Popular Machine Learning Methods: Idea, Practice and Math

Part 2, Chapter 1, Section 1: Data Preprocessing

Yuxiao Huang

Data Science, Columbian College of Arts & Sciences
George Washington University

Spring 2025

Reference

- This set of slices was largely built on the following 7 wonderful books and a wide range of fabulous papers:

HML	Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)
PML	Python Machine Learning (3rd Edition)
ESL	The Elements of Statistical Learning (2nd Edition)
PRML	Pattern Recognition and Machine Learning
NND	Neural Network Design (2nd Edition)
LFD	Learning From Data
RL	Reinforcement Learning: An Introduction (2nd Edition)

- For most materials covered in the slides, we will specify their corresponding books and papers for further reference.

Code Example

- See related code example in github repository: /p2_c1_data_preprocessing/code_example

Table of Contents

Learning Objectives: Expectation

- It is expected to understand
  - the importance of data preprocessing
  - the idea, implementation and good practice of each step in the pipeline of data preprocessing
  - the similarity and difference between:
    - the pipeline of data preprocessing for regression
    - the pipeline of data preprocessing for classification

Learning Objectives: Recommendation

- It is recommended to understand
  - the math behind the following steps in the pipeline of data preprocessing:
    - scaling the data
    - handling class imbalance
    - feature selection / feature engineering

Kaggle Competition: Predicting House Price

Figure 1: Kaggle competition: predicting house price. Picture courtesy of Kaggle.

- House Prices (Advanced Regression Techniques) dataset:
    - features: 79 explanatory variables describing (almost) every aspect of residential homes in Ames, Iowa
    - target: the sale price of homes

Table 1: The first 7 features and target (SalePrice) of House Prices dataset.

Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	SalePrice
1	60	RL	65.0	8450	Pave	NaN	208500
2	20	RL	80.0	9600	Pave	NaN	181500
3	60	RL	68.0	11250	Pave	NaN	223500
4	70	RL	60.0	9550	Pave	NaN	140000
5	60	RL	84.0	14260	Pave	NaN	250000

Kaggle Competition: Predicting Breast Cancer

Dataset

Breast Cancer Wisconsin (Diagnostic) Data Set

Predict whether the cancer is benign or malignant

UCI ML UCI Machine Learning • updated 4 years ago (Version 2)

Figure 2: Kaggle competition: predicting breast cancer. Picture courtesy of Kaggle.

- Breast Cancer Wisconsin (Diagnostic) dataset:
  - features: ID number + 30 variables computed from a digitized image of a fine needle aspirate (FNA) of a breast mass, describing characteristics of the cell nuclei present in the image
  - target: the diagnosis of breast cancer, Benign (B) or Malignant (M)

Table 2: The first 5 features and target (diagnosis) of Breast Cancer Wisconsin dataset.

id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean
842302	M	17.99	10.38	122.80	1001.0
842517	M	20.57	17.77	132.90	1326.0
84300903	M	19.69	21.25	130.00	1203.0
84348301	M	11.42	20.38	77.58	386.1
84358402	M	20.29	14.34	135.10	1297.0

Motivation

- As we discussed in /p1_introduction_to_machine_learning, if we were to directly feed the data to machine learning systems, the systems either would not work at all or, at least, would not work well.
- As a result, we need *Data Preprocessing* to processes data prior to feeding the data to the systems (hence the name of data preprocessing).
- Anecdotally, machine learning scientists could spend 80% of their time on data preprocessing.
- In this chapter, we will use the two datasets (House Prices and Breast Cancer Wisconsin) on pages 7 and 8 to introduce the pipeline for data preprocessing in shallow learning (i.e., when the data are not image or text):
  - House Prices is used for showing the pipeline for regression
  - Breast Cancer Wisconsin is used for showing the pipeline for classification
- We will introduce the pipeline for data preprocessing in deep learning in /p3_c1_data_preprocessing.

The Pipeline for Regression and Classification

Table 3: The Pipeline for Regression and Classification.

Order	Step	Regression	Classification
1	Loading the data	Always	Always
2	Splitting the data	Always	Always
3	Handling uncommon features	Always	Always
4	Handling identifiers	Always	Always
5	Handling date time variables	Always	Always
6	Handling missing data	Always	Always
7	Encoding the data	Always	Always
8	Splitting the feature and target	Always	Always
9	Scaling the data	Always	Always
10	Handling class imbalance	Never	Always
11	Feature selection & feature engineering	Sometimes	Sometimes

Good practice

- While the order for some steps in table 3 are interchangeable, it is highly recommended to follow the provided order.
- We will have to add other steps to table 3 when preprocessing some data.

Step 1: Loading the Data

- Unsurprisingly, the 1st step in data preprocessing is loading the data.
- The code below shows how to load the data:
    - see /p2_c1_data_preprocessing/code_example/regression:
        1. cell 7
    - see /p2_c1_data_preprocessing/code_example/classification:
        1. cell 7
    - the code listings above are the same in regression and classification

Step 2: Splitting the Data

- The 2nd step in data preprocessing is splitting the data into three subsets:
  - *Training data*:
    - this is where we train the models
    - we will discuss the training method for each model covered in this set of slides
  - *Validation data*:
    - this is where we perform *Hyperparameter Tuning* and *Model Selection*
    - we will discuss this in:
      /p2_c2_s2_training_shallow_models
      /p3_c2_s2_training_deep_models
  - *Test data*:
    - this is where we test how well the models generalize in reality
    - we will discuss the difference between training and test performance in:
      /p2_c2_s2_training_shallow_models
- Here we only focus on methods for splitting data that is at least moderately large (e.g., with at least hundreds of samples).
- For small datasets, we use *Cross Validation* (see HML: Chap 2).

