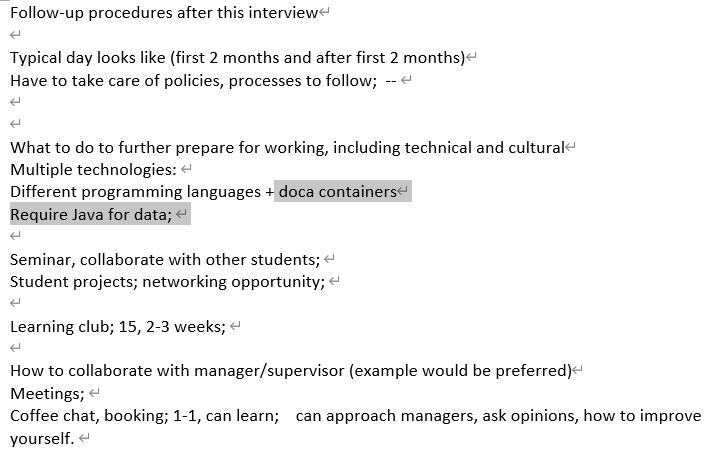
<https://scikit-learn.org/stable/user_guide.html>

all references are from this site, with titles and simulation code inside this file.

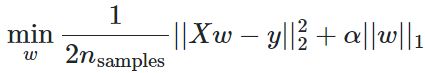
**WARNING: ALL “fit” METHOD SEEM TO HAVE WRONG INTERPRETATION;**

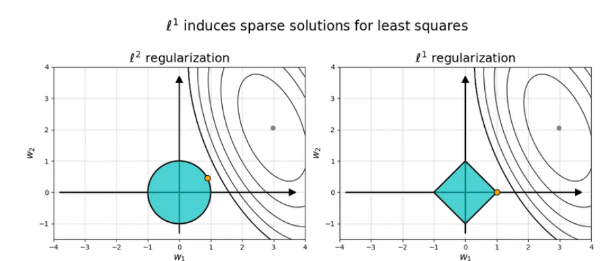
**NEED TO MODIFY**

##### Supervised Learning

###### Linear Model:

Key points:

* Least Squares & ordinary linear regression
* Ridge regression and L2 regularization of coefficient: 
* Lasso regression and L1 regularization of coefficient: 



* Difference between ridge regression and lasso regression:

1: the norm adopted is different;

2: realizing that Lasso regression’s norm could lead to ABSOLUTE zero coefficients, thus lead to sparse parameters and reduce dependence on some input features.

However Ridge regression could preserve all parameters (none of them would be zero), so that all features, when required, will be somehow preserved.

<https://satishkumarmoparthi.medium.com/why-l1-norm-creates-sparsity-compared-with-l2-norm-3c6fa9c607f4> Idea why L1 leads to zero coefficients.

* Linear classifiers:

Linear classifiers basically first convert targets into {-1, 1} and then treat the problem as a regression problem, using corresponding regression techniques.

 Linear\_model.RidgeClassifier()

Hinge loss is commonly used for linear classifier training; formula see image right;

“t” is the “desired target”; “y” is the “raw output” of classification model;

* N\_jobs: some methods includes this parameter, for utilizing more CPU cores to speed up fitting and prediction operations.

Code:

From sklearn import linear\_model

# below initializes model

Model = linear\_model.LinearRegression(fit\_intercept=True, positive=False,) # models shall be initialized

Model = linear\_model.Ridge(alpha=1.0, fit\_intercept=True, positive=False) # this initializes a ridge regression model.

Model = linear\_model.Lasso(“same parameters as Ridge()”) # this initializes a lasso regression model.

# alpha: the regularization strength; positive: when True, will restrict model parameters to be all positive.

# note: if data is required to be normalized, use normalization methods (refer to data processing part) to standardize/normalize data before fitting;

# below fits model, prints trained weights and how to predict new data.

Model.fit([x\_data\_points], [y\_target]) # x\_data and y\_target should have same

# outer\_dimension.

Print(Model.coef\_) # this prints the coefficient as a VECTOR

Print(model.predict([data\_points]))

# these methods are callable among all linear regression models;

# below are classification models

Model = linear\_model.LogisticRegression(penalty=”l2”, fit\_intercept=True, )

Confidence\_score = Model.decision\_function(X) # gives confidence of making each prediction; this function is common among linear classifiers;

Model.fit(X)

Result = model.predict(X)

Probability = model.predict\_proba(X) # predicts the probability of belonging to each class, logistic regression only

Log\_prob = model.predict\_log\_proba(X) # gives log probability of each sample belonging to each class, logistic regression only

