Training a model with all features](https://mlbook.explained.ai/bulldozer-intro.html#sec:7.6)
* [Summary](https://mlbook.explained.ai/bulldozer-intro.html#sec:7.7)

We've learned a great deal so far about preparing data, feature engineering, and training models, but the apartment rent dataset is relatively small with few features. Over the next few chapters, we're going to explore and build models for a [bulldozer auction prices dataset](https://www.kaggle.com/c/bluebook-for-bulldozers) from Kaggle that has 8 times as many observations and 52 features. A large dataset presents a new set of problems, such as figuring out which features to focus on for feature engineering and being able to train and test models quickly. The bulldozer dataset is also rife with missing values. After working through the process we've laid out for this dataset, though, your final model will perform near the top of the leaderboard for the (now closed) bulldozer competition.

Another wrinkle with this dataset is that records represent bulldozer sales, and prices can drift over time due to inflation, financial crises, and so on. As a general rule, we can't train and test models for time-sensitive data the same way we do for time-insensitive data. The out-of-bag (OOB)  score isn't usually appropriate because OOB scores measure performance only within the training data period, not against future predictions. It's more appropriate to sort a dataset by date and then take the last, say, 20% as a hold-out validation set. That leaves the first 80% as the training set, which we use to train the model. Evaluating the performance of the model on the validation set gives a much more accurate estimate of model generality than the OOB score. That said, in order to tackle this bulldozer problem in pieces, we're going to start out measuring model performance using the OOB  score, dramatically simplifying our process. Just keep in mind that the OOB score is overestimating model performance.

In this chapter, we'll examine the bulldozer data set, normalize and cleanup various values, fill in missing values, and then train an initial model. Next, in **Chapter 8** *Bulldozer Feature Engineering*, we'll learn a few more encodings for categorical variables and improve the features identified as important by the initial model. After we figure out how to prepare this dataset, we'll explore in **Chapter 9** *Train, Validate, Test* how to properly prepare validation and test sets for use in tuning the model and getting a true estimate of model generality.

**7.1 Loading the bulldozer data**

Our first step is to grab the bulldozer dataset from Kaggle's [Blue Book for Bulldozers](https://www.kaggle.com/c/bluebook-for-bulldozers/data) competition. Download files Train.zip (and uncompress), Valid.csv, and ValidSolution.csv into your data directory beneath the directory where you launch Jupyter. (You must be a registered Kaggle user and logged in.) The Train.csv file you get after uncompressing Train.zip is 116M, which takes about 35 seconds to load using Pandas' read\_csv() function. That load time would be unbearably slow while staring at the screen and would make it harder to iterate quickly on our models. Instead, we're going to use the [feather data format](https://github.com/wesm/feather), which lets us load the data in about one second. The [prep-bulldozer.py](https://mlbook.explained.ai/data/prep-bulldozer.py) script (from this book's [data](https://mlbook.explained.ai/data/index.html) directory) loads the training CSV data (for the first and only time), splits out a validation set, and saves them both using the fast feather format. The script also merges Valid.csv, and ValidSolution.csv into a single test dataframe and saves it for later use. From the command-line and in your data directory, execute the following to create the data files we need.

$ cd data $ pip install feather-format $ python prep-bulldozer.py Created bulldozer-train-all.feather Created bulldozer-train.feather Created bulldozer-valid.feather Created bulldozer-test.feather

1Don't forget the [notebooks](https://mlbook.explained.ai/notebooks/) aggregating the code snippets from the various chapters.

For the next three chapters, we'll use bulldozer-train.feather as our data starting point. To start the coding process, create a notebook in the directory above data and paste in our usual preamble:1

import pandas as pd import numpy as np from sklearn.ensemble import RandomForestRegressor from rfpimp import \* # feature importance plot

2If you get an error “read\_feather() got an unexpected keyword argument 'nthreads',” then try:  
import feather  
feather.read\_dataframe("data/bulldozer-train.feather")

Anytime we need a fresh copy of the data, we can load it like this:2

df\_raw = pd.read\_feather("data/bulldozer-train.feather") df = df\_raw.copy()

It's a good idea to keep the original data around in df\_raw so that we can undo any data transformations that end up being unhelpful.

**7.2 Taking an initial look at the data**

When inspecting a dataset for the first time, look for this key summary information: the column names, column datatypes, sample data elements, and how much data is missing. Here's a handy function to sniff a dataframe and return a different dataframe containing a summary where each row describes a column in the original dataframe:

def sniff(df): with pd.option\_context("display.max\_colwidth", 20): info = pd.DataFrame() info['sample'] = df.iloc[0] info['data type'] = df.dtypes info['percent missing'] = df.isnull().sum()\*100/len(df) return info.sort\_values('data type')

You can call sniff(df) from your notebook to get a complete summary, but in the interest of space, here are just the first 14 entries:

sniff(df).head(14)

|  | **sample** | **data type** | **percent missing** |
| --- | --- | --- | --- |
|  |  |  |  |
| **SalesID** | 1646770 | int64 | 0.0000 |
| **SalePrice** | 9500 | int64 | 0.0000 |
| **MachineID** | 1126363 | int64 | 0.0000 |
| **ModelID** | 8434 | int64 | 0.0000 |
| **datasource** | 132 | int64 | 0.0000 |
| **YearMade** | 1974 | int64 | 0.0000 |
| **auctioneerID** | 18.0000 | float64 | 5.1747 |
| **MachineHoursCurrentMeter** |  | float64 | 64.7178 |
| **saledate** | 1989-01-17 00:00:00 | datetime64[ns] | 0.0000 |
| **Coupler** |  | object | 46.8269 |
| **Tire\_Size** |  | object | 76.3297 |
| **Tip\_Control** |  | object | 93.6982 |
| **Hydraulics** | 2 Valve | object | 20.1663 |
| **Ripper** | None or Unspecified | object | 73.9670 |

We can learn a lot just from this quick sniff. There are three kinds of data: numeric, date time objects, and strings (object). Some columns are complete, but others have missing data, including column Tip\_Control that is 94% missing. Some values are just plain missing (represented as either the None object or “not a number” np.nan in Python), but other “missing” values are actually physically-present strings like “None or Unspecified”.

Columns such as SalesID and ModelID are represented as integers (int64), but they are really nominal categorical variables. Model 8434 is not somehow greater than model 8433. There are also columns represented as strings that contain numeric values, such as Hydraulics (“2 Valve”). Other columns are represented as strings but are actually purely numeric but with units such as feet or inches. For example, the values in column Tire\_Size should be converted to just the number of inches:

print(df['Tire\_Size'].unique())

[None '14"' 'None or Unspecified' '20.5' '23.5' '26.5' '17.5' '29.5' '13"' '20.5"' '23.5"' '17.5"' '15.5' '15.5"' '7.0"' '23.1"' '10"' '10 inch']

It's a good idea to look at the unique set of values for the other columns too as you experiment with this dataset. The next step in our exploration is to train a model.

**7.3 Baseline model**

While we only have a few numeric columns out of the 52 total, and some of those values are missing, it's still a good idea to train a model early on in our process. First, it tells us how long training the model takes. If training time is significant, we should consider working with a subset of the data. Second, it gives us an initial appraisal of the strength of the relationship between numeric features and the SalePrice target variable. The OOB  from this initial model is our lower bound, so if it's pretty good, we can be optimistic about the performance of our model after feature engineering. Finally, a feature importance graph derived from the model helps to focus our cleanup efforts on the most predictive columns.

Let's identify the features represented as numbers so far are:

basefeatures = ['SalesID', 'MachineID', 'ModelID', 'datasource', 'YearMade', # some missing values but use anyway: 'auctioneerID', 'MachineHoursCurrentMeter']

3We've chosen to create RFs with 50 trees because it gives a stable and accurate  score while not requiring too much processing time.

We can reuse the test() function from **Chapter 6** *Categorically Speaking* to train an RF and print the OOB  score:3

def test(X, y, n\_estimators=50): rf = RandomForestRegressor(n\_estimators=n\_estimators, n\_jobs=-1, oob\_score=True) rf.fit(X, y) oob = rf.oob\_score\_ n = rfnnodes(rf) h = np.median(rfmaxdepths(rf)) print(f"OOB R^2 {oob:.5f} using {n:,d} tree nodes with {h} median tree height") return rf, oob

Columns auctioneerID and MachineHoursCurrentMeter have some missing values, represented as Numpy np.nan, but we can flip missing values to zeros as an expedient with function call fillna(0):

X, y = df[basefeatures], df['SalePrice'] X = X.fillna(0) # flip missing numeric values to zeros rf, oob\_baseline\_initial = test(X, y)

OOB R^2 0.78075 using 22,500,374 tree nodes with 56.0 median tree height

That OOB score is not horrible and hints that there is a strong relationship to capture in this dataset.

Unfortunately, training the model via function fit() takes about 25 seconds, which would seem like an eternity as we repeatedly transformed data and retrained the model. To reduce training time, we could subsample the dataframe with df.sample(n=100\_000), and that's how we'd do it if this were not time-sensitive data. Instead, let's grab the last 100,000 records (which are sorted by date), taking advantage of the fact that more recent data will be better at predicting the near future:

df = df.iloc[-100\_000:] # take only last 100,000 records

After reducing the size of the dataframe, repeat the steps to train the model:

X, y = df[basefeatures], df['SalePrice'] X = X.fillna(0) rf, oob\_baseline = test(X, y)

OOB R^2 0.84512 using 5,556,904 tree nodes with 45.0 median tree height

Training the model now takes only about 5 seconds down from 25 seconds and, as a bonus, the  of 0.845 is much better than the previous 0.781.

Now, let's see which features the model thinks are most predictive of bulldozer sale price:

I = importances(rf, X, y) plot\_importances(I)

The most important features are consistent with what we'd expect when evaluating a vehicle's value: what kind (model) of bulldozer it is, when it was made, how long it's been in use, etc.

Now that we have a good understanding of the data and a baseline model, let's start cleaning up the data.

**7.4 Cleaning up**

The easiest things to fix in the cleanup process are the small administrative details like changing column datatypes and deleting unusable columns, so let's start with those. According to the data description at Kaggle, SalesID is a unique identifier for a particular transaction. This is clearly not predictive as a SalesID value will never be seen again as a feature, so we can remove it. We can also remove MachineID because this variable has [errors and inconsistencies](https://www.kaggle.com/c/bluebook-for-bulldozers/discussion/3694); besides, it's not strongly-predictive according to our feature importance graph. Our first step is then:

del df['MachineID'] # dataset has inconsistencies del df['SalesID'] # unique sales ID so not generalizer

The auctioneerID column values look like numbers, but they are really categorical variables, specifically nominal variables that have no order:

print(df['auctioneerID'].unique())

[nan 5. 2. 27. 1. 23. 3. 4. 20. 7. 8. 12. 10. 6. 21. 13. 9. 18. 99. 16. 14. 19. 28. 15. 22. 25. 17. 11. 24. 26. 0.]

Just to make this clear, let's change the data type to be string:

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

If we leave this as a number, our process below for dealing with missing numeric values would replace the missing auctioneerID values with the median auctioneer ID, which is clearly meaningless.

So much for the numbers, let's take a look at the string-valued columns. Some are nice and tidy such as:

print(df['ProductGroup'].unique())

['TTT' 'TEX' 'WL' 'SSL' 'BL' 'MG']

but others have missing values:

print(df['Drive\_System'].unique())

[None 'Two Wheel Drive' 'Four Wheel Drive' 'No' 'All Wheel Drive']

and others have physically-present strings to mean “missing:”

print(df['Backhoe\_Mounting'].unique())

['None or Unspecified' None 'Yes']

Columns fiSecondaryDesc and fiModelSeries even have a value of “#NAME?”. This dataset has multiple ways to say “missing” or “unspecified,” but our model won't be able to figure the equivalencies on its own. We need to normalize all of these strings so that None, None or Unspecified, and #NAME? all mean “missing.” Let's encapsulate this equivalents in a function that transforms the dataframe so only np.nan means missing:

from pandas.api.types import is\_string\_dtype, is\_object\_dtype def df\_normalize\_strings(df): for col in df.columns: if is\_string\_dtype(df[col]) or is\_object\_dtype(df[col]): df[col] = df[col].str.lower() df[col] = df[col].fillna(np.nan) # make None -> np.nan df[col] = df[col].replace('none or unspecified', np.nan) df[col] = df[col].replace('none', np.nan) df[col] = df[col].replace('#name?', np.nan) df[col] = df[col].replace('', np.nan)

After calling df\_normalize\_strings(df), all of the different ways to say none are collapsed to np.nan and strings all are lowercase:

df\_normalize\_strings(df) print(df['Drive\_System'].unique()) print(df['Backhoe\_Mounting'].unique())

[nan 'two wheel drive' 'four wheel drive' 'no' 'all wheel drive'] [nan 'yes']

Some strings are actually numeric values, but include unit names or symbols that force the dataframe to treat them as strings:

print(df['Tire\_Size'].unique()) print(df['Undercarriage\_Pad\_Width'].unique())

[nan '26.5' '20.5' '17.5' '23.5' '14"' '13"' '29.5' '17.5"' '15.5"' '20.5"' '15.5' '23.5"' '7.0"' '10"' '23.1"'] [nan '36 inch' '24 inch' '20 inch' '34 inch' '26 inch' '30 inch' '28 inch' '32 inch' '16 inch' '31 inch' '18 inch' '22 inch' '33 inch' '14 inch' '27 inch' '25 inch' '15 inch']

It's a simple matter to strip off the " and inch characters to convert these two columns to numeric values. Here's a function to convert a column of strings to a numeric column by extracting any integer or floating-point numbers on the front of the string:

def extract\_sizes(df, colname): df[colname] = df[colname].str.extract(r'([0-9.]\*)', expand=True) df[colname] = df[colname].replace('', np.nan) df[colname] = pd.to\_numeric(df[colname])

extract\_sizes(df, 'Tire\_Size') extract\_sizes(df, 'Undercarriage\_Pad\_Width') print(df['Tire\_Size'].unique()) print(df['Undercarriage\_Pad\_Width'].unique())

[ nan 26.5 20.5 17.5 23.5 14. 13. 29.5 15.5 7. 10. 23.1] [nan 36. 24. 20. 34. 26. 30. 28. 32. 16. 31. 18. 22. 33. 14. 27. 25. 15.]