Step 2: Splitting the Data

- When both training and test data are available (this is usually the case for kaggle competitions):
  - we only need to split the training data into:
    - training data
    - validation data
  - some recommended ratios for splitting the training data:
    - 80% for training, 20% for validation
    - 70% for training, 30% for validation
    - ...
- When only one dataset is available (this is usually the case when we collect data ourselves):
  - we need to split the data into:
    - training data
    - validation data
    - test data
  - some recommended ratios for splitting the training data:
    - 60% for training, 20% for validation, 20% for test
    - 70% for training, 15% for validation, 15% for test
    - ...

Step 2: Splitting the Data

- Generally, the larger the data the larger the proportion for training, since:
  - we would rather use a sample for training than for validation or test
  - for large data (e.g., with 1 million samples), even a small proportion (e.g., 10%) for validation and test may still be sufficient
- For classification (i.e., the target is discrete), it is essential to use the *Stratified* approach to split the data, so that:
  - the proportion for each class in the original data (prior to the split) will be similar to the proportion for each class in the split data
  - for example:
    - both training and test data are available (thus we only need to split the training data into training and validation)
    - the proportion for class 0 / class 1 are 70% / 30% in the training data
    - after the split, the proportion for class 0 / class 1 should be similar to 70% / 30% in the training and validation data

Good practice

- Generally, the larger the data the larger the proportion for training.
- For classification, it is essential to use the *Stratified* approach to split the data.

Step 2: Splitting the Data

- The code below shows how to divide the training data into training and validation:
    - see /p2_c1_data_preprocessing/code_example/regression:
        1. cells 12 to 14
- The code below shows how to divide the data into training, validation and test:
    - see /p2_c1_data_preprocessing/code_example/classification:
        1. cells 10 to 13
- The code for splitting the data are different in regression and classification:
    - in regression, it is not essential to use the stratified approach
    - in classification, it is essential to use the stratified approach

Step 3: Handling Uncommon Features

- After splitting the data into training, validation and test data, we should handle the *Uncommon Features*, which are features that are not shared between the three kinds of data (i.e., training, validation and test):
  - assume the features in the three kinds of data are as follows:
    - training: $\mathbf{x}_1$ and $\mathbf{x}_2$
    - validation: $\mathbf{x}_1$ and $\mathbf{x}_2$
    - test: $\mathbf{x}_1$
  - then the uncommon feature is $\mathbf{x}_2$, as it is only in training and validation data but not in test data
- We need to handle uncommon features to make sure training, validation and test data have the exact same features (i.e., there are no uncommon features).
- Otherwise, we will not be able to validate or test the model trained on the training data (since the input features are different).

Identifying Uncommon Features

- Finding the uncommon features manually could be difficult, particularly when there are many features in the data.
- The code below shows how to automatically find the uncommon features from the data:
  - see /utilities/p2_shallow_learning/pmlm_utilities_shallow:
    1. cell 1
  - see /p2_c1_data_preprocessing/code_example/regression:
    1. cells 15 to 18
  - see /p2_c1_data_preprocessing/code_example/classification:
    1. cells 14 to 17
  - the code listings above are the same in regression and classification

Removing Uncommon Features

- After finding the uncommon features, we need to remove them from the data, so that the training, validation and test data will have the exact same features.
- The code below shows how to remove the uncommon features from the data:
  - see /p2_c1_data_preprocessing/code_example/regression:
    1. cells 19 to 21
  - see /p2_c1_data_preprocessing/code_example/classification:
    1. cells 18 to 20
  - the code listings above are the same in regression and classification

Step 4: Handling Identifiers

- An *Identifier* is a feature whose value is unique for each sample.
- For example, the first feature in tables 1 and 2 are identifiers.

Table 1: The first 7 features and target (SalePrice) of House Prices dataset.

Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	SalePrice
1	60	RL	65.0	8450	Pave	NaN	208500
2	20	RL	80.0	9600	Pave	NaN	181500
3	60	RL	68.0	11250	Pave	NaN	223500
4	70	RL	60.0	9550	Pave	NaN	140000
5	60	RL	84.0	14260	Pave	NaN	250000

Table 2: The first 5 features and target (diagnosis) of Breast Cancer Wisconsin dataset.

id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean
842302	M	17.99	10.38	122.80	1001.0
842517	M	20.57	17.77	132.90	1326.0
84300903	M	19.69	21.25	130.00	1203.0
84348301	M	11.42	20.38	77.58	386.1
84358402	M	20.29	14.34	135.10	1297.0

Identifying Identifiers

- Finding the identifiers manually could be difficult, particularly when there are many columns (i.e., variables) and rows (i.e., samples) in the data.
- The code below shows how to automatically find identifiers from the data:
    - see /utilities/p2_shallow_learning/pmlm_utilities_shallow:
        1. cell 2
    - see /p2_c1_data_preprocessing/code_example/regression:
        1. cells 22 and 23
    - see /p2_c1_data_preprocessing/code_example/classification:
        1. cells 21 and 22
    - the code listings above are the same in regression and classification

Removing Identifiers

- After finding the identifiers we should remove them from the data.
- The reason is closely related to the definition of these features:
  - since their value vary from one sample to another, they are irrelevant to any useful pattern (a relationship true in more than one sample)
  - in other words, the identifiers have no predictive power over the target
  - as a result, we can remove the identifiers without compromising prediction accuracy
  - conversely, if we did not remove the identifiers, it is very likely that we would find them at least remotely related to the target, resulting in compromising prediction accuracy
- The code below shows how to remove identifiers from the data:
  - see /p2_c1_data_preprocessing/code_example/regression:
    1. cells 24 to 27
  - see /p2_c1_data_preprocessing/code_example/classification:
    1. cells 23 to 26
  - the code listings above are the same in regression and classification