# recall linear classifiers typically convert target into {-1, 1} and do regression.

Model = linear\_model.RidgeClassifier(alpha=1.0, fit\_intercept=True, positive=False)

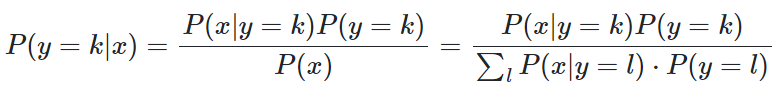
Model.decision\_function(X)

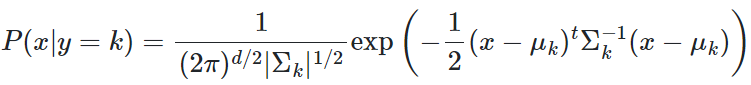
Model.fit(X)

Model.predict(X)

Model = linear\_model.SGDClassifier()

###### Discriminant Analysis:

Key points:

* Linear discriminant analysis; boundary is always linear
* Quadratic discriminant analysis; boundary can be non-linear
* Bayesian posterior distribution and class classification:
* Multi-dimensional gaussian distribution and class formation
* Model fitting: taking logarithm of posterior distribution and maximize using gradient descent.

Code:

From sklearn.discriminant\_analysis import LinearDiscriminantAnalysis

From sklearn.discriminant\_analysis import QuadraticDiscriminantAnalysis

# below initializes data points and corresponding tagets

X = np.array([points\_to\_classify]) # this array is all data points awaiting to classify

Y = np.array([targets) # this array must have same OUTER dimension with “X”

# below initializes model;

Model = LinearDiscriminantAnalysis(solver=’svd’)

# parameter for linear discriminant analysis:

# solver: ‘svd’: singular value decomposition, default

# ‘lsqr’: least squares solution

# ‘eigen’: eigenvalue decomposition

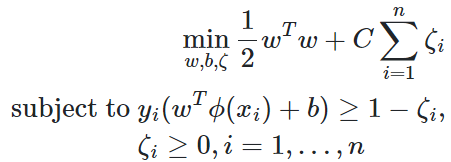
Model = QuadraticDiscriminantAnalysis()

# below shows how to fit and use model;

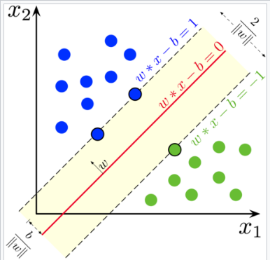
Model.fit(X, Y)

Print(Model.predict([data\_points]))

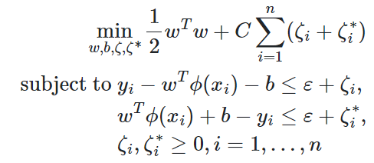
###### Support Vector Machines:

Key points:

* Effective for high dimensional spaces, when d>n
* Linear programming problem resemblence for classification;

(\phi is identity function, each \zeta\_{i} is the distance of each data point to the boundary)

Intuition: maximizing margin is the same as minimizing the norm of “w”;

Margin is shown in right graph: “b/||w||”.

* Linear programming problem for SVR (R for regression):

Problem is shown right.

Code:

From sklearn import svm

# initialize data

X = [data points]

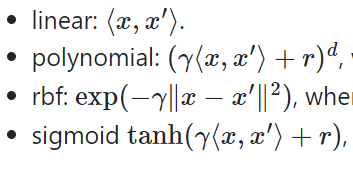
Y\_class = target\_vector # Y must have same outer dimension as X;

# classification target should contain either strings or integers

Y\_regres = target\_vector # Y must have the same outer diemnsion as X;

# below initializes models

Model = svm.SVC() # support vector classifier

Model = svm.SVR() # support vector regression

# parameter for “kernel”:

# kernel=’linear’/’rbf’/’plynomial’/’sigmoid’;

# kernel is the parameterization of boundary-line;

# below fits model and predicts target

Model.fit(X, Y\_)

Model.predict([data\_point])

# below is parameters of the CLASSIFICATION model

Model.support\_vectors\_ # this gives data points satisfying formula wx-b=1 and wx-b=-1

Model.support\_ # gives indices of support vectors

Clf.n\_support\_ # number of support vector for each class

###### Stochastic Gradient Descent:

Key points:

Code:

###### Gaussian Processes:

Allows for probabilistic analysis, such as computing confidence interval;

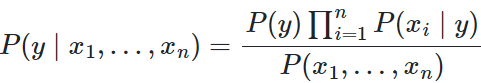
Might be ineffective in high dimension;