There are two other columns that are numeric in nature but would be more complicated to parse apart (as they have both feet and inch units):

print(df['Blade\_Width'].unique()) print(df['Stick\_Length'].unique())

[nan "12'" "14'" "13'" "16'" "<12'"] [nan '10\' 6"' '9\' 6"' '9\' 7"' '10\' 2"' '12\' 8"' '12\' 10"' '9\' 10"' '9\' 8"' '11\' 0"' '10\' 10"' '8\' 6"' '9\' 5"' '14\' 1"' '11\' 10"' '6\' 3"' '12\' 4"' '8\' 2"' '8\' 10"' '8\' 4"' '15\' 9"' '13\' 10"' '13\' 7"' '15\' 4"' '19\' 8"']

The Blade\_Width column even has a range in the form of <12'. It's better to leave these as strings, which we'll treat as categorical variables when prepping the data for use in a model.

**7.5 Dealing with missing data**

Missing data in CSV files is often indicated as physically missing (two commas in a row like “,,”), but some records use physically-present string values such as None or Unspecified. Some files use special indicator numbers to represent missing numeric values, such as -1 or 0. Pandas uses Numpy's np.nan (“not a number”) to represent values missing from data files in memory, for both numeric and string data types. Pandas stores physically-present numeric and string values in files as-is in memory. The point is that the definition of missing is ambiguous and depends on the dataset. That's why we normalized strings in the previous section so that only np.nan indicates “missing.” We'll do the same for numeric indicator values in this section.

Once the entire dataframe has a single definition of missing value, we still have to do something intelligent with these holes. Models can't train on “not a number” values. Our recipe to handle missing values looks like this: For numeric columns, we replace missing values with the median of that column and introduce a new boolean column that is true for any record where we replace a missing value. (Statisticians call replacing missing values *imputation*.) The strategy for nonnumeric columns simply is to leave them as-is with np.nan values. Our default string/categorical variable encoding is to label encode them, which will automatically replace np.nan values with zeros. (Label encoding assigns a unique integer for every unique string or category value.) Dealing with missing nonnumeric values is easiest so let's start by seeing how that works.

**7.5.1 Replacing missing categorical values**

In **Section 6.2** *Encoding categorical variables*, we converted the string display\_address column to numeric values by converting the column to an ordered categorical column and then replacing the categories with their category integer codes + 1. Pandas represents np.nan with category code -1 and so adding one shifts np.nan to 0 and all category codes to be 1 and above. For convenience, let's create two functions that implement our label-encoding strategy:

from pandas.api.types import is\_categorical\_dtype def df\_string\_to\_cat(df): for col in df.columns: if is\_string\_dtype(df[col]): df[col] = df[col].astype('category').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

Let's see the mechanism in action on a toy dataset:

df\_toy = pd.DataFrame(data={'Name':['Xue',np.nan,'Tom']}) df\_toy

|  | **Name** |
| --- | --- |
|  |  |
| **0** | Xue |
| **1** |  |
| **2** | Tom |

Converting the string column to a categorical variable means Pandas will replace each string with a unique integer representation, which we can include in the dataframe:

df\_string\_to\_cat(df\_toy) df\_toy['catcodes'] = df\_toy['Name'].cat.codes df\_toy

|  | **Name** | **catcodes** |
| --- | --- | --- |
|  |  |  |
| **0** | Xue | 1 |
| **1** |  | -1 |
| **2** | Tom | 0 |

Pandas still displays the Name column values as strings because that's more meaningful, but the Name column is now categorical:

print(df\_toy.dtypes)

Name category catcodes int8 dtype: object

To complete the label encoding, we call the second function to replace the category values with the integer codes:

df\_cat\_to\_catcode(df\_toy) df\_toy

|  | **Name** | **catcodes** |
| --- | --- | --- |
|  |  |  |
| **0** | 2 | 1 |
| **1** | 0 | -1 |
| **2** | 1 | 0 |

The Name column is one more than the catcodes column and so missing values become integer value 0 at the end of the encoding process.

To handle all missing nonnumeric values and label-encode nonnumeric columns, takes just two function calls:

df\_string\_to\_cat(df) df\_cat\_to\_catcode(df) df.head(2).T.head(10)

|  | **289125** | **289126** |
| --- | --- | --- |
|  |  |  |
| **SalePrice** | 8300 | 15500 |
| **ModelID** | 4663 | 11859 |
| **datasource** | 136 | 132 |
| **auctioneerID** | 31 | 25 |
| **YearMade** | 1985 | 1995 |
| **MachineHoursCurrentMeter** | 0.0000 |  |
| **UsageBand** | 0 | 0 |
| **saledate** | 2009-01-23 00:00:00 | 2009-01-23 00:00:00 |
| **fiModelDesc** | 654 | 3081 |
| **fiBaseModel** | 211 | 1127 |

We've now converted all string columns to numbers and dealt with missing string values.

**The unreasonable effectiveness of label encoding categorical variables**  
You might be wondering why it's “legal” to convert all of those unordered (nominal) categorical variables to ordered integers. We know for sure that assuming an order between categories is wrong. The short answer is that RF models can still partition such converted categorical features in a way that is predictive, possibly at the cost of a more complex tree model. This is definitely not true for many models, such as linear regression models (which require so-called “dummy” boolean columns, one for each unique categorical value). In practice, we've found label encoding categorical variables surprisingly effective, even when it seems more advanced methods would work better.

**7.5.2 Replacing missing numeric values**

To handle missing numeric values, we recommend a two step process:

1. For column *x*, create a new boolean column *x*\_na where *x*[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.

Those two steps have simple and direct equivalents in Python, thanks to Pandas:

def fix\_missing\_num(df, colname): df[colname+'\_na'] = pd.isnull(df[colname]) df[colname].fillna(df[colname].median(), inplace=True)

Let's make a toy dataframe with a numeric column that's missing a value:

df\_toy = pd.DataFrame(data={'YearMade':[1995,2001,np.nan]}) df\_toy

|  | **YearMade** |
| --- | --- |
|  |  |
| **0** | 1995.0000 |
| **1** | 2001.0000 |
| **2** |  |

and then run it through our function to see its effect on the dataframe:

fix\_missing\_num(df\_toy, 'YearMade') df\_toy

|  | **YearMade** | **YearMade\_na** |
| --- | --- | --- |
|  |  |  |
| **0** | 1995.0000 | False |
| **1** | 2001.0000 | False |
| **2** | 1998.0000 | True |

The missing value in the third row has been replaced by 1998, the median of 1995 and 2001, and there is a new column called YearMade\_na indicating we replaced a value.

The logic behind using the median is that we have to choose a number and so we might as well choose a number that's not going to skew the distribution of the data in that column. We also don't want to choose an extreme value that the model might latch onto as predictive. But, we should include a column in our dataset that indicates we've done this replacement because sometimes missing values are strongly predictive. For example, a bulldozer with an unknown manufacturing date is presumably less valuable because of the uncertainty. This approach is supported by recent academic research: [On the consistency of supervised learning with missing values](https://hal.archives-ouvertes.fr/hal-02024202v2).

Turning back to the full dataset now, we previously converted Tire\_Size from a string to a numeric column by parsing out the number of inches:

print(f"Values {df['Tire\_Size'].unique()}") print(f"Median {df['Tire\_Size'].median()}")

Values [ nan 26.5 20.5 17.5 23.5 14. 13. 29.5 15.5 7. 10. 23.1] Median 20.5

That still leaves a lot of missing values:

df[['Tire\_Size']].head(6)

|  | **Tire\_Size** |
| --- | --- |
|  |  |
| **289125** |  |
| **289126** |  |
| **289127** |  |
| **289128** |  |
| **289129** | 26.5000 |
| **289130** |  |

After applying fix\_missing\_num(), all np.nans representing missing values have been replaced with 20.5, the median of Tire\_Size:

fix\_missing\_num(df, 'Tire\_Size') df[['Tire\_Size']].head(6)

|  | **Tire\_Size** |
| --- | --- |
|  |  |
| **289125** | 20.5000 |
| **289126** | 20.5000 |
| **289127** | 20.5000 |
| **289128** | 20.5000 |
| **289129** | 26.5000 |
| **289130** | 20.5000 |

We also have to fix missing values in the other column we converted to numbers:

fix\_missing\_num(df, 'Undercarriage\_Pad\_Width')

Not all missing values are represented by np.nan. Sometimes people represent missing values by special indicator values during data entry or some conversion process. There are two numeric columns with such indicator values that we should fix because the feature importance graph suggests they are important.

One look at the relationship between YearMade and the bulldozer sale price shows that we have a problem:

» *Generated by code to left*

df\_small = df.sample(n=5\_000) # don't draw too many dots df\_small.plot.scatter('YearMade','SalePrice', alpha=0.02, c=bookcolors['blue'])

It's unlikely that humans were manufacturing bulldozers in the year 1000. Either the seller does not want to admit the age or does not know the age of the bulldozer. It's unclear why someone chose an indicator value of 1000 instead of 0 or -1, but we can fix this problem by replacing 1000 with np.nan. Then, we can apply our standard procedure for missing numeric values:

# There are some unlikely 1919, 1920 values too # Assume < 1950 is "unknown" df.loc[df.YearMade<1950, 'YearMade'] = np.nan fix\_missing\_num(df, 'YearMade')

Now the manufacturing year versus sale price looks a lot more reasonable.

There's one last problem with this column. Some records indicate that the bulldozer was sold before it was made, although there is only one in the last 100,000 records of our training subset:

inverted = df.query("saledate.dt.year < YearMade")[['SalePrice','YearMade','saledate']] inverted

|  | **SalePrice** | **YearMade** | **saledate** |
| --- | --- | --- | --- |
|  |  |  |  |
| **344948** | 35000 | 2012.0000 | 2010-05-06 |

That's easy enough to fix by setting the YearMade to the year of the sale date (using the assumption that the sale date is more recent and likely more accurate than the manufacturing date):

df.loc[df.eval("saledate.dt.year < YearMade"), 'YearMade'] = df['saledate'].dt.year

The other numeric column with a special value is MachineHoursCurrentMeter. At first glance, a bulldozer with 0 machine hours appears to be simply a new bulldozer. Let's filter for records with 0 missing hours and look at the histogram of YearMade:

» *Generated by code to left*

df.query("MachineHoursCurrentMeter==0")['YearMade'].plot.hist(bins=30)

Those manufacturing dates all precede 2009, which is the first sale year in our data subset. It's unlikely that all of those bulldozers sat idle from the time of their manufacture until their sale date years later. From this, we can conclude that 0 must indicate an unknown or “you really don't want to know” number of machine hours. Let's flip those zeros to np.nan and call fix\_missing\_num():

df.loc[df.eval("MachineHoursCurrentMeter==0"), 'MachineHoursCurrentMeter'] = np.nan fix\_missing\_num(df, 'MachineHoursCurrentMeter')

After handling these missing numeric values, there are three new columns on the end of the dataframe:

df[df.columns[-3:]].head(5)

|  | **Undercarriage\_Pad\_Width\_na** | **YearMade\_na** | **MachineHoursCurrentMeter\_na** |
| --- | --- | --- | --- |
|  |  |  |  |
| **289125** | True | False | True |
| **289126** | True | False | True |
| **289127** | True | False | True |
| **289128** | False | False | True |
| **289129** | True | False | True |

At this point all features are numeric, except for saledate, and missing values have been fixed.

**7.6 Training a model with all features**

Now that we have the dataframe prepped as pure numbers, we can use all the features to train a model and compare its performance to the baseline. The one exception is that saledate is still a time stamp, but we'll do something special with that in the next chapter. Here's the usual training sequence:

X, y = df.drop(['SalePrice','saledate'], axis=1), df['SalePrice'] rf, oob\_all = test(X, y)

OOB R^2 0.89871 using 5,151,648 tree nodes with 43.0 median tree height

That 0.899 is a big improvement upon our baseline score of 0.845, but we can do better through feature engineering, which is the subject of the next chapter. Let's take a snapshot of this cleaned up dataset to avoid repeating the same process:

df = df.reset\_index(drop=True) df.to\_feather("data/bulldozer-train-clean.feather")

To help focus our feature engineering efforts, let's check the feature importance graph (right gutter) to see what the model finds predictive. YearMade is still very important, but there's nothing left to do on that feature. Given their importance, we should take a close look at ProductSize, fiProductClassDesc, Enclosure, Hydraulics\_Flow, fiSecondaryDesc, and so on. Also notice the long tail of unimportant features. These features could be truly unimportant or could be extremely important, but for a small subset of the records. The best strategy is to leave all features in the model until the end, and then gradually remove them until accuracy drops.

**7.7 Summary**

In this chapter, we did a lot of cleanup work on the bulldozer data set, mostly related to converting column datatypes and dealing with missing numeric and string values. Part of the cleanup process was to identify physically-present numbers or strings that actually represent missing values. Our df\_normalize\_strings() function normalizes the notion of missing to np.nan for strings, but we had to identify indicators of missing values, such as medieval sale dates of 1000, manually.

The most important lesson of this chapter is how to deal with missing values. Missing categorical values are dealt with automatically because of our recommended label-encoding process: Convert categories to unique integer values; missing values, np.nan, become category code 0 and all other categories are codes 1 and above. 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*[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.

At this point, you've got some good data cleaning skills, you know how to normalize and encode string columns as numeric values, and you know how to deal with missing values. That means you know how to prepare datasets for model training purposes. In the next chapter, were going to beef up your feature engineering skills.

[*Book contents*](http://mlbook.explained.ai/) *Work in progress*  
Book version 0.4

**8 Bulldozer Feature Engineering**

[Terence Parr](http://parrt.cs.usfca.edu/) and [Jeremy Howard](http://www.fast.ai/about/#jeremy)

Copyright © 2018-2019 Terence Parr. All rights reserved.  
*Please don't replicate on web or redistribute in any way.*  
This book generated from markup+markdown+python+latex source with [Bookish](https://github.com/parrt/bookish).

You can make **comments or annotate** this page by going to the [annotated version of this page](https://via.hypothes.is/https:/mlbook.explained.ai/bulldozer-feateng.html). You'll see existing annotated bits highlighted in yellow. They are *PUBLICLY VISIBLE*. Or, you can send comments, suggestions, or fixes directly to [Terence](mailto:parrt@cs.usfca.edu).

