**Preprocessing with sklearn: a complete and comprehensive guide**

For aspiring data scientist it might sometimes be difficult to find their way through the forest of **preprocessing techniques**. *Sklearn*its **preprocessing library** forms a solid foundation to guide you through this important task in the data science pipeline. Although *Sklearn*a has pretty solid documentation, it often misses streamline and intuition between different concepts.



This article intends to be a complete guide on preprocessing with **sklearn v0.20.0**. It includes *all*utility functions and transformer classes available in sklearn, supplemented with some useful functions from other common libraries. On top of that, the article is structured in a **logical order** representing the order in which one should execute the transformations discussed.

The following subjects will be handled:

* Missing values
* Polynomial features
* Categorical features
* Numerical features
* Custom transformations
* Feature scaling
* Normalization

Note that step three and four can be performed interchangeable, since these transformations should be executed independently of each other.

**Missing values**

Handling missing values is an essential preprocessing task that can drastically deteriorate your model when not done with sufficient care. A few questions should come up when handling missing values:

*Do I have missing values? How are they expressed in the data? Should I withhold samples with missing values? Or should I replace them? If so, which values should they be replaced with?*

Before starting handling missing values it is important to **identify the missing values** and know with which value they are replaced. You should be able to find this out by combining the metadata information with exploratory analysis.

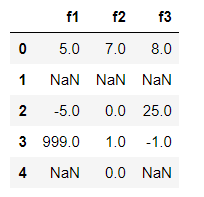
Once you know a bit more about the missing data you have to decide whether or not you want to keep entries with missing data. According to Chris Albon (*Machine Learning with Python Cookbook*), this decision should partially depend on **how random missing values are**.

If they are completely at random, they don’t give any extra information and can be omitted. On the other hand, if they’re not at random, the fact that a value is missing is itself information and can be expressed as an extra binary feature.

Also keep in mind that deleting a whole observation because it has one missing value, might be a poor decision and lead to information loss. Just like keeping a whole row of missing values because it has a meaningful missing value might not be your best move.

Let’s materialize this theory with some coding examples using sklearn’s *MissingIndicator*. To give our code some meaning, we’ll create a very small data set with three features and five samples. The data contains obvious missing values expressed as *not-a-number*or*999*.

import numpy as np  
import pandas as pdX = pd.DataFrame(  
 np.array([5,7,8, np.NaN, np.NaN, np.NaN, -5,  
 0,25,999,1,-1, np.NaN, 0, np.NaN])\  
 .reshape((5,3)))X.columns = ['f1', 'f2', 'f3'] #feature 1, feature 2, feature 3



Data set with three features and five samples

Take a quick look at the data so you know where the missing values are situated. Rows or columns with to many non-meaningful missing values can be deleted from you data with pandas’ *dropna*function. Let take a look at the most important parameters:

* *axis*: 0 for rows, 1 for columns
* *tresh*: the number of non-NaN’s not to drop a row or column
* inplace: update the frame

We update our dataset by deleting all the rows (*axis*=0) with *only*missing values. Note that in this case instead of setting *tresh*to 1, you can also set the *how*parameter to *‘all’*. As a result our second sample is dropped, since it only consist of missing values. Note that we reset the index and drop the old index column for future convenience.

X.dropna(axis=0, thresh=1, inplace=True)X.reset\_index(inplace=True)X.drop(['index'], axis=1, inplace=True)

Let’s also create some extra boolean features that tell us if a sample has a missing value for a certain feature. Start by importing the *MissingIndicator* from *sklearn.impute* (note that **version 0.20.0** is required (update with ‘*conda update scikit-learn*’)).

Unfortunately, the *MissingIndicator*does not support multiple types of missing values ([see this question on Stackoverflow](https://stackoverflow.com/questions/53002403/sklearn-impute-missingindicator-valueerror-input-contains-nan-infinity-or-a-va)). Hence why we have to convert the *999*values in our dataframe to *NaN*’s. Next, we create, fit and transform a MissingIndicator object that will detect all *NaN’s*in our data.