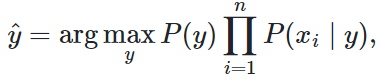
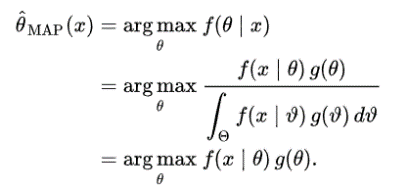
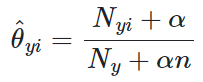
Key points:

* Regression:
* Classification:

Code:

###### Naïve Bayes:

Key points:

* Naïve assumption: each feature’s presence is conditionally independent of each other
* Can use MAP estimator to predict posterior distribution
* Gaussian Naïve Bayes: P(x|y) is Gaussian;
* Multinomial Naïve Bayes: a vector of probability(theta), each has the form: 

Code:

From sklearn.naive\_bayes import GaussianNB, MultinomialNB

X = …

Y = …

Model = GaussianNB()

Model = MultinomialNB()

Predict\_result = Model.fit(X, Y).predict([data\_points])

# below are some common attributes of gaussian model:

Model.class\_count\_

Model.class\_prior # reveals probability of each class

Model.n\_faetures\_in\_: # reveals number of features

# below are common attributes for multinomial model:

Model.class\_count\_ # reveals number of classes for this model

Model.class\_log\_prior # reveals log probability for each class

Model.feature\_count # shape (n\_classes, n\_features)

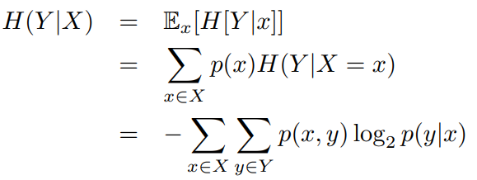
###### Decision Trees:

Advantages: easy to understand, easy to construct, wide application;

Disadvantages: can be over complex when constructing; unstable; can be highly biased;

Key points:

* Classifier;
* Regressor;
* Complexity: construction of tree requires checking all samples with features, and there are “tree-height” many times.
* Entropy related:

Entropy and information gain is one criterion for dividing tree;

Code:

From sklearn import tree

Model = tree.DecisionTreeClassifier(criterion=, max\_depth=, max\_features=,max\_leaf\_nodes=,)

Model = tree.DecisionTreeRegressor(criterion=, max\_depth=, max\_featuers=, max\_leaf\_nodes=,)

Model = Model.fit(X, Y) # looks like decision tree models’ fit method returns a new model

Model.predict([data\_])

# classification decision tree visualization:

tree.plot\_tree(Model)

###### Multi Layer Perceptron/Neural Network

Can virtually predict all non linear functions

But sensitive to feature and hyperparameter tuning and can converge to local minimum

Key points:

* Classifier;
* Regressor;
* Require data scaling due to sensitivity (normalize/standardize/restrict to a range)

Code:

from sklearn.neural\_network import MLPClassifier

from sklearn.neural\_network import MLPRegressor

model = MLPClassifier(hidden\_layer\_sizes=, activation=, solver=, learning\_rate=’constant’, early\_stopping=, )

model = MLPRegressor(hidden\_layer\_sizes=, activation=, solver=, learning\_rate=’constant’, early\_stopping=,)

model.fit(X, Y)

model.predict(X\_test, Y\_test)

##### Unsupervised Learning:

###### Gaussian Mixture Model:

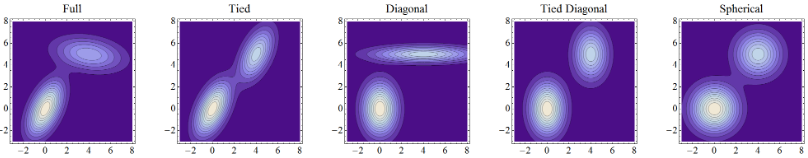
Fastest algorithm for leaning mixture models;

Require large data amount/apply regularization on covariance matrix

Key points:

* Expectation Maximization /EM:

Idea: randomly initialize several distributions (usually gaussian is adopted, with various initial mean and varaince) -> calculate each data point and assign them a class with highest probability -> update all classes’ distribution based on newly assigned data sets -> go back to step 2 and iteratively update

* Bayesian Gaussian Mixture: allows auto-identifying number of clusters required

Code:

From sklearn.mixture import GaussianMixture

From sklearn.mixture import BayesianGaussianMixture

Model = GaussianMixture(n\_components=, covariance\_type=, init\_params=, )

# also supports weight and mean initialization, & random\_state for reproducibility

# init\_params is by default, k-means;

# n\_components sets the number of groups/clusters

Model = BayesianGaussianMixture(n\_components=, covariance\_type=, init\_params=, )

Model.fit(X) # as this is unsupervised learning method, no target shall be provided

Model.predict(X\_predict)

# below are attributes for model:

Model.weights\_

Model.means\_

Model.covariances\_

Modle.n\_features\_in\_

###### Clustering(K-means):

* There are a variety of clustering algorithms, but K-means seem to be the most common one.