**Contents**

* [Synthesizing date-related features](https://mlbook.explained.ai/bulldozer-feateng.html#sec:8.1)
* [ProductSize is an ordinal variable](https://mlbook.explained.ai/bulldozer-feateng.html#sec:8.2)
* [One-hot encoding Hydraulics\_Flow](https://mlbook.explained.ai/bulldozer-feateng.html#onehot-hf)
* [One-hot encoding Enclosure](https://mlbook.explained.ai/bulldozer-feateng.html#sec:8.4)
* [Splitting apart fiProductClassDesc](https://mlbook.explained.ai/bulldozer-feateng.html#sec:8.5)
* [Training with log(price)](https://mlbook.explained.ai/bulldozer-feateng.html#sec:8.6)
* [The effect of feature engineering on model performance](https://mlbook.explained.ai/bulldozer-feateng.html#sec:8.7)
* [Summary](https://mlbook.explained.ai/bulldozer-feateng.html#sec:8.8)

In the last chapter, we cleaned up the bulldozer dataset and fixed missing values. The resulting model's OOB  score was good, but we can improve on that score by transforming some existing columns and synthesizing others. The techniques in this chapter extend and improve upon the feature engineering we did in **Chapter 6** *Categorically Speaking*. The features to focus on are derived from the feature importance graph from the last chapter, and our engineering process will look like this:

1. Shatter the saledate feature into its constituent components
2. Convert ProductSize to an ordered categorical variable
3. *One hot encode* Hydraulics\_Flow and Enclosure
4. Split fiProductClassDesc into its constituent components
5. Take the logarithm of SalePrice target variable

As we go along, we'll examine the change in the model's OOB  score and see what pops up in the feature importance graph. At the end, we'll plot the feature transformations versus model scores to visualize the improvements gained from our handiwork.

1If you get an error “read\_feather() got an unexpected keyword argument 'nthreads',” then try:  
import feather  
feather.read\_dataframe("data/bulldozer-train.feather")

Let's get started by loading the cleaned up dataframe we computed in the last chapter. We'll also load the original dataset so that we apply different transformations to ProductSize, Hydraulics\_Flow, fiProductClassDesc, and Enclosure.1

df\_raw = pd.read\_feather("data/bulldozer-train.feather") df\_raw = df\_raw.iloc[-100\_000:] # same 100,000 records as before df = pd.read\_feather("data/bulldozer-train-clean.feather")

2Don't forget the [notebooks](https://mlbook.explained.ai/notebooks/) aggregating the code snippets from the various chapters.

We should get a new baseline OOB score, this time using a model with 150 trees in the forest, rather than the 50 we used previously. As we increase the number of trees, the accuracy of an RF tends to improve but only up to a certain point. More importantly for assessing model improvements, another interesting thing happens as we increase the number of trees. Scores from the larger and larger models start to converge to the same score for the same OOB test set, run after run. As the Random Forest averages more and more predictions from the individual trees, the variance of its combined predictions goes down and, hence, so does the variance of the overall OOB  scores. Here's how to get a stable baseline using the test() function used in previous chapters:2

X, y = df.drop(['SalePrice','saledate'], axis=1), df['SalePrice'] rf, oob\_clean = test(X, y, n\_estimators=150)

OOB R^2 0.90318 using 15,454,126 tree nodes with 43.0 median tree height

You'll notice that the OOB score of 0.903 slightly higher than the score we got at the end of the last chapter, due to the increased number of trees in the forest. Now let's dive into feature engineering.

**Bias and variance**  
Statisticians use the words *bias* and *variance* when evaluating the effect of model complexity on model accuracy. A model that systematically predicts values that differ from known true values is said to be *biased*. The larger that difference, the higher the bias. High bias models are not complex enough to capture the relationship between features and the target variable--they are *underfit*. If the model gives widely fluctuating results, depending on the test set, the model has high variance and is *overfit*. The simple way to think of these terms is that bias is accuracy and variance is generality.

The use of “variance” in this way is suboptimal, particularly when “overfit” is available and more explicit, because variance is used in so many contexts. For example, the term variance is also applicable when discussing the effect of increasing the number of trees in a Random Forest. Constructing an RF is an inherently random process and so multiple RF models trained on the exact same training set will be different. That means the prediction of these models on the exact same test set or OOB set will also be different. It's appropriate to describe the variation in the models' predictions as, well, variance.

Imagine that we build a model with, say, 10 trees and get an OOB  score. Repeat that train-and-score process many times, and you will find lots of variation in the scores. Now, increase the number of trees to 100 and repeat the trials. You will notice that the variation between model OOB scores is significantly lower than models trained with only 10 trees. (The [central limit theorem](https://en.wikipedia.org/wiki/Central_limit_theorem) comes into play when we average the predictions of multiple trees.)

So some people use variance to mean generality when comparing multiple test sets but also use variance to mean reduced prediction fluctuations on the same test set but larger forests. We recommend that you shy away from bias and variance, in favor of the more explicit accuracy and generality or underfit and overfit.

**8.1 Synthesizing date-related features**

Date columns in datasets are often predictive of target variables, such as the saledate in the bulldozer dataset. The date of sale and the year of manufacture together are strongly predictive of the sale price. As a general rule, we recommend shattering date columns into their constituent components to include: year, month, day, day of week (1..7), day of year (1..365), and even things like “end of quarter” and “end of month.” Pandas provides convenient functions to extract all of this information from a single datetime64 entity. After extracting the components, convert the datetime64 to an integer with the number of seconds since 1970 (the usual UNIX time measurement). Here's a basic function that illustrates how to synthesize date-related features:

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

After using the function, we can use Pandas' filter() to examine the newly-created columns:

df\_split\_dates(df, 'saledate') df.filter(regex=('sale\*')).head(2).T

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

Since we don't know which components, if any, will be predictive it's a good idea to just add whatever you can derive from the date. For example, you might want to add a column indicating that a day was a business holiday or even whether there was a big storm. Beyond the usual year/month/day and other numeric components, the new columns you synthesize will be application-specific. RF models won't get confused by the extra columns and we can excise useless features later, after finishing feature engineering. Let's check the effect of date-splitting on model accuracy:

X, y = df.drop('SalePrice', axis=1), df['SalePrice'] rf, oob\_dates = test(X, y, n\_estimators=150)

OOB R^2 0.91315 using 14,917,750 tree nodes with 43.0 median tree height

We get a nice bump from our clean baseline score of 0.903 to 0.913 and the number of nodes is smaller.

Now that we have a saleyear column in addition to the YearMade, let's create an age feature that explicitly states the age of a bulldozer for sale. The age of a vehicle is obviously important and, while the model has access to both fields already, it's a good idea to make life as easy as possible on the model:

df['age'] = df['saleyear'] - df['YearMade'] X, y = df.drop('SalePrice', axis=1), df['SalePrice'] rf, oob\_age = test(X, y, n\_estimators=150)

OOB R^2 0.91281 using 14,896,494 tree nodes with 43.0 median tree height

The OOB score is roughly the same after we add age, but the number of nodes is a little smaller.

Looking at the feature importance graph, none of the date-related features we added appear to be important, other than the converted saledate. YearMade is still very important, but age appears to be not that important.

» *Generated by code to left*

I = importances(rf, X, y) plot\_importances(I.head(15))

The reason for this is subtle but has to do with the fact that all of the date-related features are highly correlated, meaning that if we dropped one of them, the other features would “cover” for it. It's better to treat all of those date-related features as a meta-feature for feature importance graphs:

» *Generated by code to left*

features = list(df.drop('SalePrice',axis=1).columns) datefeatures = list(df.filter(regex=("sale\*")).columns) for f in datefeatures: features.remove(f) features.remove('YearMade') features.remove('age') features += [['YearMade','age']+datefeatures] I = importances(rf, X, y, features=features) plot\_importances(I.head(15))

While age might not individually pop up in the importance graph, a graph of age in years versus sale price confirms our belief that older vehicles sell for less on average. That correlation (relationship between the age feature and target price) implies at least some predictability for age.

» *Generated by code to left*

fig,ax = plt.subplots() df\_small = df.sample(n=5\_000) # don't draw too many dots ax.scatter(df\_small['age'], df\_small['SalePrice'], alpha=0.03, c=bookcolors['blue']) ax.set\_ylabel("SalePrice") ax.set\_xlabel("Age in years")

Let's keep all of these features for now and move on to the next task.

**8.2 ProductSize is an ordinal variable**

The ProductSize feature is important according to the feature importance graph so it's worth revisiting the feature to see if we can improve upon the default label encoding. To get corroborating evidence of its importance, we can also look at the relationship between product size and sale price using Pandas' groupby. By grouping the data by ProductSize then calling mean(), we get the average SalePrice (and other columns) across product sizes:

» *Generated by code to left*

temp = df\_raw.fillna('nan') # original dataset temp = temp.groupby('ProductSize').mean() temp[['SalePrice']].sort\_values('SalePrice').plot.barh()

3When copying a column from one dataframe to another, df\_raw to df, using the assignment operator, Pandas can silently do weird things depending on how the two dataframes are indexed. The safest approach is to copy over the NumPy version of the column by using .values, as we have done here.

(Here we're using the built-in Pandas shortcut to matplotlib's bar chart: .plot.barh().) There's a clear relationship between the size of the product and the sale price, as we would expect. Since Large is bigger than Small, the ProductSize feature is ordered, which means we can use an *ordinal encoding*. That just means that we assign numbers to the category values according to their size. A quick web search also shows that Mini and Compact bulldozers are the same size, leading to the following encoding:3

sizes = {None:0, 'Mini':1, 'Compact':1, 'Small':2, 'Medium':3, 'Large / Medium':4, 'Large':5} df['ProductSize'] = df\_raw['ProductSize'].map(sizes).values print(df['ProductSize'].unique())

[0 2 4 3 1 5]

By using an ordinal encoding rather than a label encoding, we get a small bump in OOB  score:

X, y = df.drop('SalePrice', axis=1), df['SalePrice'] rf, oob\_ProductSize = test(X, y, n\_estimators=150)

OOB R^2 0.91526 using 14,873,450 tree nodes with 45.0 median tree height

There are two other important features, Hydraulics\_Flow and Enclosure, that we can easily encode in a more structured way than label encoding.

**8.3 One-hot encoding Hydraulics\_Flow**

When in doubt, we encode categorical variables using label encoding. As we saw in the last section, however, if we notice that the variable is ordinal, we use that type of encoding. When the number of category levels is small, say, 10 or less we *one hot encode* the variable, assuming the category is important. One-hot encoding yields what people call *dummy variables*, boolean variables derived from a categorical variable where exactly one of the dummy variables is true for a given record. There is a new column for every categorical level. Missing category values yield 0 in each dummy variable.

**One-hot encoding**

The easiest way to pickup the idea behind one-hot encoding is through a trivial example. Imagine we have a categorical variable with three levels (three departments). We start out with a dataframe like this:

df\_toy = pd.DataFrame() df\_toy['Dept'] = ['Math','CS','Physics',np.nan] df\_toy

|  | **Dept** |
| --- | --- |
|  |  |
| **0** | Math |
| **1** | CS |
| **2** | Physics |
| **3** |  |

Pandas can give us the dummy variables for that column and concatenate them onto the existing dataframe:

onehot = pd.get\_dummies(df\_toy['Dept']) df\_toy = pd.concat([df\_toy, onehot], axis=1) df\_toy

|  | **Dept** | **CS** | **Math** | **Physics** |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| **0** | Math | 0 | 1 | 0 |
| **1** | CS | 1 | 0 | 0 |
| **2** | Physics | 0 | 0 | 1 |
| **3** |  | 0 | 0 | 0 |

Now, instead of a number, the “hot” position indicates the category. Notice how the missing value ends up with none hot. If you're wondering what “hot” refers to, “one hot” is an electrical engineering term referring to multiple chip outputs where at most one output has a nonzero voltage at any given time.

Feature Hydraulics\_Flow has “High Flow” and “Standard” levels, but the vast majority of records are missing a value for this feature:

print(df\_raw.Hydraulics\_Flow.value\_counts(dropna=False))

NaN 86819 Standard 12761 High Flow 402 None or Unspecified 18 Name: Hydraulics\_Flow, dtype: int64

Before one-hot encoding let's normalize the columns so string “None or Unspecified” is the same as missing (np.nan) then get dummy variables:

df['Hydraulics\_Flow'] = df\_raw['Hydraulics\_Flow'].values df['Hydraulics\_Flow'] = df['Hydraulics\_Flow'].replace('None or Unspecified', np.nan) onehot = pd.get\_dummies(df['Hydraulics\_Flow'], prefix='Hydraulics\_Flow', dtype=bool)

Next, we replace column Hydraulics\_Flow with the dummy variables by concatenating them onto the df DataFrame and deleting the unused column:

del df['Hydraulics\_Flow'] df = pd.concat([df, onehot], axis=1)

Checking the OOB , we see that it's about the same as before and the feature importance graph shows that the most predictive category is Standard. (See feature Hydraulics\_Flow\_Standard in the graph.)

X, y = df.drop('SalePrice', axis=1), df['SalePrice'] rf, oob\_Hydraulics\_Flow = test(X, y, n\_estimators=150)

OOB R^2 0.91558 using 14,872,994 tree nodes with 45.0 median tree height

**8.4 One-hot encoding Enclosure**

Let's follow this same one-hot procedure for Enclosure because it's also a categorical variable with only a few levels:

print(df\_raw.Enclosure.value\_counts(dropna=False))

OROPS 40904 EROPS w AC 34035 EROPS 24999 NaN 54 EROPS AC 6 NO ROPS 2 Name: Enclosure, dtype: int64

First, let's normalize the categories:

df['Enclosure'] = df\_raw['Enclosure'].values df['Enclosure'] = df['Enclosure'].replace('EROPS w AC', 'EROPS AC') df['Enclosure'] = df['Enclosure'].replace('None or Unspecified', np.nan) df['Enclosure'] = df['Enclosure'].replace('NO ROPS', np.nan)

Let's also look at the relationship between this variable and the sale price:

» *Generated by code to left*

temp = df.groupby('Enclosure').mean() temp[['SalePrice']].sort\_values('SalePrice').plot.barh()

That's interesting, “EROPS AC” gets, on average, twice the price of the other bulldozers. A web search reveals that ROP means “Roll Over Protection Structure,” so EROPS is an enclosed ROPS (a cabin) and OROP is an open protective cage. AC means “Air Conditioning.” The model suggests that a bulldozer with an enclosed cab gets a higher price and one with air conditioning gets the highest price, on average. Unfortunately, the OOB score drops a little bit but dummy variable Enclosure\_EROPS AC is important per the importance graph.

onehot = pd.get\_dummies(df['Enclosure'], prefix='Enclosure', dtype=bool) del df['Enclosure'] df = pd.concat([df, onehot], axis=1) X, y = df.drop('SalePrice', axis=1), df['SalePrice'] rf, oob\_Enclosure = test(X, y, n\_estimators=150)

OOB R^2 0.91356 using 14,896,328 tree nodes with 44.0 median tree height

» *Generated by code to left*

I = importances(rf, X, y) plot\_importances(I.head(20))

There are other important categorical variables, such as fiSecondaryDesc, but it has 148 levels, which would create 148 new columns, which would more than triple the number of overall columns in the dataframe. We recommend label encoding such categorical variables.