From this indicator, we can create a new dataframe with boolean values indicating if an instance has a missing value for a certain feature. But why do we only have two new columns while we had three original features? After the deletion of our second sample, *f2*did not have missing values anymore. If the *MissingIndicator*does not detect any missing values in a feature, it does not create a new feature from this feature.

We will add this new features later to our original data, for now we can store them in the *indicator*variable.

from sklearn.impute import MissingIndicatorX.replace({999.0 : np.NaN}, inplace=True)indicator = MissingIndicator(missing\_values=np.NaN)indicator = indicator.fit\_transform(X)indicator = pd.DataFrame(indicator, columns=['m1', 'm3'])

After deciding to keep (some of) your missing values and creating missing value indicators, the next question is if you should replace the missing values. Most learning algorithms perform poorly when missing values are expressed as *not a number* (*np.NaN*) and need some form of missing value imputation. Be aware that some libraries and algorithms, such as *XGBoost*, *can*handle missing values and impute these values automatically by learning.

**Imputing values**

For filling up missing values with common strategies, sklearn provides a *SimpleImputer*. The four main strategies are *mean*, *most\_frequent*, *median*and *constant*(don’t forget to set the *fill\_value*parameter**)**. In the example below we impute missing values for our dataframe X with the feature’s mean.

from sklearn.impute import SimpleImputerimp = SimpleImputer(missing\_values=np.nan, strategy='mean')imp.fit\_transform(X)

Note that the values returned are put into an *Numpy*array and we lose all the meta-information. Since all these strategies can be mimicked in *pandas*, we are going to use pandas *fillna*method to impute missing values. For ‘*mean*’ we can use the following code. This *pandas*implementation also provides options to fill forward (*ffill*) or fill backward (*bfill*), which are convenient when working with time series.

X.fillna(X.mean(), inplace=True)

Other popular ways to impute missing data are **clustering the data with the k-nearest neighbor** (KNN) **algorithm**or **interpolating**the values using a wide range of interpolation methods. Both techniques are not implemented in *sklearn*’s preprocessing library and won’t be discussed here.

**Polynomial features**

Creating polynomial features is a simple and common way of feature engineering that adds complexity to numeric input data by combining features.

Polynomial features are often created **when we want to include the notion that there exists a nonlinear relationship between the features and the target**.They are mostly used to add complexity to linear models with little features, or when we suspect the effect of one feature is dependent on another feature.

Before handling missing values, you need to decide if you want to use polynomial features or not. If you for example replace all the missing values by 0, all the cross-products using this feature will be 0. Moreover, if you don’t replace missing values (*NaN*), creating polynomial features will raise a value error in the *fit\_transform*phase, since the input should be finite.

In this respect, replacing missing values by the median or the mean seems to be a reasonable choice. Since I’m not completely sure about this, and can’t find any consistent information, [I asked this question on the data science StackExchange](https://datascience.stackexchange.com/questions/40003/handling-missing-values-to-optimize-polynomial-features).

*Sklearn*provides a *PolynomialFeatures*class to create polynomial features from scratch. The *degree*parameter determines the maximum degree of the polynomial. For example, when *degree*is set to two and X=x1, x2, the features created will be 1, x1, x2, x1², x1x2 and x2². The *interaction\_only*parameter let the function know we only want the interaction features, i.e. 1, x1, x2 and x1x2.

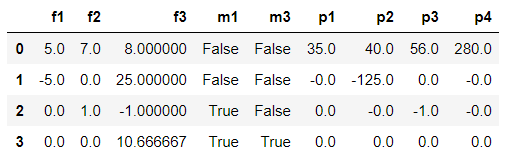
Here, we create polynomial features to the third degree and only interaction feature. As a result we get four new features: *f1.f2*, *f1.f3*, *f2.f3*and *f1.f2.f3*. Note that our original features are also included in the output and we slice off the new features to add to our data later.

from sklearn.preprocessing import PolynomialFeaturespoly = PolynomialFeatures(degree=3, interaction\_only=True)polynomials = pd.DataFrame(poly\  
 .fit\_transform(X),   
 columns=['0','1','2','3',   
 'p1', 'p2', 'p3', 'p4'])\  
 [['p1', 'p2', 'p3', 'p4']]

Just as with any other form of feature engineering, it is important to create polynomial features *before*doing any feature scaling.

