

## 5 Basic Practice (part 1 of 3)

### Feature Engineering

*Feature engineering* is the transforming of data into a set of labeled examples of selected features.

#### One-Hot Encoding

For an algorithm requiring only numerical features,<sup>1</sup> use *one-hot encoding* to transform a categorical feature into several binary features.

e.g. Transform categorical `color` feature into three binary features; only one is *hot* at a time:

(input)	(output)		
color	green	red	yellow
green	1	0	0
yellow	0	0	1
red	0	1	0
green	1	0	0

(Encoding `color` as `green=0`, `yellow=1`, `red=2` causes trouble because `red` isn't twice `yellow`, etc.)

This causes collinearity, in which one feature can be predicted from others, leading to numerically unstable computations. Address this by removing one column from the set of one-hot columns.

#### Python

- `import pandas as pd`  
  
# Make one-hot dummy variables from column(s) in DataFrame (or array) data:  
`pd.get_dummies(data, drop_first=False)` # also try `drop_first=True`  
  
# Join columns from other DataFrame:  
`df.join(other)`  
  
# Do both:  
`df.join(pd.get_dummies(df['column_to_encode'], drop_first=False))`
- User guide: [https://pandas.pydata.org/docs/user\\_guide/reshaping.html#reshaping-dummies](https://pandas.pydata.org/docs/user_guide/reshaping.html#reshaping-dummies)
- Reference manual:  
[https://pandas.pydata.org/docs/reference/api/pandas.get\\_dummies.html](https://pandas.pydata.org/docs/reference/api/pandas.get_dummies.html)  
<https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.join.html>

---

<sup>1</sup>Scikit-learn requires numerical features, but some of its algorithms convert categorical features automatically. Best practice does not rely on this conversion.

## Binning

*Binning* (or *bucketing*) converts a numeric feature to categorical.

e.g. Map numeric ages:

[0, 3):	baby
[3, 18):	child
[18, 65):	adult
[65, $\infty$ ):	senior

This can reduce the number of necessary training examples by reducing the complexity of the model and telling it values within a bin should be treated the same.

## Python

- `import pandas as pd`
- `pd.cut(x, bins, right=True, labels=None)` puts values from `x` (array-like) into `bins` bins, if `bins` is a number, or into bins whose edges are in the sequence `bins`, that (by default) include the rightmost edge. Use provided `labels` (if not `None`).
- User guide: [https://pandas.pydata.org/docs/user\\_guide/reshaping.html#tiling](https://pandas.pydata.org/docs/user_guide/reshaping.html#tiling)
- Reference manual: <https://pandas.pydata.org/docs/reference/api/pandas.cut.html>

→

## Rescaling via Normalization or Standardization

- *Normalization* converts a numeric range into a standard range.

e.g. *Min-max* normalization replaces  $x^{(j)}$  with  $\frac{x^{(j)} - \min^{(j)}}{\max^{(j)} - \min^{(j)}} \in [0, 1]$ .

This can improve training speed by preventing a large-scale feature from dominating a small-scale one early in a gradient descent search. It can help prevent numeric underflow or overflow. It can improve performance.

- *Standardization* (or *z-score* normalization) replaces  $x^{(j)}$  with  $\frac{x^{(j)} - \mu^{(j)}}{\sigma^{(j)}}$ . If  $x^{(j)} \sim N(\mu^{(j)}, \sigma^{(j)})$ , where  $\mu^{(j)}$  and  $\sigma^{(j)}$  are the feature mean and standard deviation, then the standardized values follow the *standard normal distribution*  $N(0, 1)$ .

Experiment with both, considering:

- standardization often works better for:
  - unsupervised algorithms
  - an  $\approx$  normal feature
  - a feature with outliers (as normalization would squeeze values into a small range)
- normalization often works better otherwise
- rescaling often helps

## Python

- `from sklearn.preprocessing import MinMaxScaler`

```
scaler = MinMaxScaler()
scaler.fit_transform(X) # do scaling
scaler.inverse_transform(X) # undo scaling
```

- `from sklearn.preprocessing import StandardScaler`

```
scaler = StandardScaler()
scaler.fit_transform(X) # do scaling
scaler.inverse_transform(X) # undo scaling
```

- User guide: <https://scikit-learn.org/stable/modules/preprocessing.html>

- Reference manual:

<https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html>

<https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html>

## Missing Features and Data Imputation

Three options for handling missing features:

- Remove examples with missing features.
- Use an algorithm (or implementation) that can handle missing features.
- Use *data imputation*, which replaces a missing feature value  $x_i^{(j)}$  with a computed value:
  - the feature mean,  $\frac{1}{M} \sum_{i=1}^M x_i^{(j)}$ , where  $M < N$  is the number of examples with feature  $j$  present and the sum is over non-missing values
  - a value outside the feature's normal range; e.g. if the range is  $[0, 1]$ , use  $-1$  or  $2$  (then the model can learn how to handle the non-normal value)
  - the midpoint of the feature's normal range; e.g. if the range is  $[0, 1]$ , use  $0.5$  (hoping this will not significantly affect the prediction)
  - the predicted value from a regression model for  $y = x^{(j)}$  trained from the remaining features over the examples not missing  $x^{(j)}$
  - $0$  (or some other value) while adding a feature with values in  $\{0, 1\}$  to say for each example whether  $x_i^{(j)}$  is present

When predicting  $y$  for a new  $\mathbf{x}$ , apply the same imputation method.