Key points:

Code:

From sklearn.cluster import KMeans

Model = KMeans(n\_clusters=8, init=,)

# n\_cluster is 8 by default!!! Might require changing when using;

Model.fit(X)

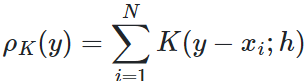
Model.predict(X\_predict)

###### Covaraince estimation:

Key points:

Code:

###### Kernel Density Estimation

Key points:

* Kernel: “K” is the kernel function, constrained by bandwidth “h”; each “x” is the point

in dataset

Code:

From sklearn.neighbors import KernelDensity

Kde = KernelDensity(bandwidth=, algorithm={tree\_algorithm}, kernel={‘gaussian’ as default}, )

Kde.fit(X)

Kde.score(X\_predict) # this provides the LOG-likelihood of total X\_predict in the model;

Kde.score\_sample(X\_predict) # this provides the log-likelohood of each element in X\_predict.

##### Data processing:

###### Feature Extraction:

Package: sklearn.feature\_extraction;

Dictvectorizer:

Key:

* Used for loading data from a “feature-value mapping like” objects in python (dict, etc.)
* Resulting transformation can be directly fitted by sklearn models.
* Strings as features/values: will use one-hot key, and counting method for each possible string value’s occurrences.
* To better understand one-hot-encoder:

<https://datagy.io/sklearn-one-hot-encode/>

code:

from sklearn.feature\_extraction import DictVectorizer

vectorizer = DictVectorizer(sparse=True, separator=”=”, sort=True)

# sparse: whether the transform will provide “scipy.sparse matrix”;

# separator:

D = [{‘a’: 1, ‘b’: 2}, {‘a’: 3, ‘c’: 1}]

X = vectorizer.fit\_transform(D) # learn a list of feature-value mapping, and transform result into

# ARRAY;

X = vectorizer.fit(D) # learn a mapping of feature-value;

X\_prime = vectorizer.transform(X) # transform resulting mapping into an array

# the advantage of using “fit\_transform” is, it saves space, which is required to store “X” when

# broken down.

D\_prime = vectorizer.inverse\_transform(X\_prime) # transform array into a feature-value dict

# mapping;

vectorizer.get\_feature\_names\_out() # get output feature names for transformation

vectorizer.get\_params() # get parameters for this estimator.

FeatureHasher:

Keys:

* Fast speed and low memory usage, due to applying a hash function instead of constructing a hashtable like other vectorizers(Dict and Count);
* Resulting in problem of inspectability, and no way of making inverse transformations.
* Can cause collisions when using hash function; a signed hash function can reduce the effect;

Code:

From sklearn.feature\_extraction import FeatureHasher

Hasher = Featurehasher(n\_features=2\*\*20, input\_type=’dict’, alternate\_sign=True)

# n\_features: always recommended to provide, for space reduction;

# input\_type: can be chosen from: ‘dict’({key: value}), ‘pair’(tuple(key, value)), ‘str’;

# alternate sign: whether to use signed hash function;

D = [{‘a’: 1, ‘b’: 2}, {‘a’: 3, ‘c’: 1}]

X = hasher.transform(D).toarray() # “fit” method doesn’t do anything for FeatureHasher.

Hasher.get\_params()

Text feature:

Keys:

* Text feature processing involves 3 steps: tokenizing by separation -> counting occurrences -> normalizing and reweighing (especially for tf-idf)
* Sparse matrix is frequently used to reduce storge space and accelerate arithmetic operations (scipy.sparse)
* CountVectorizer: produces a sparse matrix recording the counting of each term’s occurrence in a given text (from a batch of texts)
* TF-IDF: the weighing of each word is calculated as: “term frequency” times “inverse document frequency”;

Allows putting more focus on less-occurred words

IDF is calculated by the formula shown right; “df(t)” is the number of documents containing the term “t”; “n” is total number of documents;

* Word analysis & character analysis:

Character analysis makes model more robust against misspelling and word-segments analysis;

* To resolving the problem of creating hashtables for quick accessing in some vectorizers, “HashingVectors”, adopted from “FeatureHashing”, helps by using hashing function to replace hashtable construction, for direct index accessing of sparse matrix;

Code:

# Part 1: CountVectorizer;

From sklearn.feature\_extraction.text import CountVectorizer

Cv = CountVectorizer(input=’content’, lowercase=’True’, analyzer=”word”, preprocessor=None, tokenizer=None, stop\_words=None, ngram\_range=(1, 1), vocabulary=None, max\_df=1.0, min\_df=1.0, max\_features=None, …)

# analyzer: can be “word”, “char”, “char\_wb”, corresponding to ‘whole words’, ‘single letter’,

# and ‘characters with word boundaries’;

# preprocessor & tokenizer: if not None, must be a function; if None, will use built-in functions;

# stop\_words: list of words which shouldn’t be tokenized , usually due to too frequent; (max\_df

# has similar effect)

# ngram\_range: a tuple of range (min\_ngrams, max\_ngrams);

# vocabulary: a mapping from terms to indices, or an iterable of terms; when “None”, will

# construct from “input”;

# max\_df: when constructing “vocabulary”, will ignore terms with a frequency higher than this

# float value, between 0 and 1; when “vocabulary” is not None, this term is not effective.

Text = [“text one”, “text two”]

Fitted = cv.fit(text)

data = cv.transform(fitted)

data = cv.fit\_transform(text)

inverted = cv.inverse\_transform(data)

cv.build\_preprocessor() # returns the function used as preprocessor

cv.build\_tokenizer() # returns the function used as tokenizer

cv.get\_feature\_names()

cv.get\_stop\_words()

# Part 2: tf-idf tokenizer

From sklearn.feature\_extraction.text import TfidfTransformer, TfidfVectorizer

Tt = TfidfTransformer(norm=”l2”, use\_idf=True, smooth\_idf=True, sublinear\_tf=False)

# norm: the final vectors will be normalized by either “l2” or “l1”;

# smooth\_idf: if False, will not have extra “1”s when calculating “idf”

# sublinear\_tf: if True, will replace “tf” as “1 + log(tf)”, where “tf” is “term frequency”;

Tv = TfidfVectorizer(all parameters from CountVectorizer and TfidfTransformer)

# realizing this vectorizer is a sequential vectorizer, first perform count vectorizing, then

# transform using “tf-idf” technique

Fitted = Tt.fit(data) # recall “data” is the result of CountVectorizer’s fitted sparse matrix;

Fitted = tv.fit(text) # here “text” is the input text strings;

Tf\_data = tt.transform(fitted)

Tf\_data = tt.fit\_transform(data)

Tf\_data = tv.fit\_transform(text)

Tt.get\_params()

Tt.get\_feature\_names\_out()

Tt.set\_params()

# tv contains all methods from CountVectorizer and TfidfTransformer;

# Part 3: HashingVectorizer

From sklearn.feature\_extraction.text import HashingVectorizer

Hv = HashingVectorizer(input=”content”, lowercase=True, preprocessor=None, tokenizer=None, stop\_words=None, ngram\_range=(1, 1), analyzer=’word’, n\_features=2\*\*20, binary=False, norm=”l2”, alternate\_sign=True, …)

# realizing most parameters are directly inherited from previous vectorizers and feature-hashing

Hv.build\_analyzer()

Hv.build\_preprocessor()

Hv.build\_tokenizer()

Fitted = hv.fit(text)

Transformed=hv.transform(fitted)

Transformed=hv.fit\_tramsform(text)

Hv.get\_params()

Hv.get\_stop\_words()

###### Data preprocessing:

Package: sklearn.preprocessing;

Standardization:

**Keys:**

* Most machine learning algorithms assume input data is distributed with zero mean and unit variance; (idea of standardization)
* Standardized\_data = (data - mean) / std
* MinMaxScaler: formula: scaled\_data = (data - min)/(max-min)

Data will be in range [0, 1] after scaling

* MaxAbsScaler: formula: scaled\_data = (data / abs(max\_in\_feature))

The formula shows zeros will be preserved after this scaling.

Quite useful for scaling sparsed data (due to zero-preservation)

* RobustScaler is useful for scaling data with outliers;

When doing scaling: will only select those data excluding first and last quantiles(or customed intervals), to eliminate the effect of outliers.