Now, let’s concatenate our new missing indicator features and polynomial features to our data with pandas *concat*method.

X = pd.concat([X, indicator, polynomials], axis=1)



Dataframe with original features (f), missing value indicators (m) and polynomial features (p)

**Categorical features**

Munging categorical data is another essential process during data preprocessing. Unfortunately, *sklearn’s*machine learning library does not support handling categorical data. Even for tree-based models, it is necessary to **convert categorical features to a numerical representation**.

Before you start transforming your data, it is important to figure out if the feature you’re working on is ordinal (as opposed to nominal). **An ordinal feature** is best described as **a feature with natural, ordered categories and the distances between the categories is not known**.

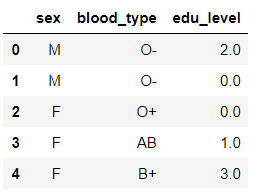
Once you know what type of categorical data you’re working on, you can pick a suiting transformation tool. In sklearn that will be a *OrdinalEncoder*for ordinal data, and a *OneHotEncoder*for nominal data.

Let’s consider a simple example to demonstrate how both classes are working. Create a dataframe with five entries and three features: *sex*, *blood type* and *education level*.

X = pd.DataFrame(  
 np.array(['M', 'O-', 'medium',  
 'M', 'O-', 'high',  
 'F', 'O+', 'high',  
 'F', 'AB', 'low',  
 'F', 'B+', np.NaN])  
 .reshape((5,3)))X.columns = ['sex', 'blood\_type', 'edu\_level']

Looking at the dataframe, you should notice education level is the only ordinal feature (it can be ordered and the distance between the categories is not known). We’ll start with encoding this feature with the *OrdinalEncoder*class. Import the class and create a new instance. Then update the education level feature by fitting and transforming the feature to the encoder. The result should look as below.

from sklearn.preprocessing import OrdinalEncoderencoder = OrdinalEncoder()X.edu\_level = encoder.fit\_transform(X.edu\_level.values.reshape(-1, 1))



Notice that we have a rather annoying issue here: **our missing value is encoded as a separate class (3.0)**. Looking thoroughly at the documentation reveals that there is no solution for this issue yet. A good sign is that the *sklearn*developers are [discussing the possibilities of implementing a suiting solution](https://github.com/scikit-learn/scikit-learn/issues/11996).

Another problem is that **the order of our data is not respected**. This can luckily be solved by passing an ordered list of unique values for the feature to the *categories*parameter.

encoder = OrdinalEncoder(categories=['low', 'medium', 'high'])

To resolve the first issue, we have to turn to pandas. The *factorize*method provides an alternative that can handle missing values and respects the order of our values. The first step is to convert the feature to an ordered pandas *Categorical*. Pass on a list of *categories*(including a category for missing values) and set the *ordered*parameter to *True*.

cat = pd.Categorical(X.edu\_level,   
 categories=['missing', 'low',   
 'medium', 'high'],   
 ordered=True)

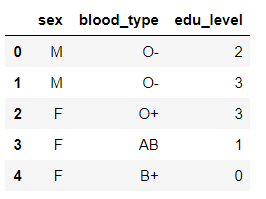
Replace the missing values by the *missing*category.

cat.fillna('missing')

Then, factorize the *Categorical*with the sort parameter set to *True*and assign the output to the *education level* feature.

labels, unique = pd.factorize(cat, sort=True)X.edu\_level = labels

The results are more satisfying this time as the data is numerical, still ordered and the missing values are replaced by 0. Note that replacing missing values with the smallest value might not always be the best choice. Other options are to **put it in the most common category or to put it in the category of the value in the middle when the feature is sorted**.



Let’s turn to the other two, nominal features now. Remember that we can’t replace these features by a number since this would imply the features have an order, which is untrue in case of sex or blood type.