Step 5: Handling Date Time Variables

- A *Date Time* variable say, 2017-09-19 16:09:00, is a variable that has date and time information:
    - year: 2017
    - month: 09
    - day: 19
    - hour: 16
    - minute: 09
    - second: 00
- We should transform data time variables into the above 6 datetime types before feeding them to a model.
- The code below shows how to transform data time variables into 6 datetime types (year, month, day, hour, minute and second):
    - see /utilities/p2_shallow_learning/pmlm_utilities_shallow:
        1. cell 3
    - see /p2_c1_data_preprocessing/code_example/regression:
        1. cells 28 to 31
    - see /p2_c1_data_preprocessing/code_example/classification:
        1. cells 27 to 30
    - the code listings above are the same in regression and classification

Step 6: Handling Missing Data

- While we expect that a variable's value was recorded in each sample, in reality their values could be missing in some or even most of the samples.
- We should handle missing data prior to feeding them to a model, otherwise:
    - either training will crash
    - or the accuracy of the trained model will be compromised

Identifying Missing Values

- Missing data can be represented by many symbols, for example:
  - blank space
  - NA or N/A (i.e., Not Applicable)
  - NaN or nan (i.e., Not A Number)
- For simplicity, hereafter we will use NaN to denote all these symbols that can represent missing data.
- One way to find missing data is looking for NaN.
- However, finding NaN manually could be difficult, particularly when there are many columns and rows in the data
- The code below shows how to automatically find variables with NaN:
  - see /utilities/p2_shallow_learning/pmlm_utilities_shallow:
    1. cell 4
  - see /p2_c1_data_preprocessing/code_example/regression:
    1. cells 32 and 33
  - see /p2_c1_data_preprocessing/code_example/classification:
    1. cells 31 and 32
  - the code listings above are the same in regression and classification

Identifying Missing Values

Table 1: The first 7 features and target (SalePrice) of House Prices dataset.

Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	SalePrice
1	60	RL	65.0	8450	Pave	NaN	208500
2	20	RL	80.0	9600	Pave	NaN	181500
3	60	RL	68.0	11250	Pave	NaN	223500
4	70	RL	60.0	9550	Pave	NaN	140000
5	60	RL	84.0	14260	Pave	NaN	250000

- It is worth noting that, while NaN can represent missing value, it can also be meaningful value.
- For example, NaN in feature Alley in table 1:
  - does not represent missing value
  - instead, it means no alley access
- As a result, we need to find whether NaN in a variable represents missing value.

⚠ **Caveat**

- While NaN can represent missing value, it can also be meaningful value.

Identifying Missing Values

- The best way to find whether NaN in a variable represents missing value is using the prior knowledge (e.g., the NaN in Alley means no alley access).
- However, when such knowledge is not available, a lazy workaround is using the data type of the variable:
  - if the data type of the variable is numerical (e.g., `int` or `float`), we treat NaN in the variable as missing value
  - otherwise, we treat NaN as meaningful value
- In other words, a lazy way to find variables with missing values is finding numerical variables with NaN.
- The code below shows how to find variables with missing values:
  - see /p2_c1_data_preprocessing/code_example/regression:
    1. cells 34 and 35
  - see /p2_c1_data_preprocessing/code_example/classification:
    1. cells 33 and 34
  - the code listings above are the same in regression and classification

Removing Missing Values

- After identifying missing data, the most straightforward way to handle such data is simply removing them.
- That is, if row i (i.e., ith sample in the data) has missing values, we remove this row from the data.
- The code below shows how to remove rows with missing values:
  - see /p2_c1_data_preprocessing/code_example/regression:
    1. cell 40
  - see /p2_c1_data_preprocessing/code_example/classification:
    1. cell 39
  - the code listings above are the same in regression and classification

Removing Missing Values

- While removing rows with missing values is simple, it may also be problematic:
  - if many values in a column (i.e., a feature) were missing, we would end up removing many rows
  - the remaining data may be too small to be useful (for training / validation / test)
- To handle this problem, instead of removing rows with missing values, we can remove the columns where many values are missing.
- The code below shows how to remove a column from the data:
  - see /p2_c1_data_preprocessing/code_example/regression:
    1. cells 19 to 21
  - see /p2_c1_data_preprocessing/code_example/classification:
    1. cells 18 to 20
  - the code listings above are the same in regression and classification

Good practice

- If a feature has many missing values (e.g., $> 50\%$), before removing rows with missing values, we should remove the feature first.

Imputing Missing Values

- Besides removing missing data, another way to handle missing data is imputing the data.
- The idea is using the statistics of a feature's non-missing values to impute its missing values:
  1. we calculate the statistics of a feature's non-missing values on the training data, a step usually referred to as *fit*
  2. we use the calculated statistics to impute the feature's missing values on the training / validation / test data, a step usually referred to as *transform*
- **Q:** Why should not we use the statistics of non-missing values on the validation / test data to impute the missing values on the validation / test data?

Imputing Missing Values

- Besides removing missing data, another way to handle missing data is imputing the data.
- The idea is using the statistics of a feature's non-missing values to impute its missing values:
  1. we calculate the statistics of a feature's non-missing values on the training data, a step usually referred to as *fit*
  2. we use the calculated statistics to impute the feature's missing values on the training / validation / test data, a step usually referred to as *transform*
- **Q:** Why should not we use the statistics of non-missing values on the validation / test data to impute the missing values on the validation / test data?
- **A:** This is largely because the training data is usually much bigger than the validation / test data, so that the statistics calculated on the training data could be much more accurate (much closer to the statistics of the population).
- **Q:** Why should not we use the statistics of non-missing values on the combined training, validation and test data to impute the missing values on the training / validation / test data?