**8.5 Splitting apart fiProductClassDesc**

Feature fiProductClassSpec is string variable rather than a categorical variable. The values are descriptions of the product class and some components of the string appear to correlate with higher prices:

temp = df\_raw.groupby('fiProductClassDesc').mean() temp[['SalePrice']].sort\_values('SalePrice').head(15).plot.barh()

The bulldozers with higher operating capacity values seem to fetch higher prices. The model is clearly getting some kind of predictive power out of this feature when label encoded (per the feature important graph). But, we can make the information more explicit by splitting the description into four pieces:

The description is a categorical variable, chosen from a finite set of categories such as “Skip Steer Loader.” The lower and upper components are numerical features and the units is a category, such as “Horsepower” or “Lb Operating Capacity.” We can call the latter three components the “spec”. Because the spec is sometimes Unidentified, the spec components could be missing.

To pull apart the description string, let's do it in two steps. First, copy the original non-label-encoded string from df\_raw and split it at the hyphen:

# careful when copying between dataframes; use .values df\_split = df\_raw.fiProductClassDesc.str.split(' - ',expand=True).values df['fiProductClassDesc'] = df\_split[:,0] df['fiProductClassSpec'] = df\_split[:,1] # temporary column print(df['fiProductClassDesc'].unique())

['Track Type Tractor, Dozer' 'Hydraulic Excavator, Track' 'Wheel Loader' 'Skid Steer Loader' 'Backhoe Loader' 'Motorgrader']

This leaves the right-hand side of the string as the spec string:

print(df['fiProductClassSpec'].unique()[:5])

['20.0 to 75.0 Horsepower' '12.0 to 14.0 Metric Tons' '14.0 to 16.0 Metric Tons' '33.0 to 40.0 Metric Tons' '225.0 to 250.0 Horsepower']

Next, split that string using a regular expression that captures the two numbers and units to the right:

pattern = r'([0-9.\+]\*)(?: to ([0-9.\+]\*)|\+) ([a-zA-Z ]\*)' df\_split = df['fiProductClassSpec'].str.extract(pattern, expand=True).values df['fiProductClassSpec\_lower'] = pd.to\_numeric(df\_split[:,0]) df['fiProductClassSpec\_upper'] = pd.to\_numeric(df\_split[:,1]) df['fiProductClassSpec\_units'] = df\_split[:,2] del df['fiProductClassSpec'] # remove temporary column df.filter(regex=('fiProductClassSpec\*')).head(3)

|  | **fiProductClassSpec\_lower** | **fiProductClassSpec\_upper** | **fiProductClassSpec\_units** |
| --- | --- | --- | --- |
|  |  |  |  |
| **0** | 20.0000 | 75.0000 | Horsepower |
| **1** | 12.0000 | 14.0000 | Metric Tons |
| **2** | 14.0000 | 16.0000 | Metric Tons |

Because we have introduced columns with potentially missing values and new categorical variables, we have to prepare the dataset following our usual procedure:

fix\_missing\_num(df, 'fiProductClassSpec\_lower') fix\_missing\_num(df, 'fiProductClassSpec\_upper') # label encode fiProductClassDesc fiProductClassSpec\_units df\_string\_to\_cat(df) df\_cat\_to\_catcode(df)

We see a small bump in model performance from the transformation of this feature and a feature importance graph shows that the individual components we synthesized are important.

X, y = df.drop('SalePrice', axis=1), df['SalePrice'] rf, oob\_fiProductClassDesc = test(X, y, n\_estimators=150)

OOB R^2 0.91429 using 14,873,080 tree nodes with 43.0 median tree height

**8.6 Training with log(price)**

The original Kaggle competition measured model performance based upon the logarithm of the price, so we should also do that because we're going to compare our model's performance to the competition leaders in the next chapter. Also, as we discussed in **Section 5.5** *Log in, exp out*, it often helps to take the logarithm of the target variable when dealing with prices. (We usually care more that two prices are different by 20% than by a fixed $20.) Transforming the target variable is a simple matter of calling the log() function:

X, y = df.drop('SalePrice', axis=1), df['SalePrice'] y = np.log(y) rf, oob\_log = test(X, y, n\_estimators=150)

OOB R^2 0.91881 using 14,865,740 tree nodes with 44.0 median tree height

That 0.919 score is a nice bump in accuracy, all from a simple mathematical transformation.

**8.7 The effect of feature engineering on model performance**

We've done a lot of work in this chapter to improve the features presented to the RF model, so let's compare the effect of these changes on model performance. The following figure zooms in on the range of OOB scores from our baseline to the final log feature improvement. (Code for this figure is in the [notebook](https://mlbook.explained.ai/notebooks/bulldozer-feateng/eng.ipynb) for this chapter.)

Overall the model OOB score has improved from 0.903 to 0.919, a 16.141% improvement; (oob\_log-oob\_clean)\*100/(1-oob\_clean)).

For the most part, our feature engineering efforts have paid off. The one-hot encoding of Hydraulics\_Flow and Enclosure, however, doesn't seem to have improved model performance. In fact, Enclosure's one-hot encoding seems to have hurt performance. But, remember, we are measuring accuracy and looking at a feature importance graph using the training set, not a validation set. As it turns out, one-hot encoding Enclosure does seem to improve metrics where it counts, on validation and other test sets. (While working on the next chapter, we compared scores from models with and without one-hot encoded Enclosure columns.) For the moment, it's best to keep all features available to the model.

**8.8 Summary**

Let's summarize the techniques that we learned in this chapter.

**Dates**

As a general rule, break apart date columns into components such as day, month, year, day of week, day of year, and any other elements relevant to your application, such as “end of quarter” or “is holiday.” Synthesizing new columns based upon the date looks like this:

df["salemonth"] = df[colname].dt.month

Then convert the original date column to an integer representing the number of seconds since 1970 using:

df[colname] = df[colname].astype(np.int64) # convert date to seconds

**Ordinal encoding**

Categorical variables whose elements have order should be ordinal encoded using integers that mirror the relationship between category levels. For example, if a column has low, medium, and high levels, an encoding such as the following would work where missing values become 0:

m = {np.nan:0, 'low':1, 'medium':2, 'high':3} df[colname] = df\_raw[colname].map(m)

**One-hot encoding**

Non-ordered (nominal) categorical variables with about 10 or fewer levels can be one-hot encoded. Here is the basic procedure to replace a column with multiple dummy columns that one-hot encode the column:

onehot = pd.get\_dummies(df[colname]) df = pd.concat([df, onehot], axis=1) del df[colname]

**Split strings encoding**

In this chapter, we split a string based upon the hyphen character using split():

# get two columns df\_descr\_spec\_split = df[colname].str.split(' - ',expand=True)

and then used regular expressions to extract three components from the right-hand string using extract():

# get three columns pattern = r'([0-9.\+]\*)(?: to ([0-9.\+]\*)|\+) ([a-zA-Z ]\*)' df\_split = df['fiProductClassSpec'].str.extract(pattern, expand=True)

We created new columns in df from the columns extracted in this way.

There are lots of other kinds of strings (in other datasets) you might want to split apart, such as URLs. You can create new columns that indicate https vs http, the top level domain, domain, filename, file extension etc.

[*Book contents*](http://mlbook.explained.ai/) *Work in progress*  
Book version 0.4

**9 Train, Validate, Test**

[Terence Parr](http://parrt.cs.usfca.edu/) and [Jeremy Howard](http://www.fast.ai/about/#jeremy)

Copyright © 2018-2019 Terence Parr. All rights reserved.  
*Please don't replicate on web or redistribute in any way.*  
This book generated from markup+markdown+python+latex source with [Bookish](https://github.com/parrt/bookish).

You can make **comments or annotate** this page by going to the [annotated version of this page](https://via.hypothes.is/https:/mlbook.explained.ai/bulldozer-testing.html). You'll see existing annotated bits highlighted in yellow. They are *PUBLICLY VISIBLE*. Or, you can send comments, suggestions, or fixes directly to [Terence](mailto:parrt@cs.usfca.edu).

**Contents**

* [The testing trilogy](https://mlbook.explained.ai/bulldozer-testing.html#sec:trilogy)
  + [Splitting time-insensitive datasets](https://mlbook.explained.ai/bulldozer-testing.html#sec:split-time-insens)
  + [Splitting time-sensitive datasets](https://mlbook.explained.ai/bulldozer-testing.html#sec:9.1.2)
* [Rectifying training and validation sets](https://mlbook.explained.ai/bulldozer-testing.html#sec:rectifying)
  + [Preparing the training set](https://mlbook.explained.ai/bulldozer-testing.html#sec:9.2.1)
  + [Preparing consistent training and validation sets](https://mlbook.explained.ai/bulldozer-testing.html#sec:9.2.2)
  + [Getting baseline validation metrics](https://mlbook.explained.ai/bulldozer-testing.html#sec:9.2.3)
* [Tuning a Random Forest model](https://mlbook.explained.ai/bulldozer-testing.html#sec:tuning)
  + [Choosing a time-sensitive training set](https://mlbook.explained.ai/bulldozer-testing.html#sec:9.3.1)
  + [Choosing max\_features and min\_samples\_leaf](https://mlbook.explained.ai/bulldozer-testing.html#sec:9.3.2)
  + [Dropping irrelevant features](https://mlbook.explained.ai/bulldozer-testing.html#sec:9.3.3)
  + [Adjusting prices for inflation](https://mlbook.explained.ai/bulldozer-testing.html#sec:9.3.4)
* [Getting a true measure of generality](https://mlbook.explained.ai/bulldozer-testing.html#sec:test-set)
* [Summary](https://mlbook.explained.ai/bulldozer-testing.html#sec:9.5)

Being able to properly measure the accuracy of a model is a critical skill for a machine learning practitioner, and the goal of this chapter is to acquire that skill by applying it to our bulldozer dataset. Evaluating the accuracy of a model requires two key elements:

1. a metric that quantifies accuracy between predicted and true values
2. a set of observations outside of the training data whose predictions are tested by the metric

Models that perform poorly on training data will never generalize well to data they have never seen. Models that perform well on training data might or might not generalize well.

**Section 5.2** *Training and evaluating an initial model* introduced the  and MAE (mean absolute value) metrics, and we'll explore more metrics in **Chapter 11** *Measuring regression model performance*. In **Section 3.2.4** *Checking model generality*, we separated validation data from training data because we care about a model's performance on future observations, not how well it does on its own training data.

Keeping separate training and validation sets can be a hassle, though, as we'll see in this chapter. To avoid the hassle so far, we've been using the RF model's handy OOB (out-of-bag) samples as a substitute for a validation set. In general, accuracy metrics derived from the OOB samples are excellent estimates of the true validation scores but only for time-insensitive data.

2Linear regression models, on the other hand, do assume a linear relationship between features and price, which allows them to extrapolate.

Time-sensitive datasets, such as the bulldozer dataset, can look very different depending on the time period. Inflation alone means that future prices far beyond the training period will be much higher. An RF bulldozer price predictor trained on data from years 2000-2005 won't make accurate predictions for bulldozers sold in 2020. Metrics derived from OOB samples are, therefore, overly optimistic about the generality of a model and how it will perform on future predictions. Per **Section 3.4.3** *Comparing the digit classifier's performance to a linear model*, RF models make no assumption about the underlying relationship between features and target variable, which means that RFs cannot extrapolate beyond the range of their training experience.2 See also Rachel Thomas' [How (and why) to create a good validation set](https://www.fast.ai/2017/11/13/validation-sets/) article.

That means we must obtain a validation set beyond the date range of the training set in order to properly measure an RF's accuracy on time-sensitive data. We'll kick off this chapter by splitting off a validation set in **Section 9.1** *The testing trilogy*. Unfortunately, measuring accuracy with a separate validation set triggers a bit of unpleasantness, which we'll experience in **Section 9.2** *Rectifying training and validation sets*. In a nutshell, we have to make sure that categories in training and validation sets use the same encoding and that missing numeric values are filled in with medians computed only from the training set. Once we have a useful measure of accuracy via the validation set, we'll tune our model to improve its accuracy and generality in **Section 9.3** *Tuning a Random Forest model*.

The final step in our model development process is to evaluate the performance of the model on a test set, which we'll do in **Section 9.4** *Getting a true measure of generality*. The test set is outside of the training set, like the validation set, but must be hidden away and never run through intermediate models. The metric reported by this final test is the only objective estimate of a model's generality.

**9.1 The testing trilogy**

The only true measure of model generality comes from computing metrics on a test set that has never previously been run through the model.

Developing a machine learning model requires three sets of observations: training, validation, and test sets. The model trains just on the training set and model accuracy is evaluated using the validation set during development. After tuning the model on the validation set, we run the test set through the model to get our final measure of model accuracy and generality. If we peek at the test set and run it through an intermediate model rather than our final model, the test set becomes just another validation set. Every change made to a model after testing it on a dataset, tailors the model to that dataset; that dataset is no longer an objective measure of generality.

To develop a model in practice, we're usually given a single dataset, rather than separate training, validation, and test sets. That means we need a general procedure for splitting datasets appropriately.