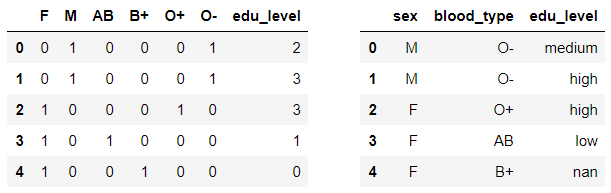
The most popular way to encode nominal features is **one-hot-encoding**. Essentially, **each categorical feature with *n* categories is transformed into *n* binary features**.

Let’s take a look at our example to make things clear. Start with importing the *OneHotEncoder*class and creating a new instance with the output data type set to integer. This doesn’t change anything to how our data will be interpreted, but will improve the readability of our output.

Then, fit and transform our two nominal categoricals. The output of this transformation will be a sparse matrix, this means we’ll have to transform the matrix into an array (*.toarray()*) before we can pour it into a dataframe. You can omit this step by setting the *sparse* parameter to *False*when initiating a new class instance. Assign column names and the output is ready to be added to the other data (*edu\_level*feature).

from sklearn.preprocessing import OneHotEncoderonehot = OneHotEncoder(dtype=np.int, sparse=True)nominals = pd.DataFrame(  
 onehot.fit\_transform(X[['sex', 'blood\_type']])\  
 .toarray(),  
 columns=['F', 'M', 'AB', 'B+','O+', 'O-'])nominals['edu\_level'] = X.edu\_level

Compare the output (*nominals*) to our original data to make sure everything came through the right way.



Encoded data versus original data

Since there were no missing values in our data, it is important to have a word on how to handle missing values with the *OneHotEncoder*. A missing value can easily be handled as an extra feature. Note that to do this, you need to replace the missing value by an arbitrary value first (e.g. *‘missing’*) If you, on the other hand, want to ignore the missing value and create an instance with all zeros (*False*), you can just set the *handle\_unkown*parameter of the *OneHotEncoder*to *ignore*.

**Numerical features**

Just like categorical data can be encoded, numerical features can be ‘decoded’ into categorical features. The two most common ways to do this are **discretization**and **binarization**.

**Discretization**

Discretization, also known as quantization or binning, **divides a continuous feature into a pre-specified number of categories** (bins), and thus makes the data discrete.

One of the main goals of a discretization is to significantly **reduce the number of discrete intervals of a continuous attribute**. Hence, why this transformation can increase the performance of tree based models.

*Sklearn*provides a *KBinsDiscretizer*class that can take care of this. The only thing you have to specify are the number of bins (*n\_bins*) for each feature and how to encode these bins (*ordinal*, *onehot*or *onehot-dense*). The optional *strategy*parameter can be set to three values:

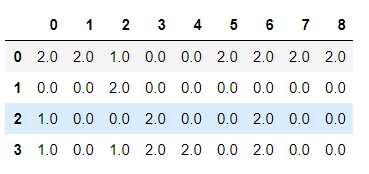
* *uniform*, where all bins in each feature have identical widths.
* *quantile (*default), where all bins in each feature have the same number of points.
* *kmeans*, where all values in each bin have the same nearest center of a 1D k-means cluster.

It is important to pick the strategy parameter with care. Using the uniform strategy for example, is very sensitive for outliers and can make you end up with bins with just a few data points, i.e. the outliers.

Let’s turn to our example for some clarifications. Import the *KBinsDiscretizer*class and create a new instance with three bins, ordinal encoding and a uniform strategy (all bins have the same width). Then, fit and transform all our original, missing indicator and polynomial data.

from sklearn.preprocessing import KBinsDiscretizerdisc = KBinsDiscretizer(n\_bins=3, encode='uniform',   
 strategy='uniform')disc.fit\_transform(X)

If the output doesn’t make sense to you, invoke the *bin\_edges\_*attribute on the discretizer (*disc*) and take a look at how the bins are divided. Then try another strategy and see how the bin edges change accordingly.



Discretized output

**Binarization**

Feature binarization is **the process of tresholding numerical features to get boolean values**. Or in other words, assign a boolean value (*True*or *False*) to each sample based on a threshold. Note that binarization is an extreme form of two-bin discretization.

In general binarization is useful as a **feature engineering technique for creating new features that indicate something meaningful**. Just like the above-mentioned *MissingIndicator*is used to mark meaningful missing values.

