Data Cleaning and Preparation

Part 3

Data Transformation

Part 2

Discretization and Binning

- Continuous data is often discretized or otherwise separated into “bins” for analysis.
- Suppose you have data about a group of people in a study, and you want to group them into discrete age buckets:

```
In [73]: ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
```

- Let’s divide these into bins of 18 to 25, 26 to 35, 36 to 60, and finally 61 and older.
- To do so, you have to use `cut`, a function in pandas:

```
In [74]: bins = [18, 25, 35, 60, 100]

In [75]: cats = pd.cut(ages, bins)

In [76]: cats

Out[76]: [(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], ..., (25, 35], (60, 100], (35, 60], (35, 60], (25, 35]]
         Length: 12
         Categories (4, interval[int64]): [(18, 25] < (25, 35] < (35, 60] < (60, 100]]
```

- The object pandas returns is a special `Categorical` object.
- The output you see describes the bins computed by `pandas.cut`.
- You can treat it like an array of strings indicating the bin name; internally it contains a categories array specifying the distinct `category` names along with a labeling for the `ages` data in the `codes` attribute:

```
In [77]: cats.codes

Out[77]: array([0, 0, 0, 1, 0, 0, 2, 1, 3, 2, 2, 1], dtype=int8)

In [78]: cats.categories

Out[78]: IntervalIndex([(18, 25], (25, 35], (35, 60], (60, 100]],
                       closed='right',
                       dtype='interval[int64]')

In [79]: pd.value_counts(cats)

Out[79]: (18, 25]     5
         (35, 60]     3
         (25, 35]     3
         (60, 100]    1
         dtype: int64
```

- Consistent with mathematical notation for intervals, a parenthesis means that the side is *open*, while the square bracket means it is *closed* (inclusive).
- You can change which side is closed by passing `right=False`:

```
In [80]: pd.cut(ages, [18, 26, 36, 61, 100], right=False)

Out[80]: [[18, 26), [18, 26), [18, 26), [26, 36), [18, 26), ..., [26, 36), [61, 100), [36, 61), [36, 61), [26, 36)]
         Length: 12
         Categories (4, interval[int64]): [[18, 26) < [26, 36) < [36, 61) < [61, 100)]
```

- You can also pass your own bin names by passing a list or array to the `labels` option:

```
In [81]: group_names = ['Youth', 'YoungAdult', 'MiddleAged', 'Senior']

In [82]: pd.cut(ages, bins, labels=group_names)

Out[82]: [Youth, Youth, Youth, YoungAdult, Youth, ..., YoungAdult, Senior, MiddleAged, MiddleAged, YoungAdult]
         Length: 12
         Categories (4, object): [Youth < YoungAdult < MiddleAged < Senior]
```

- If you pass an integer number of bins to `cut` instead of explicit bin edges, it will compute equal-length bins based on the minimum and maximum values in the data.
- Consider the case of some uniformly distributed data chopped into fourths:

```
In [84]: pd.cut(data, 4, precision=2)

Out[84]: [(0.34, 0.55], (0.34, 0.55], (0.76, 0.97], (0.76, 0.97], (0.34, 0.55], ..., (0.34, 0.55], (0.34, 0.55], (0.55, 0.76], (0
         .34, 0.55], (0.12, 0.34]]
         Length: 20
         Categories (4, interval[float64]): [(0.12, 0.34] < (0.34, 0.55] < (0.55, 0.76] < (0.76, 0.97]]
```

- A closely related function, `qcut`, bins the data based on sample quantiles.
- Depending on the distribution of the data, using `cut` will not usually result in each bin having the same number of data points.
- Since `qcut` uses sample quantiles instead, by definition you will obtain roughly equal-size bins:

```
In [85]: data = np.random.randn(1000)  # Normally distributed

In [86]: cats = pd.qcut(data, 4)  # Cut into quartiles

In [87]: cats

Out[87]: [(-0.0265, 0.62], (0.62, 3.928], (-0.68, -0.0265], (0.62, 3.928], (-0.0265, 0.62], ..., (-0.68, -0.0265], (-0.68, -0.026
         5], (-2.9499999999999997, -0.68], (0.62, 3.928], (-0.68, -0.0265]]
         Length: 1000
         Categories (4, interval[float64]): [(-2.9499999999999997, -0.68] < (-0.68, -0.0265] < (-0.0265, 0.62] < (0.62, 3.928]]

In [88]: pd.value_counts(cats)

Out[88]: (0.62, 3.928]                   250
         (-0.0265, 0.62]                 250
         (-0.68, -0.0265]                250
         (-2.9499999999999997, -0.68]    250
         dtype: int64
```

- Similar to `cut` you can pass your own quantiles (numbers between 0 and 1, inclusive):

```
In [89]: pd.qcut(data, [0, 0.1, 0.5, 0.9, 1.])
Out[89]: [(-0.0265, 1.286], (-0.0265, 1.286], (-1.187, -0.0265], (-0.0265, 1.286], (-0.0265, 1.286], ..., (-1.187, -0.0265], (-1.
187, -0.0265], (-2.9499999999999997, -1.187], (-0.0265, 1.286], (-1.187, -0.0265]]
Length: 1000
Categories (4, interval[float64]): [(-2.9499999999999997, -1.187] < (-1.187, -0.0265] < (-0.0265, 1.286] < (1.286, 3.928
]]
```