Imputing Missing Values

- Besides removing missing data, another way to handle missing data is imputing the data.
- The idea is using the statistics of a feature's non-missing values to impute its missing values:
  1. we calculate the statistics of a feature's non-missing values on the training data, a step usually referred to as *fit*
  2. we use the calculated statistics to impute the feature's missing values on the training / validation / test data, a step usually referred to as *transform*
- **Q:** Why should not we use the statistics of non-missing values on the validation / test data to impute the missing values on the validation / test data?
- **A:** This is largely because the training data is usually much bigger than the validation / test data, so that the statistics calculated on the training data could be much more accurate (much closer to the statistics of the population).
- **Q:** Why should not we use the statistics of non-missing values on the combined training, validation and test data to impute the missing values on the training / validation / test data?
- **A:** This is because then the imputed training data will have information of the validation / test data (a problem usually referred to as *Data Leakage*), so that a model's performance on validation / test data will be overestimated.

Imputing Missing Values

Takeaway

- When imputing missing data we should:

1. calculate the statistics of a feature's non-missing values on the training data
2. use the calculated statistics to impute the feature's missing values on the training / validation / test data

- The statistics (of a feature) that we can use for imputation include, for example:
  - *Mean*: the average of the feature's values
  - *Median*: the middle value after sorting the feature's values
  - *Mode*: the most frequent value of the feature's values
- Which statistic we should use depends on the type of a feature:
  - numerical feature: mean / median / mode
  - categorical (a.k.a., non-numerical) feature: mode
- The code below shows how to impute missing values using the mean:
  - see /p2_c1_data_preprocessing/code_example/regression:
    1. cell 41
  - see /p2_c1_data_preprocessing/code_example/classification:
    1. cell 40
  - the code listings above are the same in regression and classification

Imputing Missing Values

- There are many methods that go way beyond simply using the mean / median / mode to impute missing data:
    - [Yoon et al., 2018, Li et al., 2019] are two nice papers that propose imputing missing data using Generative Adversarial Networks (GANs)
    - we will discuss GANs in /p3_c3_s1_deep_generative_models

Step 7: Encoding Categorical Data

Table 1: The first 7 features and target (SalePrice) of House Prices dataset.

Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	SalePrice
1	60	RL	65.0	8450	Pave	NaN	208500
2	20	RL	80.0	9600	Pave	NaN	181500
3	60	RL	68.0	11250	Pave	NaN	223500
4	70	RL	60.0	9550	Pave	NaN	140000
5	60	RL	84.0	14260	Pave	NaN	250000

- After handling missing values, we need to handle categorical (i.e., non-numerical) variables.
- For example, there are three categorical variables in table 1:
  - MSZoning
  - Street
  - Alley
- Unfortunately, most models cannot work on categorical variables (one exception is CatBoost).
- This is because training usually involves numerical calculations, which cannot be performed on categorical values.
- Thus we need to encode (transform) the categorical variables into numerical ones.

Identifying Categorical Variables

- Finding categorical variables manually could be difficult when there are many variables in the data.
- A straightforward way to automatically find categorical variables is using the data type of a variable:
  - if the data type of the variable is numerical (e.g., `int` or `float`), the variable is numerical
  - otherwise, the variable is categorical
- The code below shows how to find categorical variables:
  - see /utilities/p2_shallow_learning/pmlm_utilities_shallow:
    1. cell 5
  - see /p2_c1_data_preprocessing/code_example/regression:
    1. cells 42 and 43
  - see /p2_c1_data_preprocessing/code_example/classification:
    1. cells 41 and 42
  - the code listings above are the same in regression and classification

Encoding Categorical Features

- While there are many ways to encode categorical features, here we focus on one of the most widely used method, namely *One-Hot-Encoding* (OHE).
- The idea of OHE is quite straightforward:
  - assume a categorical feature has k unique values
  - in OHE we simply transform the feature into k feature-value pairs, with each pair taking value 0 or 1
- Table 4 shows using OHE to encode feature Alley, which has three unique values:
  - Grvl (Gravel)
  - Pave (Paved)
  - NA (No alley access)

  into three feature-value pairs:
  - Alley_Grvl
  - Alley_Pave
  - Alley_NA

Table 4: Encoding a categorical feature (Alley) into feature-value pairs using OHE.

Alley
Grvl
Pave
NA

$\xrightarrow{\text{OHE}}$

Alley_Grvl	Alley_Pave	Alley_NA
1	0	0
0	1	0
0	0	1

Encoding Categorical Features

- The code below shows how to encode categorical features:
  - see /p2_c1_data_preprocessing/code_example/regression:
    1. cell 44
  - see /p2_c1_data_preprocessing/code_example/classification:
    1. cell 43
  - the code listings above are the same in regression and classification
- As mentioned earlier, OHE encodes a feature with k unique values into k feature-value pairs.
- Thus when k is large, the one-hot-encoded data can be much larger than the original data (in terms of the number of columns, a.k.a., dimensions).
- There are several ways to handle this problem:
  - one method is using *Feature Engineering* to combine the k values into n groups (where $n \ll k$, see /p2_c3_unsupervised_learning)
  - a more advanced approach is *Embedding* (see /p3_c1_data_preprocessing)
  - another way is using CatBoost, which allows categorical features

⚠ Caveat

- When the data have categorical features with many unique values, the one-hot-encoded data can be much larger than the original data (in terms of the number of columns, a.k.a., dimensions).

Encoding Categorical Target

Table 2: The first 5 features and target (diagnosis) of Breast Cancer Wisconsin dataset.

id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean
842302	M	17.99	10.38	122.80	1001.0
842517	M	20.57	17.77	132.90	1326.0
84300903	M	19.69	21.25	130.00	1203.0
84348301	M	11.42	20.38	77.58	386.1
84358402	M	20.29	14.34	135.10	1297.0

- Similar to features that can be categorical, the target can also be categorical.
- For example, the target in table 2, diagnosis, can take two categorical values:
  - M (Malignant)
  - B (Benign)
- However, unlike categorical features (where we use OHE to encode the features into feature-value pairs), we can encode categorical values of the target into any integers.
- Table 5 shows encoding the categorical target (diagnosis) into integers.