The *Binarizer*class in sklearn implements binarization in a very intuitive way. The only parameters you need to specify are the *threshold*and *copy*. All values below or equal to the threshold are replaced by 0, above it by 1. If copy is set to *False*, inplace binarization is performed, otherwise a copy is made.

Consider *feature 3* (*f3*) of our example and let’s create an extra binary feature with *True*for positive values and *False*for negative values. Import the *Binarizer*class, create a new instance with the threshold set to zero and copy to *True*. Then, fit and transform the binarizer to *feature 3.*The output is a new array with boolean values.

from sklearn.preprocessing import Binarizerbinarizer = Binarizer(threshold=0, copy=True)binarizer.fit\_transform(X.f3.values.reshape(-1, 1))

**Custom transformers**

If you want to convert an existing function into a transformer to assist in data cleaning or processing, you can implement a transformer from an arbitrary function with *FunctionTransformer*. This class can be useful if you’re working with a *Pipeline*in *sklearn*, but can easily be replaced by applying a lambda function to the feature you want to transform (as showed below).

from sklearn.preprocessing import FunctionTransformertransformer = FunctionTransformer(np.log1p, validate=True)transformer.fit\_transform(X.f2.values.reshape(-1, 1)) #same outputX.f2.apply(lambda x : np.log1p(x)) #same output

**Feature scaling**

The next logical step in our preprocessing pipeline is to scale our features. Before applying any scaling transformations it is very important to **split your data into a train set and a test set**. If you start scaling before, your training (and test) data might end up scaled around a mean value (see below) that is not actually the mean of the train or test data, and go past the whole reason why you’re scaling in the first place.

**Standardization**

Standardization is a transformation that **centers the data by removing the mean value of each feature and then scale it by dividing (non-constant) features by their standard deviation**. After standardizing data the mean will be zero and the standard deviation one.

Standardization can drastically improve the performance of models. For instance, many elements used in the objective function of a learning algorithm (such as the RBF kernel of Support Vector Machines or the l1 and l2 regularizers of linear models) assume that all features are centered around zero and have variance in the same order. If a feature has a variance that is orders of magnitude larger than others, it might dominate the objective function and make the estimator unable to learn from other features correctly as expected.

Depending on your needs and data, sklearn provides a bunch of scalers: *StandardScaler*, *MinMaxScaler*, *MaxAbsScaler*and *RobustScaler*.

**Standard Scaler**

*Sklearn*its main scaler, the *StandardScaler*, uses a strict definition of standardization to standardize data. It **purely centers the data** by using the following formula, where *u* is the mean and *s*is the standard deviation.

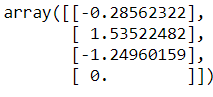
x\_scaled = (x — u) / s

Let’s take a look at our example to see this in practice. Before we start coding, we should remember that the value of our fourth instance was missing, and we replaced it by the mean. If we input the mean in the above formula, the result after standardizing should be zero. Let’s test this.

Import the *StandardScaler*class and create a new instance. Note that for sparse matrices you can set the *with\_mean*parameter to *False*in order not to center the values around zero. Then, fit and transform the scaler to *feature 3*.

from sklearn.preprocessing import StandardScalerscaler = StandardScaler()scaler.fit\_transform(X.f3.values.reshape(-1, 1))

As anticipated, the value of the fourth instance is zero.



Ouput of standard scaling feature 3

**MinMax Scaler**

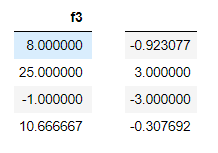
The *MinMaxScaler*transforms features by **scaling each feature to a given range**. This range can be set by specifying the *feature\_range*parameter (default at *(0,1)*). This scaler works better for cases where the distribution is not Gaussian or the standard deviation is very small. However, it is **sensitive to outliers**, so if there are outliers in the data, you might want to consider another scaler.

x\_scaled = (x-min(x)) / (max(x)–min(x))

Importing and using the *MinMaxScaler*works — just as all the following scalers — in exactly the same way as the *StandardScaler*. The only difference sits in the parameters on initiation of a new instance.