Detecting and Filtering Outliers

- Filtering or transforming outliers is largely a matter of applying array operations.
- Consider a DataFrame with some normally distributed data:

```
In [90]: data = pd.DataFrame(np.random.randn(1000, 4))

In [91]: data.describe()
Out[91]:
```

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.049091	0.026112	-0.002544	-0.051827
std	0.996947	1.007458	0.995232	0.998311
min	-3.645860	-3.184377	-3.745356	-3.428254
25%	-0.599807	-0.612162	-0.687373	-0.747478
50%	0.047101	-0.013609	-0.022158	-0.088274
75%	0.756646	0.695298	0.699046	0.623331
max	2.653656	3.525865	2.735527	3.366626

- Suppose you wanted to find values in one of the columns exceeding 3 in absolute value:

```
In [92]: col = data[2]

In [93]: col[np.abs(col) > 3]
Out[93]: 41    -3.399312
         136   -3.745356
         Name: 2, dtype: float64
```

- To select all rows having a value exceeding 3 or -3, you can use the `any` method on a boolean DataFrame:

```
In [94]: data[(np.abs(data) > 3).any(1)]
Out[94]:
```

	0	1	2	3
41	0.457246	-0.025907	-3.399312	-0.974657
60	1.951312	3.260383	0.963301	1.201206
136	0.508391	-0.196713	-3.745356	-1.520113
235	-0.242459	-3.056990	1.918403	-0.578828
258	0.682841	0.326045	0.425384	-3.428254
322	1.179227	-3.184377	1.369891	-1.074833
544	-3.548824	1.553205	-2.186301	1.277104
635	-0.578093	0.193299	1.397822	3.366626
782	-0.207434	3.525865	0.283070	0.544635
803	-3.645860	0.255475	-0.549574	-1.907459

- Values can be set based on these criteria.
- Here is code to cap values outside the interval -3 to 3:

```
In [95]: data[np.abs(data) > 3] = np.sign(data) * 3

In [96]: data.describe()
Out[96]:
```

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.050286	0.025567	-0.001399	-0.051765
std	0.992920	1.004214	0.991414	0.995761
min	-3.000000	-3.000000	-3.000000	-3.000000
25%	-0.599807	-0.612162	-0.687373	-0.747478
50%	0.047101	-0.013609	-0.022158	-0.088274
75%	0.756646	0.695298	0.699046	0.623331
max	2.653656	3.000000	2.735527	3.000000

Permutation and Random Sampling

- Permuting (randomly reordering) a Series or the rows in a DataFrame is easy to do using the `numpy.random.permutation` function.
- Calling permutation with the length of the axis you want to permute produces an array of integers indicating the new ordering:

```
In [97]: df = pd.DataFrame(np.arange(5 * 4).reshape((5, 4)))

In [98]: sampler = np.random.permutation(5)

In [99]: sampler
Out[99]: array([3, 1, 4, 2, 0])
```

- That array can then be used in `iloc`-based indexing or the equivalent `take` function:

```
In [100]: df
Out[100]:
```

	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15
4	16	17	18	19

```
In [101]: df.take(sampler)
Out[101]:
```

	0	1	2	3
3	12	13	14	15
1	4	5	6	7
4	16	17	18	19
2	8	9	10	11
0	0	1	2	3

- To select a random subset without replacement, you can use the `sample` method on Series and DataFrame:

```
In [102]: df.sample(n=3)
Out[102]:
```

	0	1	2	3
3	12	13	14	15
4	16	17	18	19
2	8	9	10	11

- To generate a sample *with* replacement (to allow repeat choices), pass `replace=True` to `sample`:

```
In [103]: choices = pd.Series([5, 7, -1, 6, 4])

In [104]: draws = choices.sample(n=10, replace=True)

In [105]: draws
Out[105]: 4    4
1    7
4    4
2   -1
0    5
3    6
1    7
4    4
0    5
4    4
dtype: int64
```

Computing Indicator/Dummy Variables

- Another type of transformation for statistical modeling or machine learning applications is converting a categorical variable into a "dummy" or "indicator" matrix.
- If a column in a DataFrame has `k` distinct values, you would derive a matrix or DataFrame with `k` columns containing all 1s and 0s.
- pandas has a `get_dummies` function for doing this, though devising one yourself is not difficult.

```
In [106]: df = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
                             'data1': range(6)})

In [107]: pd.get_dummies(df['key'])
Out[107]:
```

	a	b	c
0	0	1	0
1	0	1	0
2	1	0	0
3	0	0	1
4	1	0	0
5	0	1	0

- In some cases, you may want to add a prefix to the columns in the indicator DataFrame, which can then be merged with the other data.
- `get_dummies` has a prefix argument for doing this:

```
In [108]: dummies = pd.get_dummies(df['key'], prefix='key')

In [109]: df_with_dummy = df[['data1']].join(dummies)

In [110]: df_with_dummy
Out[110]:
```

	data1	key_a	key_b	key_c
0	0	0	1	0
1	1	0	1	0
2	2	1	0	0
3	3	0	0	1
4	4	1	0	0
5	5	0	1	0