Table 5: Encoding the categorical target (diagnosis) into integers.

diagnosis
M
B

$\xrightarrow{\text{OHE}}$

diagnosis
4
2

Encoding Categorical Target

- The code below shows how to encode categorical target:
  - see /p2_c1_data_preprocessing/code_example/classification:
    1. cell 44
- The reason why we do not use OHE to encode categorical target into target-value pairs (as what we did for categorical features) is that:
  - OHE will create a 2d $m \times n$ matrix (where m is the number of samples and n the number of target-value pairs)
  - however, most of the machine learning tools require the target to be a 1d $m \times 1$ array (where m is the number of samples)
- It does not matter which integer we use to encode the categorical target (as long as the integers are different), as these integers are simply identifiers and will not be used in downstream numerical calculations.

Takeaway

- We use one-hot-encoding to encode categorical features into feature-value pairs.
- For categorical target, we can encode their values into any integers.

Step 8: Splitting the Feature and Target

- After encoding the data, we:
  - split the feature and target
  - convert the features and target from *Pandas DataFrame* into *Numpy Array* (a data structure widely used by most machine learning tools)
- The code below shows how to split the feature and target:
  - see /p2_c1_data_preprocessing/code_example/classification:
    1. cell 49
  - see /p2_c1_data_preprocessing/code_example/classification:
    1. cell 49
  - the code listings above are the same in regression and classification

Good practice

- After splitting the feature and target, it is recommended to convert the features and target from *Pandas DataFrame* into *Numpy Array*.

Step 9: Scaling the Data

Table 6: The first five samples of features LotArea and OverallQual in the training data of House Prices dataset.

LotArea	OverallQual
8450	7
9600	6
11250	7
9550	7
14260	8

- While all the variables are numerical after encoding the categorical data, we still cannot feed the data to a model.
- One reason is that the scale of the variables may be very different.
- Table 6 shows that features LotArea and OverallQual are on quite different scales: LotArea is (most of the time) more than 1000 times larger than OverallQual.

Problem of Data at Different Scales

- It turns out that if we were to train a model on data with such different scales, the prediction accuracy would be significantly lowered.
- This is because LotArea could end up silencing OverallQual when predicting the target SalePrice (we will discuss this in /p2_c2_s1_linear_regression).
- That is, the predicted SalePrice will hardly change with OverallQual.
- This is, however, not the case in reality (a house with high overall quality could be much more expensive than one with low quality).
- To handle this problem, we need to transform the data into similar scales.
- Here we introduce two ways to do so:
  - Standardization
  - Min-Max Normalization

⚠ **Caveat**

- Training a model on data with different scales could severely compromise the performance of the model.

Standardization

- We standardize each variable in the training data, $\mathbf{v}_{\text{train}}$, in the following steps:
  1. calculate the mean and standard deviation (std) of $\mathbf{v}_{\text{train}}$, $\mu(\mathbf{v}_{\text{train}})$ and $\sigma(\mathbf{v}_{\text{train}})$
  2. calculate the standardized $\mathbf{v}_{\text{train}}$, $\mathbf{v}_{\text{ss\_train}}$:

$$\mathbf{v}_{\text{ss\_train}} = \frac{\mathbf{v}_{\text{train}} - \mu(\mathbf{v}_{\text{train}})}{\sigma(\mathbf{v}_{\text{train}})}. \tag{1}$$

- We standardize each variable in the validation and test data, $\mathbf{v}_{\text{val}}$ and $\mathbf{v}_{\text{test}}$, in the following steps:
  1. reuse the mean and std of $\mathbf{v}_{\text{train}}$, $\mu(\mathbf{v}_{\text{train}})$ and $\sigma(\mathbf{v}_{\text{train}})$
  2. calculate the standardized $\mathbf{v}_{\text{val}}$ and $\mathbf{v}_{\text{test}}$, $\mathbf{v}_{\text{ss\_val}}$ and $\mathbf{v}_{\text{ss\_test}}$:

$$\begin{aligned} \mathbf{v}_{\text{ss\_val}} &= \frac{\mathbf{v}_{\text{val}} - \mu(\mathbf{v}_{\text{train}})}{\sigma(\mathbf{v}_{\text{train}})}, \\ \mathbf{v}_{\text{ss\_test}} &= \frac{\mathbf{v}_{\text{test}} - \mu(\mathbf{v}_{\text{train}})}{\sigma(\mathbf{v}_{\text{train}})}. \end{aligned} \tag{2}$$

Min-Max Normalization

- We normalize each variable in the training data, $\mathbf{v}_{\text{train}}$, into range $[a, b]$ (two parameters of the method) in the following steps:
  1. calculate the $\min$ and $\max$ of $\mathbf{v}_{\text{train}}$, $\min(\mathbf{v}_{\text{train}})$ and $\max(\mathbf{v}_{\text{train}})$
  2. calculate the normalized $\mathbf{v}_{\text{train}}$, $\mathbf{v}_{\text{mms\_train}}$:

$$\mathbf{v}_{\text{mms\_train}} = \frac{\mathbf{v}_{\text{train}} - \min(\mathbf{v}_{\text{train}})}{\max(\mathbf{v}_{\text{train}}) - \min(\mathbf{v}_{\text{train}})} \times (b - a) + a. \tag{3}$$