Here we scale *feature 3* (*f3*) to a scale between -3 and 3. As expected our maximum value (*25*) is transformed to 3 and our minimum value (-*1*) is transformed to -3. All the other values are linearly scaled between these values.

from sklearn.preprocessing import MinMaxScalerscaler = MinMaxScaler(feature\_range=(-3,3))scaler.fit\_transform(X.f3.values.reshape(-1, 1))



Feature 3 before and after applying the MinMaxScaler

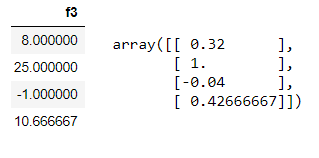
**MaxAbs Scaler**

The *MaxAbsScaler*works very similarly to the *MinMaxScaler*but automatically scales the data to a *[-1,1]* range based on the **absolute maximum**. This scaler is meant for**data that is already centered at zero or sparse data**. It does not shift/center the data, and thus does not destroy any sparsity.

x\_scaled = x / max(abs(x))

Let’s once again tackle *feature 3* by transforming it using the *MaxAbsScaler* and compare the output with the original data.

from sklearn.preprocessing import MaxAbsScalerscaler = MaxAbsScaler()scaler.fit\_transform(X.f3.values.reshape(-1, 1))



Feature 3 before and after applying the MaxAbsScaler

**Robust Scaler**

If your data contains many **outliers**, scaling using the mean and standard deviation of the data is likely to not work very well. In these cases, you can use the *RobustScaler*. **It removes the median and scales the data according to the quantile range**. The exact formula of the *RobustScaler*is not specified by the documentation. If you want full details you can always check [the source code](https://github.com/scikit-learn/scikit-learn/blob/55bf5d9/sklearn/preprocessing/data.py#L1035).

By default, the scaler uses the Inter Quartile Range (IQR), which is the range between the 1st quartile and the 3rd quartile. The quantile range can be manually set by specifying the *quantile\_range*parameter when initiating a new instance of the *RobustScaler*. Here, we transform*feature 3*using an quantile range from *10%* till *90%*.

from sklearn.preprocessing import RobustScalerrobust = RobustScaler(quantile\_range = (0.1,0.9))robust.fit\_transform(X.f3.values.reshape(-1, 1))

**Normalization**

Normalization is the process of **scaling individual samples to have unit norm**. In basic terms you need to normalize data when the algorithm predicts based on the weighted relationships formed between data points. Scaling inputs to unit norms is a common operation for **text classification or clustering**.

*One of the key differences between scaling (e.g. standardizing) and normalizing, is that normalizing is a****row-wise operation****, while scaling is a column-wise operation.*

Although there are many other ways to normalize data, *sklearn*provides three norms (the value to which the individual values are compared):*l1*, *l2* and *max*. When creating a new instance of the *Normalizer*class you can specify the desired norm under the *norm*parameter.

Below, the formula’s for the available norms are discussed and implemented in Python code — where the result is a list of denominators for each sample in data set *X*.

**‘max’**



The *max*norm uses the absolute maximum and does for samples what the *MaxAbsScaler*does for features.

x\_normalized = x / max(x)

norm\_max =   
list(**max**(list(abs(i) for i in X.iloc[r])) for r in range(len(X)))

**‘l1’**

The *l1* norm uses **the sum of all the values** as and thus gives equal penalty to all parameters, enforcing sparsity.

x\_normalized = x / sum(X)

norm\_l1 =   
list(**sum**(list(abs(i) for i in X.iloc[r])) for r in range(len(X)))

**‘l2’**

The l2 norm uses **the square root of the sum of all the squared values**. This creates smoothness and rotational invariance. Some models, like PCA, assume rotational invariance, and so *l2* will perform better.

x\_normalized = x / sqrt(sum((i\*\*2) for i in X))

norm\_l2 =   
list(math.sqrt(sum(list((i\*\*2) for i in X.iloc[r])))   
 for r in range(len(X)))

**— Please feel free to bring any inconsistencies or mistakes to my attention in the comments or by leaving a private note. —**