**Code:**

From sklearn.preprocessing import StandardScaler

Data = [[0, 0], [1, 1]]

Scaler = StandardScaler(copy=True, with\_mean=True, with\_std=True)

# copy: if false, will do “in-place” scaling; not returning a new array;

Scaler.fit(data) # acquire mean and std for future scaling

Modified\_data = scaler.transform(data)

Modified\_data = scaler.fit\_transform(data)

Data\_invert = scaler.inverse\_transform(modified\_data)

Scaler.get\_params()

From sklearn.preprocessing import MinMaxScaler

Min\_max = MinMaxScaler(feature\_range=(0, 1), copy=True, )

Min\_max.fit(data)

Min\_max.transform(data)

Scaled = Min\_max.fit\_transform(data)

Min\_max.get\_feature\_name\_out()

Min\_max.get\_params()

Min\_max.inverse\_transform(scaled)

From sklearn.preprocessing import MaxAbsScaler

Maxabs = MaxAbsScaler(copy=True)

Maxabs.fit(data)

Scaled\_data = Maxabs.transform(data)

Scaled\_data = Maxabs.fit\_transform(data)

Inverted = maxabs.inverse\_transform(scaled\_data)

Maxabs.get\_params()

Maxabs.get\_feature\_names\_out()

From sklearn.preprocessing import RobustScaler

Rs = RobustScaler(with\_centering=True, with\_scaling=True, quantile\_range=(25.0, 75.0), copy=True, unit\_variance=False)

# with\_centering: center the data before scaling; raise error when transforming sparse matrices;

# with\_scaling: controls scaling the data to interquantile range;

# quantile\_range: data in quantile (0, 25) and (75, 100) will be excluded;

# unit\_variance: will scale data to follow a variance of “1”;

Rs.fit(data)

Scaled = rs.transform(data)

Scaled=rs.fit\_transform(data)

Inverted = rs.inverse\_transform(scaled)

Rs.get\_feature\_names\_out()

Rs.get\_params()

Normalization:

Package: sklearn.preprocessing.normalize

Key:

* “Normalize” is a function for conveniently scaling input vectors into unit norms;

Code:

Import sklearn.preprocessing as prep

From sklearn.preprocessing import Normalizer

Normalized = prep.normalize(data, norm=’l2’, axis=1, copy=True, return\_norm=False)

# “data” should have the shape (n\_samples, n\_features)

# “axis”: if “1”: normalize each sample; if “0”, normalize each feature;

# “norm”: can adopt “l1”, “l2”, “max”

Norm = Normalizer(norm=”l2”, copy=True)

Norm.transform(data)

norm.get\_feature\_names\_out()

norm.get\_params()

Encoder:

* Features can be efficiently encoded as integers, for model estimation
* OrdinalEncoder: encodes input features as an integer array, using simple methods
* OneHotEncoder:

Example shown right demonstrates the effect of one-hot-encoder:

When the category is binary, one column is used; when more than 2, then for each feature, there will be a “category\*category” sized array for representing each category of the feature, using “1” for category matching and “0” for other values.

Code:

From sklearn.preprocessing import OrdinalEncoder, OneHotEncoder

Ordinal = OrdinalEncoder(categories=”auto”, handle\_unknown=’error’, unknown\_value=None, encoded\_missing\_value=np.nan)

# handle\_unknown: can take values “error”, “use\_encoded\_value”

# unknown\_value: used along with “handle\_unknown”, the value to use when encountering unknown feature; must be an “int” or “np.nan”

# encoded\_missing\_value: used for replacing “np.nan” from input data; must be “int” / “nan”

# operations can perform: fit(data), transform(data), fit\_transform(data), inverse\_transform(encoded), get\_faeture\_names\_out(), get\_params()

Onehot = OneHotEncoder(categories=’auto’, drop=None, sparse=True, handle\_unknown=”error”, min\_frequency=None, max\_categories=None)

# drop: “first”: drop the first category (first seen value) for each feature; “if\_binary”: if a feature

# has two categories, the first one will be removed; rest features remain intact; “array with size n\_features”

# specifies for each feature, which category to drop

# sparse: returns sparse matrix;

# min\_frequency: those categories with frequency less than “min\_frequency” won’t be encoded

# max\_categories: restrict the maximum of distinct values each feature can choose;

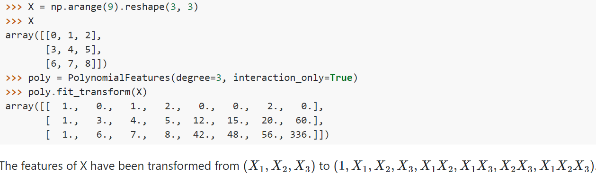
# methods can perform: fit(data), transform(data), fit\_transform(data), inverse\_transform(encoded\_data), get\_params(), get\_feature\_names\_out(),

Non-linear transformation:

Keys:

* **QuantileTransformer**: for mapping given data into either a uniform-distribution or a normal distribution, by returning quantile landmarks for each data, so that any value in between two landmarks would correspond to uniform/normal distribution’s interval;

This transformation can effectively deal with outliers, and

* **PowerTransformer**: for mapping given data distribution into approximated Gaussian distribution, for more friendly data processing with the help of Gaussian distribution’s properties (law of large numbers, easier least-squares, easy to draw conclusions, etc)
* **PolynomialFeature**: converting given features into polynomial features by applying a feature map;

This somehow allows linear models to partially fit non-linear data.