**9.1.1 Splitting time-insensitive datasets**

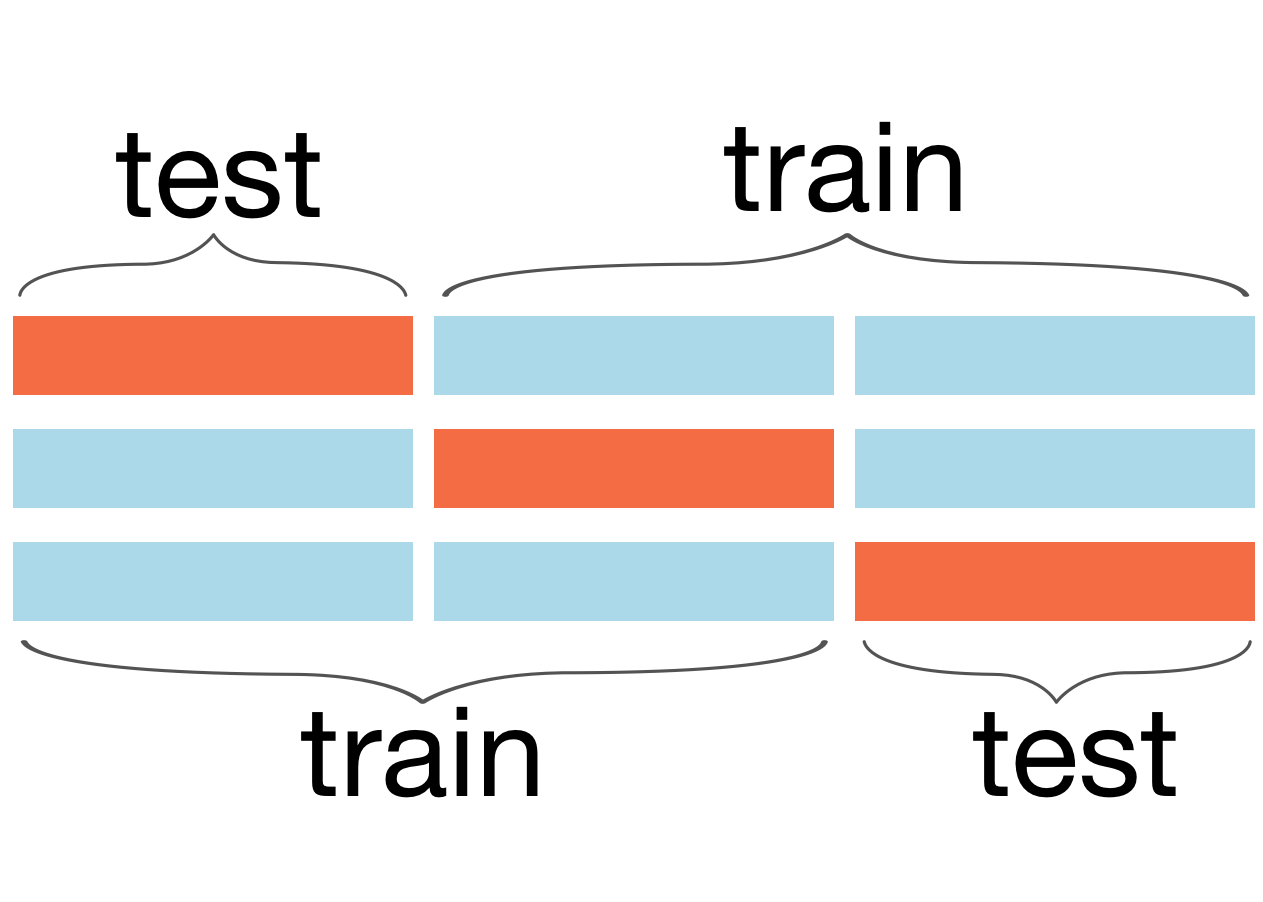
For datasets that do not change significantly over the time period of interest, we want to extract validation and test sets using random sampling of records. This is called the *holdout method*. To get (roughly) 70% of dataframe df into training and 15% into both validation and test sets, we can do this:

from sklearn.model\_selection import train\_test\_split df = df.sample(frac=1) # shuffle data df\_dev, df\_test = train\_test\_split(df, test\_size=0.15) df\_train, df\_valid = train\_test\_split(df\_dev, test\_size=0.15)

**Note**: Squirrel away the df\_test testing subset in a vault for a single use during final testing.

After training a model using df\_train, we'd run df\_valid data through the model and compute a metric, such as . Then we'd tune the model so that it's more accurate on df\_valid data. When we're happy with the model, we'd finally use df\_test to measure generality.

Because we're selecting validation and test sets randomly, it's possible that the sets will contain a disproportionate number of outlier records, such as really expensive bulldozers. Such tests are not representative and yield pessimistic accuracy metrics. Running the split-train-validate sequence in a loop, would extract different subsets each time and the resulting accuracy metrics would fluctuate. (Note that only the highlighted train\_test\_split() line would be part of the loop; never recompute df\_test.) A good strategy then would be to take the average accuracy metric over several runs.



**Figure 9.1**. Illustration of 3-fold cross validation training on two (blue) chunks, testing on the third (orange) until the model computes an accuracy metric for all three chunks

A slight variation on this procedure is called *k-fold cross validation* and splits the dataset into *k* chunks of equal size. We train the model on *k*-1 chunks and test it on the other, repeating the procedure *k* times so that we every chunk gets used as a validation set, as shown in **Figure 9.1**. The overall validation error is the average of the *k* validation errors. Here's how to use sklearn for 5-fold cross validation using an RF model:

from sklearn.model\_selection import cross\_val\_score rf = RandomForestRegressor(...) scores = cross\_val\_score(rf, X, y, cv=5) # k=5 print(scores.mean())

Cross validation and repeated subsampling are excellent techniques for measuring model accuracy, but are unsuitable for time-sensitive datasets.

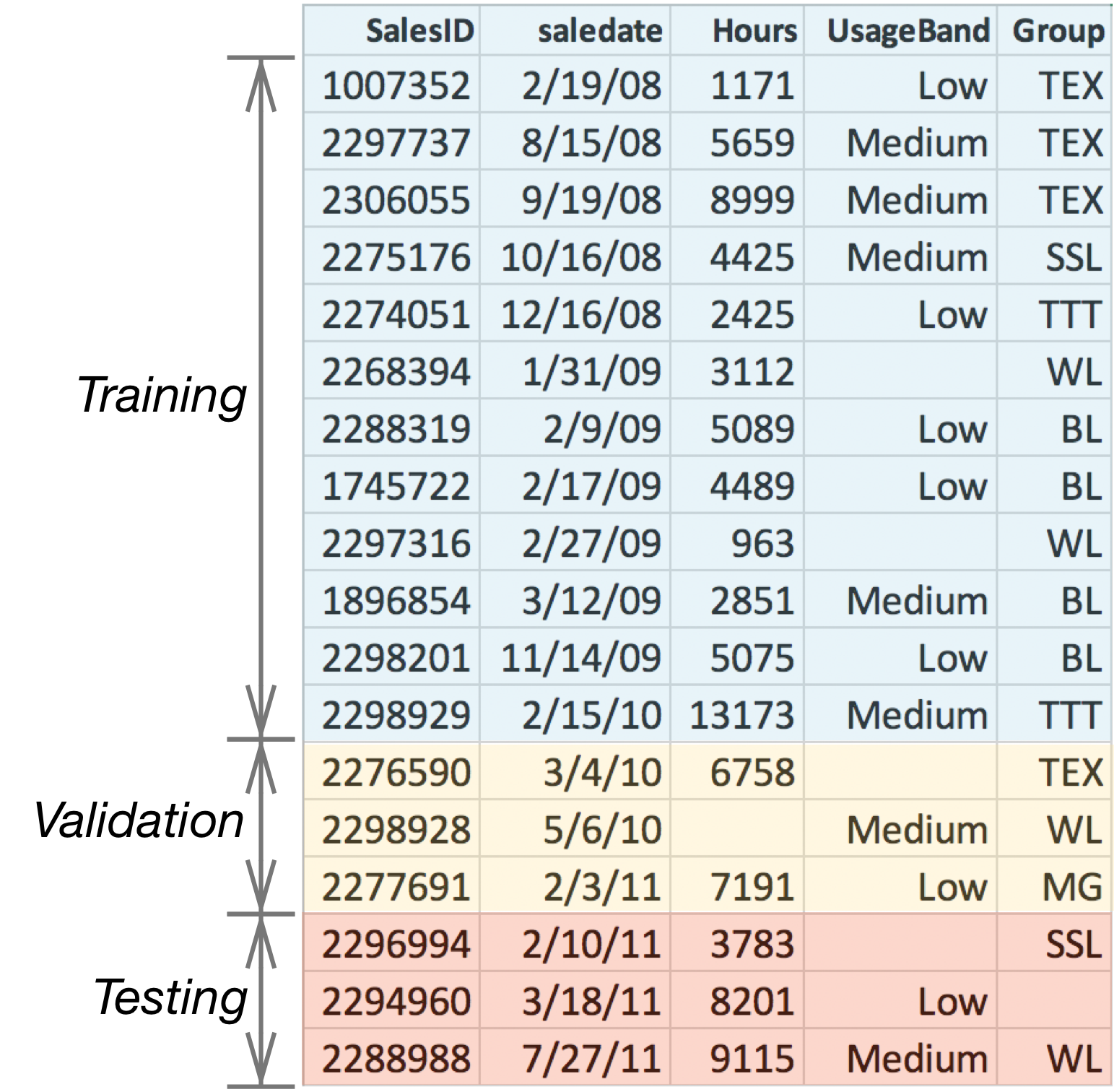
**9.1.2 Splitting time-sensitive datasets**

When observation features or target variables change meaningfully over time, random extraction of validation sets isn't appropriate. Randomly splitting a dataset would yield training and validation sets that overlap in time. That's a problem because it allows the model to train on data from the future and validation metrics would be overly optimistic. Imagine how your model would be used in practice. At some point, you must train a model on the data you have and then deploy it. Any observations subsequently submitted to the model for prediction are necessarily from dates beyond the end of the data used to train the model. Training should always mimic deployment and so our validation set should be from dates beyond the end of the training set.

The process for extracting training, validation, and test sets for time-sensitive data is:

1. Sort the records by date, earliest to latest
2. Extract the last, say, 15% of the records as df\_test
3. Extract the second to last 15% of the records as df\_valid
4. The remaining 70% of the original data is df\_train

For example, **Figure 9.2** illustrates the splitting process for a toy dataset derived from the bulldozer dataset where the records have been sorted from 2/19/08 to 07/27/11. Keep in mind that our final model will be tested on df\_test but trained on the combined data in training and validation sets.



**Figure 9.2**. Sorting and splitting time-sensitive data for testing

For the real bulldozer dataset, Kaggle provides a training set with 401,126 records and a validation set with 11,574 records. Because we need three datasets, let's use the provided validation set as our testing set and split off 12,000 records from the end of the training set as our validation set. What remains is our training set. Here are the nonoverlapping date ranges:

| **Subset** | **Start date** | **End date** | **Number of records** |
| --- | --- | --- | --- |
| Training | 1989-01-17 | 2011-08-19 | 389,126 |
| Validation | 2011-08-19 | 2011-12-30 | 12,000 |
| Testing | 2012-01-01 | 2012-04-28 | 11,574 |

If you look in prep-bulldozer.py in the data directory, you'll see code similar to the following that separates the original Kaggle training set into training and validation sets:

df = df.sort\_values('saledate') n\_valid = 12000 # same as Kaggle's test set size n\_train = len(df)-n\_valid df\_train = df[:n\_train].reset\_index(drop=True) df\_valid = df[n\_train:].reset\_index(drop=True)

**9.2 Rectifying training and validation sets**

In an ideal world, datasets would be purely numeric and without missing values. Feature engineering would still be useful, but numericalizing data such as encoding categorical variables, wouldn't be necessary. And we wouldn't have to conjure up missing values. Alas, real-world datasets are full of categorical variables and riddled with missing values, which introduces synchronization issues between training and validation/test sets. **Figure 9.2** illustrates a number of potential hazards:

1. If category TEX in column UsageBand is encoded as integer value 1 in the training set, the validation and test set must use the same encoding of 1.
2. Missing categorical values should be encoded as integer 0 in all sets.
3. Missing numeric values in column Hours should be filled with the median of just those values from Hours in the training set.
4. Categorical values in validation or test sets not present in the training set, such as MG in column Group, should be encoded as integer 0; the model has never seen MG, so we encode such values as if they were missing.

We can abstract that list into these important rules for preparing separated training and test sets:

1. Transformations must be applied to features consistently across data subsets.
2. Transformations of validation and test sets can only use data derived from the 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. In practice, that means tracking the median of all numeric columns, all category to category-to-code mappings, and which categories were one-hot encoded. 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. It sounds simple enough, but it's easy to screw up the synchronization between training and testing sets. Synchronization bugs usually show up as poor model accuracy, rather than as something obvious like a program exception.

**9.2.1 Preparing the training set**

Over the last two chapters, we've worked purely on the bulldozer training set to clean, encode, and perform feature engineering. Now, it's time to learn how to correctly prepare separate training and validation sets. We'll prepare the data as we did before, but will also track the data transformations that we perform. Then, we can apply those transformations to the validation set. We'll reuse as many functions as we can from the previous chapters, but some functions will require some updates and we'll organize some of the code snippets into new functions. You can find all the code from this chapter in the auto-generated [notebook](https://mlbook.explained.ai/notebooks/bulldozer-testing/prep.ipynb).