- We normalize each variable in the validation and test data, $\mathbf{v}_{\text{val}}$ and $\mathbf{v}_{\text{test}}$, in the following steps:
  1. reuse the $\min$ and $\max$ of $\mathbf{v}_{\text{train}}$, $\min(\mathbf{v}_{\text{train}})$ and $\max(\mathbf{v}_{\text{train}})$
  2. calculate the normalized $\mathbf{v}_{\text{val}}$ and $\mathbf{v}_{\text{test}}$, $\mathbf{v}_{\text{mms\_val}}$ and $\mathbf{v}_{\text{mms\_test}}$:

$$\begin{aligned} \mathbf{v}_{\text{mms\_val}} &= \frac{\mathbf{v}_{\text{val}} - \min(\mathbf{v}_{\text{train}})}{\max(\mathbf{v}_{\text{train}}) - \min(\mathbf{v}_{\text{train}})} \times (b - a) + a, \\ \mathbf{v}_{\text{mms\_test}} &= \frac{\mathbf{v}_{\text{test}} - \min(\mathbf{v}_{\text{train}})}{\max(\mathbf{v}_{\text{train}}) - \min(\mathbf{v}_{\text{train}})} \times (b - a) + a. \end{aligned} \tag{4}$$

- See proof of why $\mathbf{v}_{\text{mms\_train}}$ in eq. (3) falls in range $[a, b]$ in Appendix (pages 58 and 59).

Min-Max Normalization

- However, reusing the min and max in the training data to normalize the validation and test data comes at a price.
- Concretely, while the normalized training data falls in range $[a, b]$ (as proved on pages 58 and 59), the normalized validation and test data may fall out of the range.
- This is because $\mathbf{v}_{\text{val}}$ and $\mathbf{v}_{\text{test}}$ in eq. (4) could be smaller than $\min(\mathbf{v}_{\text{train}})$ (so that $\mathbf{v}_{\text{mms\_val}}$ and $\mathbf{v}_{\text{mms\_test}}$ will be smaller than a) or larger than $\max(\mathbf{v}_{\text{train}})$ (so that $\mathbf{v}_{\text{mms\_val}}$ and $\mathbf{v}_{\text{mms\_test}}$ will be larger than b):

$$\begin{aligned}
\mathbf{v}_{\text{mms\_val}} &= \frac{\mathbf{v}_{\text{val}} - \min(\mathbf{v}_{\text{train}})}{\max(\mathbf{v}_{\text{train}}) - \min(\mathbf{v}_{\text{train}})} \times (b - a) + a, \\
\mathbf{v}_{\text{mms\_test}} &= \frac{\mathbf{v}_{\text{test}} - \min(\mathbf{v}_{\text{train}})}{\max(\mathbf{v}_{\text{train}}) - \min(\mathbf{v}_{\text{train}})} \times (b - a) + a.
\end{aligned} \tag{4}$$

⚠ Caveat

- When using the $\min$ and $\max$ calculated from the training data to normalize the validation and test data, the normalized data may not necessarily fall in range $[a, b]$ (whereas the normalized training data always fall in the range).

Standardization & Min-Max Normalization

- **Q:** Why should not we use the statistics (e.g., μ, σ, $\min$ and $\max$) calculated on the validation / test data to scale (e.g., standardize or normalize) the validation / test data?

Standardization & Min-Max Normalization

- **Q:** Why should not we use the statistics (e.g., μ, σ, $\min$ and $\max$) calculated on the validation / test data to scale (e.g., standardize or normalize) the validation / test data?
- **A:** This is largely because the training data is usually much bigger than the validation / test data, so that the statistics calculated on the training data could be much more accurate (much closer to the statistics of the population).
- **Q:** Why should not we use the statistics calculated on the combined training, validation and test data to scale the training / validation / test data?

Standardization & Min-Max Normalization

- **Q:** Why should not we use the statistics (e.g., μ, σ, $\min$ and $\max$) calculated on the validation / test data to scale (e.g., standardize or normalize) the validation / test data?
- **A:** This is largely because the training data is usually much bigger than the validation / test data, so that the statistics calculated on the training data could be much more accurate (much closer to the statistics of the population).
- **Q:** Why should not we use the statistics calculated on the combined training, validation and test data to scale the training / validation / test data?
- **A:** This is because then the scaled training data will have information of the validation / test data (a problem usually referred to as *Data Leakage*), so that a model's performance on validation / test data will be overestimated.

Takeaway

- When scaling the data we should:

1. calculate the statistics on the training data
2. use the calculated statistics to scale the training / validation / test data

Standardization & Min-Max Normalization

- The code below shows how to scale the data:
  - see /p2_c1_data_preprocessing/code_example/regression:
    1. cells 50 to 55
  - see /p2_c1_data_preprocessing/code_example/classification:
    1. cells 50 to 53
  - the code for scaling the data are different in regression and classification:
    - in regression, we scale both the features and target
    - in classification, we only scale the features (since the classes are simply identifiers)

Good practice

- For regression, we scale both the features and target.
- For classification, we only scale the features.

Standardization VS Min-Max Normalization

- **Q:** Since we can use both Standardization and Min-Max Normalization to scale the data, which one shall we use?

Standardization VS Min-Max Normalization

- **Q:** Since we can use both Standardization and Min-Max Normalization to scale the data, which one shall we use?
- **A:** If we want all scaled variables in the same range, we should use normalization:
  - normalization will transform all the variables in the training data into range $[a, b]$ (as proved on pages 58 and 59)
  - on the other hand, while standardization could transform the variables into similar ranges, the ranges may still be different
- **A:** If the data have many outliers, we should use standardization:
  - normalization is more sensitive to outliers, since the outliers (which often take the $\min$ or $\max$ value) will be directly used in eqs. (3) and (4)
  - on the other hand, standardization is more robust to outliers, since the outliers will only be indirectly used (through μ and σ) in eqs. (1) and (2)

Good practice

- If we want all scaled variables in the same range, we should use normalization.
- If the data have many outliers, we should use standardization.