- If a row in a DataFrame belongs to multiple categories, things are a bit more complicated.
- Let’s look at the MovieLens 1M dataset.

```
In [111]: mnames = ['movie_id', 'title', 'genres']

In [112]: movies = pd.read_table('datasets/movielens/movies.dat', sep='::',
                                 header=None, names=mnames)

/home/joshua/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:2: FutureWarning: read_table is deprecated, use
read_csv instead.

/home/joshua/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:2: ParserWarning: Falling back to the 'python'
engine because the 'c' engine does not support regex separators (separators > 1 char and different from '\s+' are interp
reted as regex); you can avoid this warning by specifying engine='python'.

In [113]: movies[:10]

Out[113]:
```

	movie_id	title	genres
0	1	Toy Story (1995)	Animation|Children's|Comedy
1	2	Jumanji (1995)	Adventure|Children's|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama
4	5	Father of the Bride Part II (1995)	Comedy
5	6	Heat (1995)	Action|Crime|Thriller
6	7	Sabrina (1995)	Comedy|Romance
7	8	Tom and Huck (1995)	Adventure|Children's
8	9	Sudden Death (1995)	Action
9	10	GoldenEye (1995)	Action|Adventure|Thriller

- Adding indicator variables for each genre requires a little bit of wrangling.
- First, we extract the list of unique genres in the dataset:

```
In [114]: all_genres = []

In [115]: for x in movies.genres:
              all_genres.extend(x.split('|'))

In [116]: genres = pd.unique(all_genres)

In [117]: genres
Out[117]: array(['Animation', "Children's", 'Comedy', 'Adventure', 'Fantasy',
                 'Romance', 'Drama', 'Action', 'Crime', 'Thriller', 'Horror',
                 'Sci-Fi', 'Documentary', 'War', 'Musical', 'Mystery', 'Film-Noir',
                 'Western'], dtype=object)
```

- One way to construct the indicator DataFrame is to start with a DataFrame of all zeros:

```
In [118]: zero_matrix = np.zeros((len(movies), len(genres)))

In [119]: dummies = pd.DataFrame(zero_matrix, columns=genres)

In [120]: dummies[:5]
Out[120]:
```

	Animation	Children's	Comedy	Adventure	Fantasy	Romance	Drama	Action	Crime	Thriller	Horror	Sci-Fi	Documentary	War	Musical	Mystery	Film-Noir
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

- Now, iterate through each movie and set entries in each row of `dummies` to `1`.
- To do this, we use the `dummies.columns` to compute the column indices for each genre:

```
In [121]: gen = movies.genres[0]

In [122]: gen.split('|')
Out[122]: ['Animation', "Children's", 'Comedy']

In [123]: dummies.columns.get_indexer(gen.split('|'))
Out[123]: array([0, 1, 2])
```

- Then, we can use `.iloc` to set values based on these indices:

```
In [124]: for i, gen in enumerate(movies.genres):
              indices = dummies.columns.get_indexer(gen.split('|'))
              dummies.iloc[i, indices] = 1
```

- Then, as before, you can combine this with `movies`:

```
In [125]: movies_windic = movies.join(dummies.add_prefix('Genre_'))

In [126]: movies_windic.iloc[0]
Out[126]: movie_id                                       1
          title                           Toy Story (1995)
          genres               Animation|Children's|Comedy
          Genre_Animation                                1
          Genre_Children's                               1
          Genre_Comedy                                   1
          Genre_Adventure                                0
          Genre_Fantasy                                  0
          Genre_Romance                                  0
          Genre_Drama                                    0
                                      ...
          Genre_Crime                                    0
          Genre_Thriller                                 0
          Genre_Horror                                   0
          Genre_Sci-Fi                                   0
          Genre_Documentary                              0
          Genre_War                                      0
          Genre_Musical                                  0
          Genre_Mystery                                  0
          Genre_Film-Noir                                0
          Genre_Western                                  0
          Name: 0, Length: 21, dtype: object
```

- A useful recipe for statistical applications is to combine `get_dummies` with a discretization function like `cut`:

```
In [127]: np.random.seed(12345)

In [128]: values = np.random.rand(10)

In [129]: values
Out[129]: array([0.9296, 0.3164, 0.1839, 0.2046, 0.5677, 0.5955, 0.9645, 0.6532,
                 0.7489, 0.6536])

In [130]: bins = [0, 0.2, 0.4, 0.6, 0.8, 1]

In [131]: pd.get_dummies(pd.cut(values, bins))
Out[131]:
   (0.0, 0.2]  (0.2, 0.4]  (0.4, 0.6]  (0.6, 0.8]  (0.8, 1.0]
0           0           0           0           0           1
1           0           1           0           0           0
2           1           0           0           0           0
3           0           1           0           0           0
4           0           0           1           0           0
5           0           0           1           0           0
6           0           0           0           0           1
7           0           0           0           1           0
8           0           0           0           1           0
9           0           0           0           1           0
```