To get started, cut-and-paste the common set of import statements used in the last chapter into a new notebook. Then, copy in functions test(), fix\_missing\_num(), extract\_sizes(), df\_normalize\_strings(), df\_cat\_to\_catcode(), and df\_split\_dates(). Now, let's encapsulate all of the cleanup work that we did in **Chapter 7** *Exploring and Cleaning the Bulldozer Dataset* into a clean() function for use on validation and test sets later:

def clean(df): del df['MachineID'] # dataset has inconsistencies del df['SalesID'] # unique sales ID so not generalizer df['auctioneerID'] = df['auctioneerID'].astype(str) df\_normalize\_strings(df) extract\_sizes(df, 'Tire\_Size') extract\_sizes(df, 'Undercarriage\_Pad\_Width') df.loc[df['YearMade']<1950, 'YearMade'] = np.nan df.loc[df.eval("saledate.dt.year < YearMade"), 'YearMade'] = \ df['saledate'].dt.year df.loc[df.eval("MachineHoursCurrentMeter==0"), 'MachineHoursCurrentMeter'] = np.nan

In **Chapter 8** *Bulldozer Feature Engineering*, we ordinal encoded ProductSize:

def df\_order\_product\_size(df): sizes = {np.nan:0, 'mini':1, 'compact':1, 'small':2, 'medium':3, 'large / medium':4, 'large':5} df['ProductSize'] = df['ProductSize'].map(sizes).values

and one-hot encoded features Hydraulics\_Flow and Enclosure. Here's a generic function to one-hot encode categorical variables:

def onehot(df, colname): ascat = df[colname].astype('category').cat.as\_ordered() onehot = pd.get\_dummies(df[colname], prefix=colname, dtype=bool) del df[colname] df = pd.concat([df, onehot], axis=1) # return altered dataframe and column training categories return df, ascat.cat.categories

Feature fiProductClassDesc has lots of interesting information that we split into four new features:

def split\_fiProductClassDesc(df): df\_split = df.fiProductClassDesc.str.split(' - ',expand=True).values df['fiProductClassDesc'] = df\_split[:,0] df['fiProductClassSpec'] = df\_split[:,1] # temporary column pattern = r'([0-9.\+]\*)(?: to ([0-9.\+]\*)|\+) ([a-zA-Z ]\*)' spec = df['fiProductClassSpec'] df\_split = spec.str.extract(pattern, expand=True).values df['fiProductClassSpec\_lower'] = pd.to\_numeric(df\_split[:,0]) df['fiProductClassSpec\_upper'] = pd.to\_numeric(df\_split[:,1]) df['fiProductClassSpec\_units'] = df\_split[:,2] del df['fiProductClassSpec'] # remove temporary column

As we did with the cleanup procedure, let's encapsulate our feature engineering work into a function because we'll need to reuse this function on different amounts of training data later:

def feature\_eng(X): # for later use df\_split\_dates(X, 'saledate') df\_order\_product\_size(X) split\_fiProductClassDesc(X) X, hf\_cats = onehot(X, 'Hydraulics\_Flow') # normalize categories first then one-hot encode X['Enclosure'] = X['Enclosure'].replace('erops w ac', 'erops ac') X['Enclosure'] = X['Enclosure'].replace('no rops', np.nan) X, enc\_cats = onehot(X, 'Enclosure') catencoders = {'Hydraulics\_Flow':hf\_cats, 'Enclosure':enc\_cats} return X, catencoders

It's easy enough to remember to one-hot Hydraulics\_Flow and Enclosure later when working on the validation set, but we still need to track the training categories. If the categories in training and validation sets were identical, we wouldn't need to track anything; we could just apply onehot() to the validation set. If, however, the validation set contained a category not in the training set, we'd get a different number of columns in the validation set than in the training set. Before one-hot encoding, we have to line up the categories from training and validation. More on this shortly.

After cleanup and feature engineering comes the “numericalization” phase, which fixes any missing values and uses our default label encoding of categorical variables to remove any remaining non-numeric values. That means tracking the median of all numerical columns and recording the label encodings.

Rather than manually identifying numeric columns, we can ask Pandas via is\_numeric\_dtype(), which gives us a generic mechanism for replacing missing values in numeric columns and recording their medians:

def df\_fix\_missing\_nums(df:pd.DataFrame) -> dict: medians = {} # column name to median for colname in df.columns: if is\_numeric\_dtype(df[colname]): medians[colname] = df[colname].median(skipna=True) fix\_missing\_num(df, colname) return medians

You might be wondering why we're creating \_na columns even if a numeric training column has no missing values. The reason is that we have to be prepared to handle missing data in the validation or test sets, which would require \_na columns. A RF model won't be confused by a column full of False values, so it's safe to inject columns that end up being superfluous.

For our dataset, the return value of df\_fix\_missing\_nums() looks like:

{'ModelID': 4642.0, 'datasource': 136.0, 'YearMade': 2002.0, 'MachineHoursCurrentMeter': 3290.0, 'saledate': 1.2511584e+18, ...}

To apply label-encoding transformations consistently across training and validation sets, we have to update df\_string\_to\_cat() from the **Chapter 7** *Exploring and Cleaning the Bulldozer Dataset* to return a dictionary mapping a column name to the category index. The key functionality from Pandas is df[colname].cat.categories:

def df\_string\_to\_cat(df:pd.DataFrame) -> dict: catencoders = {} for colname in df.columns: if is\_string\_dtype(df[colname]) or is\_object\_dtype(df[colname]): df[colname] = df[colname].astype('category').cat.as\_ordered() catencoders[colname] = df[colname].cat.categories return catencoders

Here's an example of the categories stored in the dictionary at catencoders['Ripper']:

Index(['multi shank', 'single shank', 'yes'], dtype='object')

With those functions in place, we can encapsulate the numericalization phase with a wrapper function:

def numericalize(X, catencoders): medians = df\_fix\_missing\_nums(X) e = df\_string\_to\_cat(X) catencoders.update(e) df\_cat\_to\_catcode(X) return medians

At this point, we have functions that apply our cleanup, feature engineering, and numericalization procedures. Let's load our raw data and grab the last 100,000 records:

df = pd.read\_feather("data/bulldozer-train.feather") df = df.iloc[-100\_000:] # same 100,000 records as before X, y = df.drop('SalePrice', axis=1), df['SalePrice']

Then, our complete preparation procedure for the training records boils down to this simple sequence:

y = np.log(y) clean(X) X, catencoders = feature\_eng(X) medians = numericalize(X, catencoders)

Variables medians and catencoders track the information we need to consistently apply our transformations to the validation and test sets. We also need to remember that we one-hot encoded Hydraulics\_Flow and Enclosure.

Once we're sure all columns are numeric and that there are no missing values, we can train a model.

rf, r2\_train = test(X, y, n\_estimators=150)

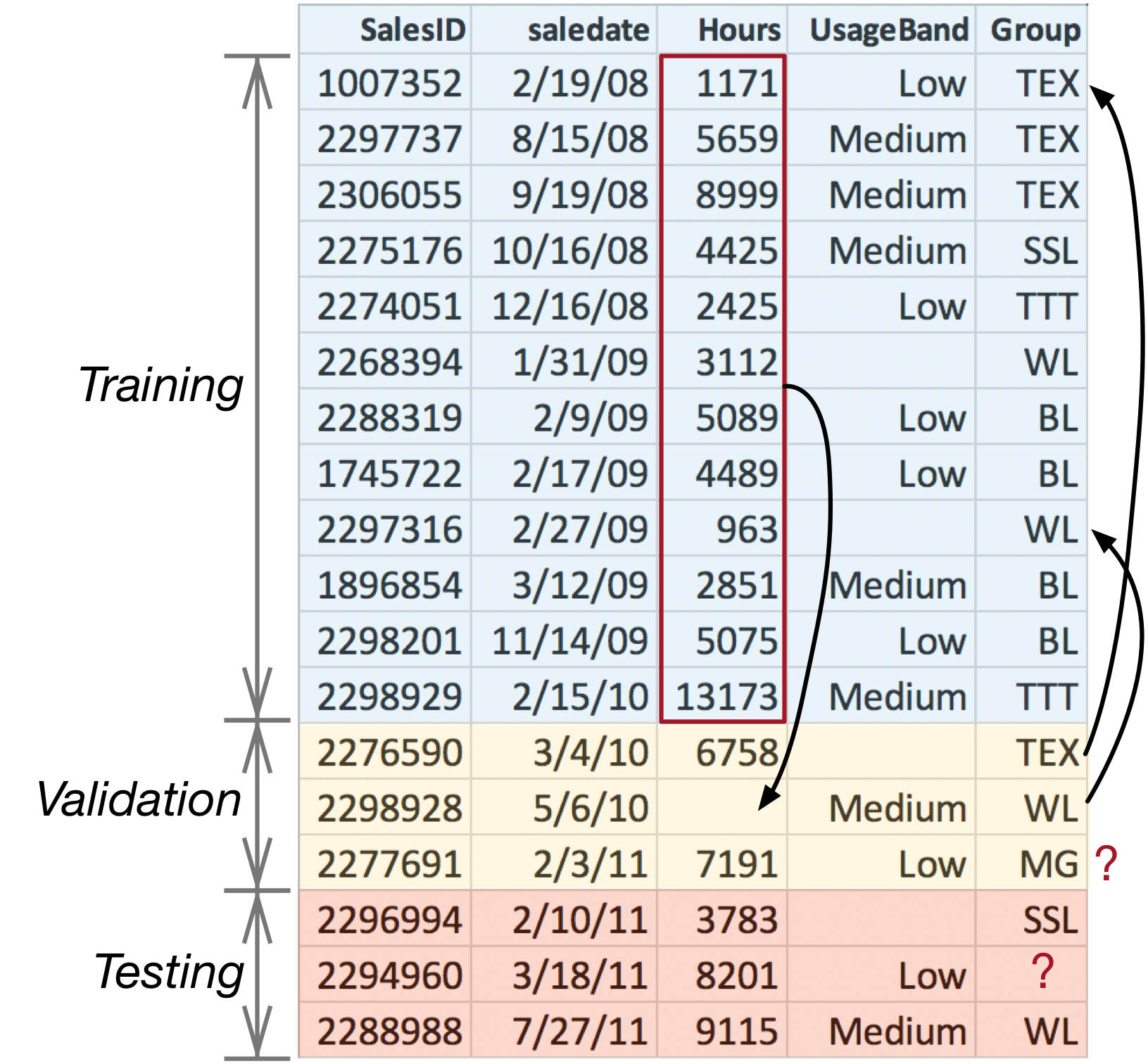
OOB R^2 0.91907 using 14,865,112 tree nodes with 45.0 median tree height

That  metric computed from OOB samples is about the same that we saw at the end of the last chapter, which is a good sanity check. Since the OOB samples are within the same date range as the training samples, the OOB  metric is overly optimistic. To get a better estimate of model generality, we really need a validation set.

**9.2.2 Preparing consistent training and validation sets**

To evaluate our model using a validation set, we need to load the raw validation data and then transform it as we did with the training set. The key difference is that we can only use data from the training set to label encode categories and fix missing values in the validation set, which we've saved in variables catencoders and medians. Models should never be trained on future data because that's a form of data leakage. Part of the answer would have leaked into the training data.

This is a very subtle point so it's worth emphasizing the potential hazards by annotating **Figure 9.2**, as shown in **Figure 9.3**.



**Figure 9.3**. Hazards in split training and test subset

The missing Hours value in the validation set should be replaced with the median of the Hours values from just the training set. When tested in production on a set of future records, we won't be retraining the model using those future records. Therefore, to get an accurate picture of future performance, no data from validation or test sets should ever be used in training the model.

And, we certainly shouldn't replace the missing value in the validation set with just the median of the validation data (6758 and 7191). One way to highlight the medians data leakage issue is to imagine testing a single future record with a missing value:



When there are literally no other Hours values with which to compute a median, our only choice is to compute the median from the training data alone. Here's a function to fill in missing values using medians from the training data:

def df\_fix\_missing\_test\_nums(df\_test, medians): for colname in medians: df\_test[colname+'\_na'] = pd.isnull(df\_test[colname]) df\_test[colname].fillna(medians[colname], inplace=True)

After missing numeric values, the next big hazard is inconsistent label encoding across training and validation sets. Consistency requires:

1. Each category must be encoded as the same numeric category code across all training and test sets.
2. Missing category values, in any set, must end up as integer code 0.
3. Categories found in test sets but not in the training set must be encoded as missing values, and ultimately as code 0.

The following function transforms all categorical variables according to those consistency rules, using the category indexes from the training set.

def df\_apply\_cats(df\_test:pd.DataFrame, catencoders:dict): for colname,encoder in catencoders.items(): # encode with categories from training set df\_test[colname] = \ pd.Categorical(df\_test[colname], categories=encoder, ordered=True)

After calling this function, missing category values in training and validation sets will be np.nan values, which will become zeros after we call df\_cat\_to\_catcode() during the final numericalization step.

Ensuring consistent one-hot encoding is another, related, hazard. The name and number of columns in the training and validation sets must be the same. This consistency is only a problem if there are categories in the validation set that are unknown to the training set. Such unknown categories should be encoded as missing values in the validation set, as we saw in the CS/Math/Physics example from **Section 8.3** *One-hot encoding Hydraulics\_Flow*. The model has no experience with that category, so we lump it together with missing values.

To ensure consistency, we just have to apply the training category index for column *x* to the validation set column *x* before one-hot encoding it:

def onehot\_apply\_cats(df\_test, colname, catencoders): df\_test[colname] = \ pd.Categorical(df\_test[colname], categories=catencoders[colname], ordered=True) onehot = pd.get\_dummies(df\_test[colname], prefix=colname, dtype=bool) del df\_test[colname] df\_test = pd.concat([df\_test, onehot], axis=1) del catencoders[colname] # simplify df\_apply\_cats() return df\_test

The reason that we delete the category index from catencoders is to simplify the loop in df\_apply\_cats(). That function does not have to test if colname in df\_test to avoid an indexing exception from df\_test[colname]. (The original categorical column *x* will not be in df\_test after one-hot encoding.)

To perform feature engineering on the validation set, we split dates, encode the product size, and split apart the product description as we did before. The only difference is how we apply the one-hot encoding:

def feature\_eng\_test(df\_test, catencoders): df\_split\_dates(df\_test, 'saledate') df\_order\_product\_size(df\_test) split\_fiProductClassDesc(df\_test) df\_test = onehot\_apply\_cats(df\_test, 'Hydraulics\_Flow', catencoders) df\_test['Enclosure'] = df\_test['Enclosure'].replace('erops w ac', 'erops ac') df\_test['Enclosure'] = df\_test['Enclosure'].replace('no rops', np.nan) df\_test = onehot\_apply\_cats(df\_test, 'Enclosure', catencoders) return df\_test

To numericalize the validation set, we apply categories with df\_apply\_cats() instead of calling df\_string\_to\_cat(), use medians from the training set to fix any missing values, and then do the usual category-to-code conversion:

def numericalize\_test(df\_test:pd.DataFrame, medians:dict, catencoders:dict): df\_apply\_cats(df\_test, catencoders) df\_fix\_missing\_test\_nums(df\_test, medians) df\_cat\_to\_catcode(df\_test)

**Program defensively**

Machine learning models are difficult to debug because bugs often only show up as fluctuations in model accuracy. It's a good idea to practice some defensive programming. Here's are two functions that do some basic sanity checking on the features before we shove them into the model:

def sanity\_check(df): for col in df.columns: if is\_string\_dtype(df[col]) or is\_object\_dtype(df[col]): print(f"Col {col} is still a string") if df[col].isnull().any(): print(f"Col {col} still has missing values") def check\_types(df1,df2): if df1.shape[1] != df2.shape[1]: print(f"Num columns differs: {df1.shape[1]} != {df2.shape[1]}") cols1 = set(df1.columns) cols2 = set(df2.columns) if cols1 != cols2: print(f"Column names differ:") if len(cols1-cols2)>0: print(f"\tIn df1 not df2: {cols1-cols2}") if len(cols2-cols1)>0: print(f"\tIn df2 not df1: {cols2-cols1}") for col in cols1.intersection(cols2): # check those in common if df1[col].dtype != df2[col].dtype: print(f"Col {col} dtypes differ {df1[col].dtype} != {df2[col].dtype}")

Here's a common invocation sequence:

sanity\_check(X) sanity\_check(X\_valid) check\_types(X, X\_valid)

With all of those functions in hand, we're finally ready to load and prepare a validation set with this simple code sequence:

df\_valid = pd.read\_feather("data/bulldozer-valid.feather") X\_valid, y\_valid = df\_valid.drop('SalePrice', axis=1), df\_valid['SalePrice'] y\_valid = np.log(y\_valid) clean(X\_valid) X\_valid = feature\_eng\_test(X\_valid, catencoders) numericalize\_test(X\_valid, medians, catencoders)

At this point, we've got prepared training data in X and y and prepared validation data in X\_valid and y\_valid; let's see how well the model does on the validation set.