An example is shown in right image.

Code:

From sklearn.preprocessing import QuantileTransformer, PowerTransformer, PolynomialFeatures,

Qt = QuantileTransformer(n\_quantiles=1000, output\_distribution=’uniform’, copy=True, subsample=1e5, )

# output\_distribution: can take value “uniform”, “normal”;

# subsample: the maximum amount of data points sampled, for estimating quantiles;

# methods can perform: qt.fit(data), qt.transform(data), qt.fit\_transform(data), qt.inverse\_transform(transformed\_data), qt.get\_params(), qt.get\_feature\_names\_out(),

Pt = PowerTransformer(method=’yeo-johnson’, standardize=True, copy=True)

# method: require further exploration; can take value: ‘yeo-johnson’(works for positive and negative values), ‘box-cox’(works for positive values only)

# standardize: if “True”, will standardize resulting distribution into zeor mean, unit variance Gaussian distribution;

# functions include: pt.fit(data), pt.transform(data), pt.fit\_transform(data), pt.inverse\_transform(transformed\_data), pt.get\_params(), pt.get\_feature\_names\_out(),

Pf = PolynomialFeatures(degree=2, include\_bias=True, interaction\_only=False)

# interaction\_only: interaction features: terms containing power 2 or higher of same element will be excluded; (x2, x2y, x3y, … are excluded)

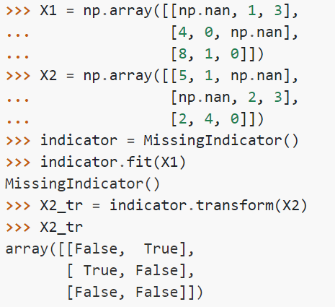
# degree: (min\_degree, max\_degree), if integer is given, min\_degree is 0 by default.

# methods can use: pf.fit(data), pf.transform(data), pf.fit\_transform(data), pf.get\_feature\_names\_out(), pf.get\_params(), pf.set\_params()

###### Imputing missing values

Package: sklearn.impute

Keys:

* SimpleImputer lists common techniques for filling missing values, such as using statistical indices(mean, median, mode), or with a specified value, or using an indicator for explicitly stating the value is missing, for the model to be aware when training.
* Ideas of applying KNN is implemented by KNNImputer; the idea is to find the nearest “k” many datapoints, and take those data’s “mean”(not median, not mode) value to fill the missing value
* IterativeImputer applies an alternative idea for filling missing values; this imputer determines the relationship between current missing feature with other features, and fills missing values based on the learned relationship. This idea is still experimental, but is worth exploration.
* MissingIndicator: an indicator that, when taking data and “transform”, outputs an BOOLEAN array indicating whether the corresponding index in “data” is missing or not. Can refer to examples provided in API.

Code:

From sklearn.impute import SimpleImputer, KNNImputer, MissingIndicator

Si = SimpleImputer(missing\_values=np.nan, strategy=’mean’, fill\_value=”None”, copy=True, add\_indicator=False)

# missing\_values: the value that will be treated as “missing”; can be “int float str, np.nan None, pandas.NA;

# strategy: can take value “mean”, “median”, “most\_frequent”(mode), “constant”

# fill\_value: when “strategy=’constant’”, “missing\_values” will be filled with “fill\_value”

# add\_indicator: when “True”, will add a MissingIndicator Object, for displaying whether the current element is a missing element;

# methods can perform: si.fit(data), si.transform(data), si.fit\_transform(data), si.inverse\_transform(imputed\_data), si.get\_params(), si.get\_feature\_names\_out(), si.set\_params()

Knni = KNNImputer(missing\_values=np.nan, n\_neighbors=5, weights=’uniform’, metric=’nan\_euclidean’, copy=True, add\_indicator=False)

# weights: can take value “uniform”, “distance”(closer points have a greater influence) or any callable function taking in an array of distances, and returning an array of weights;

# metric: distance metric for searching neighbors; can also be any function with input: (array\_1, array\_2, “missing\_values”), and with output a scalar distance.

# methods include: knni.fit(data), knni.transform(data), knni.fit\_transform(data), knni.inverse\_transform(), knni.get\_feature\_names\_out, knni.get\_params(), knni.set\_params()

Mi = MissingIndicator(missing\_values: np.nan, features=’missing-only’, sparse=’auto’, error\_on\_new=True)

# features: “missing-only”: only represent features with missing values; “all” regardless of containing missing values or not, every feature will be represented.