Step 10: Handling Class Imbalance

Table 2: The first 5 features and target (diagnosis) of Breast Cancer Wisconsin dataset.

id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean
842302	M	17.99	10.38	122.80	1001.0
842517	M	20.57	17.77	132.90	1326.0
84300903	M	19.69	21.25	130.00	1203.0
84348301	M	11.42	20.38	77.58	386.1
84358402	M	20.29	14.34	135.10	1297.0

- A key challenge in classification is *Class Imbalance*, where the number of samples for one class is higher than the number for the other classes.
- We usually call the class with more samples the *Majority Class*, whereas the class with fewer samples the *Minority Class*.
- Take the target, diagnosis, in table 2 for example:
  - majority class: benign (with 357 or 63% samples)
  - minority class: malignant (with 212 or 37% samples)

Step 10: Handling Class Imbalance

- The problem of class imbalance is that, when training a model from imbalanced data such as Breast Cancer Wisconsin (table 2), the model will have a bias against the minority class (malignant):
  - that is, even before looking at any clinical or pathological status of a patient, the model already decides that it is less likely (or, in the extreme case, impossible) for the patient to have malignant cancer cells (we will discuss the mathematical explanation in /p2_c2_s3_logistic_regression)
  - however, wrongly predicting cancer cells as benign when they are actually malignant could delay the treatment, which has profound implications
- Generally, there are two ways to handle class imbalance:
  - Cost-based method
  - Sampling-based method

⚠ **Caveat**

- When training a model from imbalanced data, the model will have a bias against the minority class.
- This problem has profound implications in healthcare, where the minority class is usually the Disease class whereas the majority class usually the Non-disease class.

Cost-based Method

- The idea of cost-based method is that, instead of giving equal weight to all the classes, we assign higher weight to minority class but lower weight to majority class.
- By doing so, predicting the minority classes incorrectly will incur a higher cost than predicting the majority class incorrectly.
- This could somehow alleviate the bias against the minority class.
- However, since assigning higher weight to the minority class is similar to duplicating samples of the class, this could lead to overfitting (we will discuss this in /p2_c2_s2_training_shallow_models), particularly when samples of the minority class are usually not separable from samples of the majority class.

Class Weight Hyperparameter

- The simplest way to use cost-based method is setting the `class_weight` hyperparameter (supported by many sklearn models) as ‘`balanced`’ (we will discuss this in /p2_c2_s3_logistic_regression and /p2_c2_s5_tree_based_models).
- In the current version of sklearn (0.24), this assigns the weight to class as the inverse frequency of the class, divided by the number of unique classes:

$$\underbrace{\frac{\text{number of samples in the data}}{\text{number of samples of the class}}}_{\text{inverse class frequency}} \times \underbrace{\frac{1}{\text{number of unique classes}}}_{\text{scaler}}. \tag{5}$$

Good practice

- If a model has hyperparameter `class_weight`, it is recommended to set the hyperparameter as ‘`balanced`’, so as to use cost-based method to address class imbalance.

Sampling-based Method

- The idea of sampling-based method is sampling the data in such a way that all the classes eventually have the same (or similar) number of samples (so that the imbalanced data becomes balanced).
- Based on how we sample the data, we can generally divide sampling-based method into three categories
  - *Oversampling*: adding samples of the minority classes
  - *Undersampling*: removing samples of the majority class
  - combination of oversampling and undersampling
- If computational cost is not a top concern, we usually prefer oversampling since the undersampled data could be too small to be useful for training.

Good practice

- We usually prefer oversampling to undersampling.

The Pipeline of Oversampling

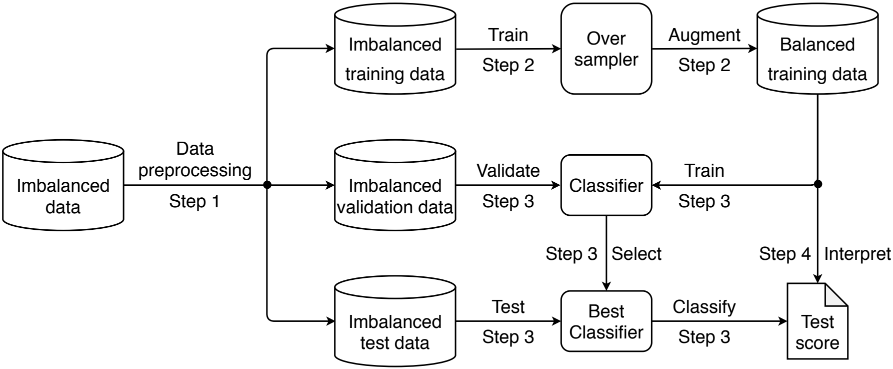

Figure 3: The pipeline of oversampling.

The Pipeline of Oversampling

- Fig. 3 shows the pipeline of oversampling:
  1. step 1: we preprocess the imbalanced data into imbalanced training, validation and test data
  2. step 2: we feed the imbalanced training data to an oversampling method (a.k.a., oversampler), which arguments the minority class and generates the balanced training data
  3. step 3: we
     1. feed the balanced training data and imbalanced validation data to a classifier to train the classifier and fine-tune its hyperparameters (see /p2_c2_s2_training_shallow_models)
     2. select the best classifier that corresponds to the best hyperparameter setting (which leads to the highest classification performance on the validation data, see /p2_c2_s2_training_shallow_models)
     3. feed the test data to the best classifier to generate the test score (classification performance on the test data)
  4. step 4: we use the balanced training data (augmented in step 2) to interpret the results

Traditional Oversampling Methods

- Among all the oversampling methods, the simplest one is Random Oversampler (RO), which duplicates the minority classes data (that is, the newly added data are exact replications of the original data).
- Another line of work, with Synthetic Minority Over-sampling Technique (SMOTE) [Chawla et al., 2002] being the most popular one, uses the average of a cluster of similar samples of the minority class to generate new samples for the class.
- The code below shows how to use oversampling to augment the minority class (so as to produce balanced data):
  - see /p2_c1_data_preprocessing/code_example/classification:
    1. cells 54 to 63