**9.2.3 Getting baseline validation metrics**

For this bulldozer dataset, we've been measuring model performance with the unitless  score (in range negative infinity to 1.0), because that's what sklearn's RF implementation gives us easily for OOB samples. The MAE measure is easier to interpret, though, and so let's compute that as well for validation set predictions. To compare how well our model performs in comparison to the Kaggle competitors, we also need to compute the so-called *root mean squared log error* (*RMSLE*) (see the *Common regression metrics* box). Because we've already taken the log of y\_valid, squaring the difference between predicted and true values and then taking the square root gives us RMSLE. Good RMSLE errors are down near 0.23 for this data set.

**Common regression metrics**

There are a number of very common, and very similar, metrics used to evaluate the accuracy of regressors. We've already used mean absolute value, MAE, which is just the average absolute difference between predicted and true values:

def MAE(y\_pred, y\_true): return np.mean(np.abs(y\_pred - y\_true))

The absolute value prevents negative and positive deviations from canceling each other out. Instead of taking the absolute value, we could square the differences, giving us *mean squared error* (*MSE*). Squaring the difference also has the effect of emphasizing any predictions that are very far away from their true values:

def MSE(y\_pred, y\_true): return np.mean((y\_pred - y\_true)\*\*2)

To ignore a few significantly-deviant predictions, it's better to use MAE than MSE. It all depends on what you care about.

Because the units of MSE are the square of the target variable units, such as square dollars, practitioners often use *root mean squared error* (*RMSE*):

def RMSE(y\_pred, y\_true): return np.sqrt(MSE(y\_pred, y\_true))

If we take the logarithm of the target variable, as we've done with the bulldozer dataset (y=np.log(y)), then computing MSE is actually computing *mean squared log error* (*MSLE*). Similarly, if we take the square root of that, we get *root mean squared log error* (*RMSLE*).

Avoid a common pitfall by ensuring the columns of the validation set line up with the columns in the training set. Use Pandas' reindex() function before running a validation set through a model.

Model rf has been trained on X and y, but we can't immediately ask for predictions for the observations in X\_valid. The order of columns in the validation set could differ from the training set because the order in which we transform and inject new columns could be different. The sklearn models convert Pandas dataframes to numpy 2D arrays in fit() and predict() without concern for column order, so let's make sure they line up:

X\_valid = X\_valid.reindex(columns=X.columns)

Now, we can make predictions from X\_valid and compute , RMSLE, and MAE scores:

from sklearn.metrics import mean\_squared\_error, mean\_absolute\_error y\_pred = rf.predict(X\_valid) # Use np.exp(y\_valid) to get back into dollars space mae\_valid\_baseline = mean\_absolute\_error(np.exp(y\_valid), np.exp(y\_pred)) rmsle\_valid\_baseline = np.sqrt( mean\_squared\_error(y\_valid, y\_pred) ) r2\_valid\_baseline = rf.score(X\_valid, y\_valid) print(f"Validation R^2 {r2\_valid\_baseline:.5f}, "+ f"RMSLE {rmsle\_valid\_baseline:.5f}, "+ f"MAE ${mae\_valid\_baseline:.0f}")

Validation R^2 0.88148, RMSLE 0.24890, MAE $5939

The average bulldozer price is about $31,000 so being off by $5939 is not great, but we can improve on the scores a bit by tuning the model. As far as the Kaggle competition is concerned, that score would put us at the edge of the top 10% if our validation set were the same as the private leaderboard test set. (The competition is closed and so we can't test our model on the same test set; comparing to the leaderboard just gives us a ballpark performance measure.)

**9.3 Tuning a Random Forest model**

One of the nice characteristics of RF models is that they don't require a lot of tuning to get good accuracy. To get maximize accuracy, though, there are three common hyperparameters that we can tweak: the number of decision trees in the forest (n\_estimators), the number of randomly-selected features considered during training for each node in the trees (max\_features), and the minimum number of samples grouped into a leaf node (min\_samples\_leaf). We'll define these hyperparameters precisely in **Chapter 17** *Forests of Randomized Decision Trees*, but for now, let's just assume they are important.

Tuning a model involves repeatedly wiggling hyperparameters, retraining the model, and computing a metric based upon predictions from a validation set. The optimal hyperparameters are those that give us the best validation metric. During model development and experimentation related to feature engineering, we're more concerned with short training duration than finding optimal hyperparameters. It's only after we've more-or-less finished feature engineering that we tweak hyperparameters, looking for the best model. As a final tuning step, we'll try removing unimportant features.

To increase the speed with which we can try different models, we want to keep training time low, so we typically work with a subset of the training data and use between 20 and 50 trees in the RF model. (We've been using 100,000 out of about 400,000 records so far.) We sometimes set min\_samples\_leaf to the highest value that still gives us decent accuracy, because larger values of min\_samples\_leaf decrease tree size and smaller trees are faster to build. min\_samples\_leaf=1 is the default and finds the most detail in the training data, but results in the biggest trees.

When it's time to tune the model for accuracy, we gradually increase the number of trees until accuracy levels off. In the last chapter, we used 150 trees during feature engineering because we wanted to compare the effect of various features with a fairly stable metric. Due to the randomness of model construction, the same metric on the same data and same hyperparameters will fluctuate. The fewer the trees, the more the metric will fluctuate. For the remainder of this chapter, we'll bump the number of trees to 200, hoping to squeeze out a tiny bit more accuracy and increase metric stability. As we tune the model, we want to know that differences in accuracy are due to changes in hyperparameters, not random fluctuations.

Practitioners will often use a technique called a *grid search* to tune hyperparameters, which would try lots of combinations, looking for the best set of hyperparameters. Such a search takes forever to run and isn't necessary for RF models. We can choose hyperparameters in sequence. First, increase the number of trees until accuracy stops improving. Next, using the number of trees from the first step, try a few values of max\_features ('sqrt', 0.1 to 0.6) and pick the one that gives the best metric. Finally, using the best max\_features, run min\_samples\_leaf from 1 to about 15, again picking the best one.

**9.3.1 Choosing a time-sensitive training set**

Always choose your training and validation sets before attempting to tune the model. The idea is to start with the characteristics that affect accuracy the most and work your way to the more modest improvements obtained from tuning model hyperparameters.

Before we start the tuning process, let's choose a larger training set than the 100,000 records we've been using. In general, we want to use all training data available to us, but that's not always the case in time-sensitive datasets such as this. Clearly, the prices from 1989, at the start of the training set, would be lowball figures for the bulldozers sold in 2011. We have find a balance between using more training data and the biased prices of earlier observations. Through experimentation, we arrived at a training dataset with all samples from 2007 and later by simply iterating through multiple training set date ranges. Here's how to load just that training set:

df = pd.read\_feather("data/bulldozer-train.feather") df = df.query('saledate.dt.year>=2007').copy() X, y = df.drop('SalePrice', axis=1), df['SalePrice']

With new training data, we have to run the same data preparation process as before. That means also reprocessing the validation set because our medians and catencoders will have changed, since they are computed from the training set. Here's the complete sequence for preparing both training and validation sets:

y = np.log(y) clean(X) X, catencoders = feature\_eng(X) medians = numericalize(X, catencoders) df\_valid = pd.read\_feather("data/bulldozer-valid.feather") X\_valid, y\_valid = df\_valid.drop('SalePrice', axis=1), df\_valid['SalePrice'] y\_valid = np.log(y\_valid) clean(X\_valid) X\_valid = feature\_eng\_test(X\_valid, catencoders) df\_apply\_cats(X\_valid, catencoders) df\_fix\_missing\_test\_nums(X\_valid, medians) df\_cat\_to\_catcode(X\_valid)

Let's also create a method that trains a model, measures accuracy, and reports some metrics:

def test\_valid(X, y, X\_valid, y\_valid, n\_estimators=200, max\_features='auto', min\_samples\_leaf=1): X\_valid = X\_valid.reindex(columns=X.columns) rf = RandomForestRegressor(n\_estimators=n\_estimators, n\_jobs=-1, oob\_score=True, max\_features=max\_features, min\_samples\_leaf=min\_samples\_leaf) rf.fit(X, y) n = rfnnodes(rf) h = np.median(rfmaxdepths(rf)) y\_pred = rf.predict(X\_valid) mae\_valid = mean\_absolute\_error(np.exp(y\_valid), np.exp(y\_pred)) rmsle\_valid = np.sqrt( mean\_squared\_error(y\_valid, y\_pred) ) r2\_score\_valid = rf.score(X\_valid, y\_valid) print(f"OOB R^2 {rf.oob\_score\_:.5f} using {n:,d} tree nodes {h} median tree height") print(f"Validation R^2 {r2\_score\_valid:.5f}, RMSLE {rmsle\_valid:.5f}, MAE ${mae\_valid:.0f}") return rf, r2\_score\_valid, rmsle\_valid, mae\_valid

Here are the metrics for our model trained on the larger training set and tested on the updated validation set:

rf, r2\_score\_2007, rmsle\_2007, mae\_2007 = \ test\_valid(X, y, X\_valid, y\_valid)

OOB R^2 0.91934 using 34,346,352 tree nodes 45.0 median tree height Validation R^2 0.88339, RMSLE 0.24688, MAE $5967

Those are slightly better than the metrics we got from using just 100,000 training records. RMSLE has gone from 0.2489 to 0.2469. With this new baseline, let's tune the hyperparameters.

**9.3.2 Choosing max\_features and min\_samples\_leaf**

When constructing a decision tree node, an RF model randomly selects and examines a subset of the feature set. The size of that feature set affects a number of things, including accuracy and generality. The more features we allow the model to consider for each node, the higher the accuracy but the less general the model. (If we allowed the model to consider all features, the model would become overfit and much less general.) With a simple loop, we can try different values of max\_features, such as 0.1 to 0.6 stepping by 0.1 (that is, try 10% to 60% of the features):

ntrees = 200 minleaf = 1 for maxf in np.arange(.1,.6,.1): print(f"n\_estimators={ntrees}, max\_features={maxf:.1f}, min\_samples\_leaf={minleaf}") test\_valid(X, y, X\_valid, y\_valid, max\_features=maxf, min\_samples\_leaf=minleaf)

n\_estimators=200, max\_features=0.1, min\_samples\_leaf=1 OOB R^2 0.91818 using 38,555,212 tree nodes 45.0 median tree height Validation R^2 0.88949, RMSLE 0.24034, MAE $6036 n\_estimators=200, max\_features=0.2, min\_samples\_leaf=1 OOB R^2 0.92319 using 37,329,962 tree nodes 44.0 median tree height Validation R^2 0.89522, RMSLE 0.23402, MAE $5746 n\_estimators=200, max\_features=0.3, min\_samples\_leaf=1 OOB R^2 0.92398 using 36,429,076 tree nodes 44.0 median tree height Validation R^2 0.89574, RMSLE 0.23344, MAE $5681 n\_estimators=200, max\_features=0.4, min\_samples\_leaf=1 OOB R^2 0.92372 using 35,765,980 tree nodes 44.0 median tree height Validation R^2 0.89454, RMSLE 0.23479, MAE $5704 n\_estimators=200, max\_features=0.5, min\_samples\_leaf=1 OOB R^2 0.92345 using 35,358,180 tree nodes 45.0 median tree height Validation R^2 0.89359, RMSLE 0.23584, MAE $5732

It looks like max\_features=0.3 gives the lowest RMSLE error, so let's hold max\_features steady while we optimize min\_samples\_leaf, the next hyperparameter:

maxf = .3 for minleaf in range(2,7): print(f"n\_estimators={ntrees}, max\_features={maxf}, min\_samples\_leaf={minleaf}") test\_valid(X, y, X\_valid, y\_valid, max\_features=maxf, min\_samples\_leaf=minleaf)

n\_estimators=200, max\_features=0.3, min\_samples\_leaf=2 OOB R^2 0.92170 using 17,839,822 tree nodes 40.0 median tree height Validation R^2 0.89650, RMSLE 0.23259, MAE $5678 n\_estimators=200, max\_features=0.3, min\_samples\_leaf=3 OOB R^2 0.91926 using 11,489,890 tree nodes 38.0 median tree height Validation R^2 0.89635, RMSLE 0.23276, MAE $5696 n\_estimators=200, max\_features=0.3, min\_samples\_leaf=4 OOB R^2 0.91697 using 8,470,446 tree nodes 36.0 median tree height Validation R^2 0.89558, RMSLE 0.23362, MAE $5734 n\_estimators=200, max\_features=0.3, min\_samples\_leaf=5 OOB R^2 0.91485 using 6,710,466 tree nodes 35.0 median tree height Validation R^2 0.89463, RMSLE 0.23468, MAE $5762 n\_estimators=200, max\_features=0.3, min\_samples\_leaf=6 OOB R^2 0.91304 using 5,555,130 tree nodes 34.0 median tree height Validation R^2 0.89401, RMSLE 0.23537, MAE $5792

A value of 2 or 3 for min\_samples\_leaf yields the lowest error, depending on the run, so now we have decent values for both hyperparameters. Subsequent validation tests will like this:

rf, r2\_score\_valid, rmsle\_valid, mae\_valid = \ test\_valid(X, y, X\_valid, y\_valid, max\_features=.3, min\_samples\_leaf=2)

OOB R^2 0.92182 using 17,843,764 tree nodes 40.0 median tree height Validation R^2 0.89639, RMSLE 0.23272, MAE $5687

The tuned model has better scores than the untuned model. The RMSLE has gone from 0.2469 to 0.2327. That's a big bump if we're fighting tooth-and-nail in a Kaggle competition, but is not a huge win for practical model. The model is still off-line average of $5687.