# sparse: ‘auto’: mask will be same type as input; “True”: mask will be a sparse matrix; “False”: will output numpy array

# error\_on\_new: if a feature during “fit” contains no missing values but contains missing values when “transform”, report an error;

# methods include: mi.fit(data), mi.transform(data), mi.fit\_transform(data), mi.get\_feature\_names\_out(), mi.get\_params(), mi.set\_params(), mi.inverse\_transform;

##### Model selection:

###### Cross\_validation:

Keys:

* Models can overfit; to prevent, tests are required to conduct, for choosing the best parameters, from several trained models and retrain. The technique is cross validation.
* Cross validation: randomly split data into several chunks, and selectively (with no repetition) choose one or more of those chunks for testing(rest for training), and calculate model performance each time training is complete.
* cross\_val\_score: gives calculations to evaluate performance of estimator;
* cross\_validate offers more information when outputting, and allows multiple metrics for evaluating.
* More information regarding “shuffling data groups” can be referred from API

code:

from sklearn.model\_selection import cross\_val\_score, cross\_validate

from sklearn import svm

import numpy as np

test\_model = svm.SVR()

# model doesn’t need to be supervised; (test\_target is not necessary)

test\_data = np.arange(n\_samples \* n\_features).reshape(n\_samples, n\_features)

test\_target = np.arange(n\_samples \* n\_outputs).reshape(n\_samples, n\_outputs)

# pay attention to the shapes of above testing data;

Result\_score = cross\_val\_score(estimator=test\_model, X=test\_data, Y=None/test\_target, scoring=None, cv=None…)

# scoring can be a callable for calculating score; if None, will use default way to calculate.

# X: the data will be used to fit “estimator”, and then scores are calculated by comparing with “Y”.

# cv: if “None”, will perform 5-fold cross validation; when assigned with an int, will perform “int” many;

# result\_score contains the final cross validation score for each fold (has length # of times run)

Result2 = cross\_validate(estimator=test\_model, X=test\_data, Y=test\_target, scoring=None, cv=None, …)

# result2 is a dictionary object, containing the following data:

Result2[“test\_score”] # score array for each fold;

Result2[“fit\_time”] # time spent for fitting the model in each fold;

Result2[“score\_time”] # time spent for scoring the model performance in each fold;

###### Hyperparameter tuning:

Keys:

* grid search is a common hyperparameter tuning technique, which takes all possible parameter configurations (which might require human input), and try out by one-line code.
* RandomizedSearchCV: this class allows searching for parameters from a probabilistic distribution, providing maximum number of resampling times; a probabilistic distribution is required
* HalvingGridSearchCV: this search technique gradually reduces number of parameters to set, while increase the number of samples used to set remaining hyperparameters. The increase of samples and decrease of unset\_hyperparameters are inverse\_propotional by a factor;

The number of introduction and reduction is EXPONENTIAL:

# This method is still under experiment,

Code:

From sklearn.model\_selection import GridSearchCV, HalvingGridSearchCV

Param\_grid={parameter1: [possible\_values], …}

# takes form a python dictionary, where keys are each parameter, and values are possible choices

Searchor = GridSearchCV(test\_model, param\_grid, refit=True, cv=None, …)

# refit: will allow data to be re-estimated after the best hyperparameters are found and set.

Searchor.fit(test\_data, test\_target)

Transformed = Searchor.transform(test\_data)

# only supported when “estimator” supports “transform” method, and “refit=True”

Original\_data = Searchor.inverse\_transform(transformed)

# only supported when “estimator” supports “inverse\_transform” method, and “refit=True”

Prediction = Searchor.predict(test\_data)

Log\_prob = Searchor.predict\_log\_proba(test\_data)

# all predictions are based on best found hyperparameters

Scores = Searchor.score(test\_data)

Params = Searchor.get\_params()

# HalvingGridSearchCV is still under experiment, so will only provide code “COMMENTED OUT”.

# includes all parameters for GridSearchCV, plus the following:

# factor: the propotional number of samples to increase, and the number of hyperparameters reamin unset to decrease

# resource=’n\_samples’: defines what to increase for each iteration. Can also set to ‘n\_iterations’ or ‘n\_estimators’.

# max\_resources=’auto’: can set to other integers, restricting “resource” defined above

# min\_resources=’exhaust’/’smallest’: can also set to any integer, restricting “resource” defined above.

# aggressive\_elimination=False: if “True”, when there aren’t sufficient “resources”, will start REUSING the previously used resources for continuous hyperparameter fitting procedures.