## Python

- `df.dropna()` drops rows missing at least one element. For more on what you can drop, see <https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.dropna.html>.  
Missing data user guide: [https://pandas.pydata.org/docs/user\\_guide/missing\\_data.html](https://pandas.pydata.org/docs/user_guide/missing_data.html).
- `from sklearn.impute import SimpleImputer`
- `imp = SimpleImputer(missing_values=nan, strategy='mean', fill_value=None)` makes an imputer that uses the mean;
  - `strategy='median'` uses the median
  - `strategy='constant'` uses `fill_value`
  - `strategy='most_frequent'` uses the (smallest) most frequent feature value, which may be useful with strings or numeric data
- `imp.fit_transform(X)` does the imputation
- User guide: <https://scikit-learn.org/stable/modules/impute.html>
- Reference manual:  
<https://scikit-learn.org/stable/modules/generated/sklearn.impute.SimpleImputer.html>

## Feature Selection

*Feature selection* is the process of choosing a subset of features for use in a model. It can:

- Improve accuracy by reducing overfitting
- Improve computing performance (time, memory, disk space)
- Make model easier to interpret

## Python

For each of the following methods:

- Import METHOD via `from sklearn.feature_selection import METHOD`
- Set `selector = METHOD()` (with appropriate parameters)
- Call `selector.fit_transform(X)` or `...transform(X, y)` to get a smaller feature array

Here are some options for METHOD:

- `VarianceThreshold(threshold=0.0)` removes features with variance less than `threshold`. The return value includes `variances_`, an array of variances of the features.  
e.g. `threshold=0.0` removes features that do not vary.  
e.g. For  $B \sim \text{Bernoulli}(p)$ ,  $\text{VAR}(B) = p(1 - p)$ , so we could remove binary features with  $p = P(y = 1) \leq 0.1$  or  $p \geq 0.9$  with `threshold=0.09`.
- Methods based on univariate statistics use a `score_func(X, y)` which returns an array of scores or a pair of arrays of scores and p-values.
  - `SelectKBest(score_func, k=10)` retains the `k` best-scoring features.
  - `SelectPercentile(score_func, percentile=10)` retains the proportion `percentile` of features ranked by their scores.

Available `score_func` include:

- For regression, `r_regression`, whose return value includes `correlation_coefficient`, an array of correlations between the  $j$ th feature and  $y$ . This is problematic because  $r$  tending toward 1 or  $-1$  indicates predictive power, but “highest score” neglects  $-1$ .
- For regression, `f_regression`, which returns a pair of arrays, `f_statistic` of scores and `p_values` of p-values.

Recall:  $R^2 = \frac{SSR}{SST}$ ,  $1 - R^2 = \frac{SSE}{SST}$ , and  $F = \frac{MSR}{MSE} = \frac{SSR/1}{SSE/(n-2)} = \frac{R^2 SST}{(1-R^2)SST}(n-2) = \frac{R^2}{1-R^2}(n-2)$  and  $R^2 = r^2$ . So `f_regression` is computed from `r_regression`. Since  $F > 0$ , it does not suffer from the `r_regression` problem.

- For classification, `chi2` finds a  $\chi^2$  statistic between each non-negative feature and `y`.

To learn more:

- User guide: [https://scikit-learn.org/stable/modules/feature\\_selection](https://scikit-learn.org/stable/modules/feature_selection)
- API reference: [https://scikit-learn.org/stable/modules/classes.html#module-sklearn.feature\\_selection](https://scikit-learn.org/stable/modules/classes.html#module-sklearn.feature_selection)
- Examples: [https://scikit-learn.org/stable/auto\\_examples/index.html#feature-selection](https://scikit-learn.org/stable/auto_examples/index.html#feature-selection)

## Feature Importance

The *permutation feature importance* of feature  $j$  relative to a model is the decrease in the model score when the feature is randomly shuffled, breaking the relationship between it and `y`.

Cautions:

- A feature could be of low importance in one model but high in another.
- Small groups of correlated features may be favored over larger groups.
- None of a set of collinear features may show importance.

Retaining only one of each cluster (§9) of correlated features addresses the last two problems.

## Python

- `from sklearn.inspection import permutation_importance`
- `permutation_importance(estimator, X, y, scoring=None, random_state=None)`
  - `estimator` is already fit
  - `X`, `y` are the data (training or validation) on which importance is calculated
  - `scoring` is the scorer to use; if `None`, the estimator's default
  - The return value contains `importances_mean`, `importances_std`, and `importances`

To learn more:

- User guide: [https://scikit-learn.org/stable/modules/permutation\\_importance.html](https://scikit-learn.org/stable/modules/permutation_importance.html)
- API reference: [https://scikit-learn.org/stable/modules/generated/sklearn.inspection.permutation\\_importance.html](https://scikit-learn.org/stable/modules/generated/sklearn.inspection.permutation_importance.html)
- Examples: [https://scikit-learn.org/stable/auto\\_examples/inspection/plot\\_permutation\\_importance\\_multicollinear.html](https://scikit-learn.org/stable/auto_examples/inspection/plot_permutation_importance_multicollinear.html)