**9.3.3 Dropping irrelevant features**

RF models are very good about ignoring features that are not predictive of the target variable, but they're not magic. We can often improve the overall performance of the model by dropping some of the unimportant features. Dropping features could come at the cost of reduced accuracy for a few records, but our goal, in this case, is to reduce the overall RMSLE error as used in the Kaggle competition. By selectively dropping the least important features, we'll arrive at a set of 33 features from the original 83 that gives us slightly better accuracy. (Training a model on this feature subset is also much faster as a bonus.)

Our strategy will be to repeatedly train a model on the current feature set, compute the feature importances, and drop the least important 10% from the feature set. Looking at the top 30 in the feature importance graph, we see a quick drop off in importance. We want to toss out the least helpful, but we can't do it in one pass by just taking all features above an importance threshold.

The permutation importance mechanism used in these graphs effectively shares importance between collinear features, those that are not independent of each other. Features YearMade and age are good examples of collinear features. If we dropped YearMade, then age would become much more important. Those two features have such large importance values we'd never drop them, but collinearity between less important features means we must take a conservative approach. After dropping the bottom 10% of the features, we'll recompute feature importance to recheck which features the model thinks are important before dropping the next 10%, and so on. The following function embodies that strategy.

def select\_features(X, y, X\_valid, y\_valid, drop=0.10): min\_rmsle = 99999 X\_valid = X\_valid.reindex(columns=X.columns) rf, \_, rmsle, \_ = test\_valid(X, y, X\_valid, y\_valid, max\_features=.3, min\_samples\_leaf=2) I = importances(rf, X\_valid, y\_valid) features = list(I.index) keep = best\_features = features n = int(.9/drop) # how many iterations? get to 90% for i in range(1,n+1): X2 = X[keep] X\_valid2 = X\_valid[keep] print(f"\nNum features = {len(keep)}") rf2, \_, rmsle, \_ = test\_valid(X2, y, X\_valid2, y\_valid, max\_features=.3, min\_samples\_leaf=2) if rmsle < min\_rmsle: min\_rmsle = rmsle best\_features = keep I2 = importances(rf2, X\_valid2, y\_valid) # recompute since collinear features = list(I2.index) keep = features[0:int(len(features)\*(1-drop))] return min\_rmsle, best\_features

We can use that function to select the best feature set:

min\_rmsle, best\_features = \ select\_features(X, y, X\_valid, y\_valid, drop=0.10) print(f"{len(best\_features)} features is best:") print(best\_features)

That takes a long time to run as it's retraining the model for each iteration, but the output looks like:

OOB R^2 0.92191 using 0 tree nodes 0 median tree height Validation R^2 0.89617, RMSLE 0.23296, MAE $5682 ... Num features = 53 OOB R^2 0.91312 using 0 tree nodes 0 median tree height Validation R^2 0.89148, RMSLE 0.23816, MAE $5996 ... Num features = 33 OOB R^2 0.90824 using 0 tree nodes 0 median tree height Validation R^2 0.89824, RMSLE 0.23062, MAE $5649 33 features is best: ['YearMade', 'ProductSize', 'fiProductClassSpec\_lower', ...]

Using the 33 features in best\_features, let's train a model and get some metrics:

X = X[best\_features] X\_valid = X\_valid[best\_features] rf, r2\_score\_bestf, rmsle\_bestf, mae\_bestf = \ test\_valid(X, y, X\_valid, y\_valid, max\_features=.3, min\_samples\_leaf=2)

OOB R^2 0.90841 using 14,362,342 tree nodes 39.0 median tree height Validation R^2 0.89852, RMSLE 0.23031, MAE $5642

By choosing the right feature subset, the RMSLE has dropped from 0.2327 to 0.2303. Using many fewer features also significantly reduces the size of the trees in the model.

**9.3.4 Adjusting prices for inflation**

Inflation causes prices to grow over time and the bulldozer dataset has roughly 20 years of data. Earlier records in the training set have lower sale prices on average than later records. Identical bulldozers sold in 1990 and 2010 would have significantly different prices, which would give conflicting training information to a model. An RF model predicts the average price computed from all identical, or nearly identical, bulldozers. Consequently, our model must be systematically underpredicting bulldozer prices in the validation set. In fact, we can verify that easily:

y\_valid\_pred = rf.predict(X\_valid) underprediction = np.mean(y\_valid-y\_valid\_pred) dollars = np.mean(np.exp(y\_valid)-np.exp(y\_valid\_pred)) print(f"Model underpredicts by ${dollars:.0f}, {underprediction:.5f} log(dollars)")

Model underpredicts by $2352, 0.03588 log(dollars)

Because of the increasing trend in prices, our model systematically underpredicts by a few thousand dollars, but the underprediction is only an average. In some cases, the model will overpredict. While properly dealing with time-sensitive datasets is beyond the intended scope of this book, we can still illustrate the problem briefly and then make a small tweak to our model to improve overall accuracy before moving on.

Time-sensitive datasets with prices are particularly challenging to predict because there's lots more going on than just inflation. The average bulldozer price over time does not simply grow steadily, as shown in **Figure 9.4**. (Code for this figure is in the [notebook](https://mlbook.explained.ai/notebooks/bulldozer-testing/trend.ipynb) for this chapter.) For example, the financial crises of 2000 and 2007 show fairly substantial drops in the average bulldozer price. Training a model on these prices too closely risks training the model to expect future crises, even though predicting the market is generally not possible. Instead, it's safer to assume prices grow, on average, along the orange trend line, which effectively smooths out these price crashes.

**Figure 9.4**. Average Bulldozer Price

Unfortunately, adjusting a model according to the overall price trend line only affects the overall accuracy. If we'd like to improve the accuracy for individual bulldozers or population segments, we need trend lines for each segment. For example, larger bulldozers should cost more than smaller bulldozers, which we can see in **Figure 9.5**. (Code for this figure is in the [notebook](https://mlbook.explained.ai/notebooks/bulldozer-testing/prod-trend.ipynb) for this chapter.) To get a more accurate model, we'd need to adjust prices differently, depending on ProductSize. There could be lots of features and combinations of features that dictate price fluctuations. To improve accuracy, we'd need to combine the efforts of multiple models that looked at different aspects of the training set. (See the [winners' discussion on Kaggle](https://www.kaggle.com/c/bluebook-for-bulldozers/discussion/4368).)

**Figure 9.5**. Average Bulldozer Price Per ProductSize

We're sticking to a single model in this book, but we can still tweak the output of this model by adding in the underprediction amount computed from the validation set. It will hurt some predictions and help others but, on average, the adjustment will improve the RMSLE:

y\_valid\_pred = rf.predict(X\_valid) + underprediction mae\_best = mean\_absolute\_error(np.exp(y\_valid), np.exp(y\_valid\_pred)) rmsle\_best = np.sqrt( mean\_squared\_error(y\_valid, y\_valid\_pred) ) r2\_score\_best = r2\_score(y\_valid, y\_valid\_pred) print(f"Adjusted-model validation R^2 {r2\_score\_best:.5f}, RMSLE {rmsle\_best:.5f}, MAE {mae\_best:.0f}")

Adjusted-model validation R^2 0.90098, RMSLE 0.22750, MAE 5464

The highlighted code line has our simple tweak to the model predictions. Our RMSLE error drops from 0.2303 to 0.2275. We can think of underprediction as another parameter of a new meta-model or we can think of underprediction has a second (additive) model that takes the output of a previous model and generates new output. Either way, we're updating our model based upon information derived from the validation set results, just like we did with hyper parameters, which means that we've tailored our model somewhat to this validation set. Nonetheless, the tweak is still useful for any other test sets we have, as long as they are in the near future.

{TODO: What about adding "DaysSinceFirstSale" feature to detrend in model?}

**9.4 Getting a true measure of generality**

At this point, we have a final model that is based upon a selected training subset, a subset of features, tuned hyper parameters, and a tweak to adjust for inflation. Our best RMSLE on the validation set is 0.2275, but because we've turned the model hyper parameters and computed an inflation adjustment on the validation set, metrics derived from the validation set are overly optimistic. That's why we carved out a test set that we've totally ignored until now. We can finally use that test set to get an objective measure of model generality.

First, let's load and prepare a training set that includes the validation set in order to use as much training data as possible. (File bulldozer-train-all.feather is created by prep-bulldozer.py in the data directory.)

df = pd.read\_feather("data/bulldozer-train-all.feather") df = df.query('saledate.dt.year>=2007').copy() X, y = df.drop('SalePrice', axis=1), df['SalePrice'] y = np.log(y) clean(X) X, catencoders = feature\_eng(X) medians = numericalize(X, catencoders) X = X[best\_features]

Next, load and prepare the test set, just like we did for the validation set:

df\_test = pd.read\_feather("data/bulldozer-test.feather") X\_test, y\_test = df\_test.drop('SalePrice', axis=1), df\_test['SalePrice'] y\_test = np.log(y\_test) clean(X\_test) X\_test = feature\_eng\_test(X\_test, catencoders) df\_apply\_cats(X\_test, catencoders) df\_fix\_missing\_test\_nums(X\_test, medians) df\_cat\_to\_catcode(X\_test) X\_test = X\_test[best\_features]

Then train a model and compute metrics using the test set:

rf, r2\_score\_test, rmsle\_test, mae\_test = \ test\_valid(X, y + underprediction, X\_test, y\_test, max\_features=.3, min\_samples\_leaf=2)

OOB R^2 0.90900 using 15,344,480 tree nodes 39.0 median tree height Validation R^2 0.89356, RMSLE 0.23961, MAE $6018

So, how good is our model? Well, that depends on what we care about. One of the most important characteristics of a model is how well it generalizes from the training data to unseen future data. From this perspective, our model is very good because the test RMSLE error of 0.2396 is very similar to the validation RMSLE, 0.2275. The model isn't over fit to the training data and, therefore, doesn't fall apart when we move from the validation to the test set.

If we care about the model's performance in comparison to Kaggle competitors, our model's accuracy on a test set is also excellent. The Kaggle [private leaderboard](https://www.kaggle.com/c/bluebook-for-bulldozers/leaderboard) lists the performance of competitors' models on an unseen test set, just as we've done for our test set. But, the Kaggle competition is now closed, so we can't submit our model for evaluation using the actual final test set used in the competition. While the two test sets aren't identical, comparing the scores gives us a rough estimate of how well our model would perform in the competition. Our model's score of 0.2396 would put us at about 20th position, somewhere in the top 5%. (Out of 475 competitors, the highest score, lowest RMSLE, was 0.22909.)

If you compare the private and public leaderboards, you'll see that 108 out of 475 competitors were able to get an RMSLE of 0.0 on the provided validation set. In other words, if you create an ensemble of enough models and work really hard, you can tailor your model to perfectly predict a known validation set. Those models are almost certainly dramatically overfit, which is why most of them performed very poorly on the unseen test set, whose results are shown on the private leaderboard. 108 models got perfect scores on the visible validation set, but position 108 for the unseen test set is below a benchmark model provided by Kaggle. Nothing worse than the benchmark score counts for the contest. The point is that it's possible to find a particular combination of models and features that nails a specific, known data set, but that doesn't mean the model is useful. Models are only potentially useful if they generalize well to unseen data sets.

Instead of asking how our model compares to other models, a better question asks whether or not our model is useful. Let's take a look at the various scores we got after various training and testing stages:

|  | **OOB R^2** | **RMSLE** | **MAE** |
| --- | --- | --- | --- |
| **Stage** |  |  |  |
| **Training 100k records** | 0.8815 | 0.2489 | 5939 |
| **Training set >= 2007** | 0.8834 | 0.2469 | 5967 |
| **After tuning** | 0.8964 | 0.2327 | 5687 |
| **Best feature subset** | 0.8985 | 0.2303 | 5642 |
| **Inflation adjusted** | 0.9010 | 0.2275 | 5464 |
| **Test set** | 0.8936 | 0.2396 | 6018 |

After all of that hard work on the model using the validation set, the average bulldozer price prediction is still off by $5464. We started at $5939, but that might not be a meaningful improvement for a practical system. Price predictions are off by more than 15% on the validation set and about 20% on the test set. On the other hand, our model would still be useful if it were more accurate than a human.

**9.5 Summary**

We just finished a three-chapter sequence on the bulldozer data set. We've learned some new data cleaning and string normalization techniques, but more importantly, we learned how to deal with missing values. During feature engineering, we split apart dates and encoded categorical variables using ordinal encodings and one-hot encodings. In this chapter, we saw how to create validation and test sets for time-sensitive data and studied how to keep them consistent with the training data. Let's summarize the techniques from this chapter.

To put machine learning into practice, we need a training set, a validation set, and a test set. The test set is extracted first and held out to be used once by our final model as an objective measure of generality. We train the model on the training set and test it on the validation set. Model tuning occurs by observing the effects of changing hyperparameters on validation set metrics. We never tune the model on the test set and never run the test set through an intermediate model. If the data is time-sensitive, we can't extract random subsets for validation and testing. We sort by date then split off the last 15% as the test set; the second to last 15% is the validation set.

We have to prepare both the training and validation sets in the same way, but there are a number of consistency hazards. Each transformation to the training set must be tracked for application to the validation set. For example, in this chapter, we tracked the medians of all numerical columns, the categorical variables we one-hot encoded, and the category indexes for all categorical columns. To prevent one-hot encoding from introducing extra columns, apply the categorical index from the training set so that previously-unseen categories become missing values. Before making predictions using our validation set, make sure that order of the columns is the same in both data sets. Keep in mind the general rule that transformations done to the validation set can only use data computed from the training set, such as medians for filling in missing values.

Tuning your model can include choosing the right training set, particularly for time-sensitive data. Sometimes the most recent data provides a more accurate picture of the future. There are three key RF hyperparameters to tune: the number of trees, the number of features considered per decision node, and the minimum size of decision tree leaves. Using a grid search that tries lots of combinations, we've found it effective to try them one-by-one, and in that order. After tuning the hyperparameters, follow the procedure to drop irrelevant features: train a model on the current feature set, compute the feature importances, drop the least important 10% from the feature set, repeat.

As a final step, bring out your test set and run it through the model. The score you get, good or bad, is your objective measure of generality. If the test metrics are much worse than the validation metrics, then your model is overfit to the validation data. Remember that every time you try a new model or a combination of models, it's one more chance to find a model that just happens to work well on your validation set.