Problem of Traditional Oversampling Methods

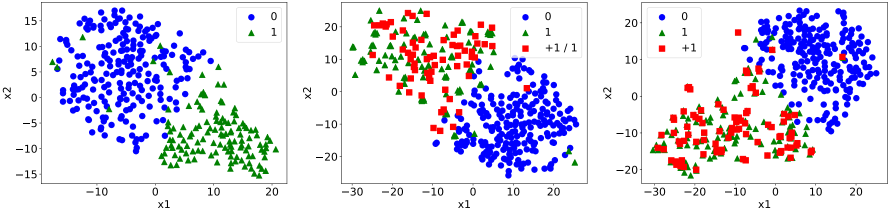

Figure 4: The scatter plot (using T-SNE) of imbalanced training data in Breast Cancer Wisconsin (left) and the training data augmented by RO (middle) and SMOTE (right). Here, 0, 1 and +1 represent the majority, minority and augmented minority class.

- Both RO and SMOTE could generate minority class samples (red squares in fig. 4) that are not separable from majority class samples (blue dots).
- When training a model on data with many such indistinguishable samples, the model will become excessively complex so as to separate these samples.
- This will lead to overfitting (we will discuss this in /p2_c2_s2_training_shallow_models), where the trained model is too complex to generalize well in reality.
- As a result, the model trained on the augmented (balanced) data may not be better (or may be even worse) than the model trained on the original (imbalanced) data, which is the opposite of the goal of data augmentation (to improve model performance).

Advanced Oversampling Methods

- Recently, methods based on *Deep Generative Models* have been proposed to augment the minority class:
  - [Mullick et al., 2019] extends *Generative Adversarial Networks* (GANs, we will discuss this in /p3_c3_s1_deep_generative_models) by:
    - adding a classifier to GANs so that the generated samples will be close to the decision boundary
    - using the discriminator and a convex generator so that the generated samples will follow the distribution of the original samples

Step 11: Feature Selection & Dimensionality Reduction

- One challenge in machine learning is the *Curse of Dimensionality* (CoD), which refers to problems that do not exist in low-dimensional data but arise in high-dimensional data.
- When feeding data with high dimensions (i.e., a large number of features) to some models:
  - either we cannot even train the models due to high computational cost
  - or the trained models will be much less accurate
- One of the models that suffer the most from CoD is *K-Nearest Neighbors* (see ESL: Chap 2).
- Two kinds of methods have been widely used to address CoD:
  - *Feature Selection*: select k features that have the highest predictive power (see /p2_c2_s5_tree_based_models)
  - *Dimensionality Reduction*: transform the data into k dimensions that capture the variance of the data the most (see /p2_c3_unsupervised_learning)

Proof of Range $[a, b]$: Page 42

- In eq. (3)

$$\mathbf{v}_{\text{mms\_train}} = \frac{\mathbf{v}_{\text{train}} - \min(\mathbf{v}_{\text{train}})}{\max(\mathbf{v}_{\text{train}}) - \min(\mathbf{v}_{\text{train}})} \times (b - a) + a, \tag{3}$$

  the ratio

$$\frac{\mathbf{v}_{\text{train}} - \min(\mathbf{v}_{\text{train}})}{\max(\mathbf{v}_{\text{train}}) - \min(\mathbf{v}_{\text{train}})} \tag{6}$$

  is strictly increasing:
    - the ratio takes the lowest value when $\mathbf{v}_{\text{train}}$ takes the lowest value
    - the ratio takes the highest value when $\mathbf{v}_{\text{train}}$ takes the highest value
- Since the lowest value of $\mathbf{v}_{\text{train}}$ is $\min(\mathbf{v}_{\text{train}})$, the lowest value of the ratio is 0.
- Conversely, since the highest value of $\mathbf{v}_{\text{train}}$ is $\max(\mathbf{v}_{\text{train}})$, the highest value of the ratio is 1.
- That is, the ratio falls in range $[0, 1]$.

Proof of Range $[a, b]$: Page 42

- Moreover, eq. (3)

$$\mathbf{v}_{\text{mms\_train}} = \frac{\mathbf{v}_{\text{train}} - \min(\mathbf{v}_{\text{train}})}{\max(\mathbf{v}_{\text{train}}) - \min(\mathbf{v}_{\text{train}})} \times (b - a) + a, \tag{3}$$

  is strictly increasing:
  - $\mathbf{v}_{\text{mms\_train}}$ takes the lowest value when the ratio takes the lowest value
  - $\mathbf{v}_{\text{mms\_train}}$ takes the highest value when the ratio takes the highest value
- Since the lowest value of the ratio is 0 (as discussed earlier), the lowest value of $\mathbf{v}_{\text{mms\_train}}$ is a.
- Conversely, since the highest value of the ratio is 1, the highest value of $\mathbf{v}_{\text{mms\_train}}$ is b.
- That is, $\mathbf{v}_{\text{mms\_train}}$ falls in range $[a, b]$, which proves the claim on page 42. □

Bibliography I

Chawla, N. V., Bowyer, K. W., Hall, L. O., and Kegelmeyer, W. P. (2002).
SMOTE: Synthetic Minority Over-sampling Technique.
Journal of Artificial Intelligence Research, 16:321–357.

Li, S. C. X., Jiang, B., and Marlin, B. (2019).
Misgan: Learning from Incomplete Data with Generative Adversarial Networks.
In *ICLR*.

Mullick, S. S., Datta, S., and Das, S. (2019).
Generative Adversarial Minority Oversampling.
In *ICCV*, pages 1695–1704.

Yoon, J., Jordon, J., and Van Der Schaar, M. (2018).
Gain: Missing Data Imputation using Generative Adversarial Nets.
In *ICML*.