The complete beginner’s guide to data cleaning and preprocessing

How to successfully prepare your data for a machine learning model in minutes

Data preprocessing is the first (and arguably most important) step toward building a working machine learning model. It’s critical!

If your data hasn’t been cleaned and preprocessed, your model does not work.

It’s that simple.

Data preprocessing is generally thought of as the boring part. But it’s the difference between being prepared and being completely unprepared. It’s the difference between looking like a pro and looking pretty foolish.

It’s kind of like getting ready for a vacation. You might not like the preparation part, but tightening down the details in advance can save you from one nightmare of a trip.

You just have to do it or you can’t start having fun.

GIF via GIPHY

But how do you do it?

This tutorial walks you through the basics of preparing any dataset for any machine learning model.

Imports first!

We want to start by importing the libraries that you’ll need to preprocess your data. A library is really just a tool that you can use. You give the library the input, the library does its job, and it gives you the output you need. There are tons of libraries available, but three are essential libraries in Python. You’ll pretty much wind up using them every time. The three most popular libraries when you’re working with Python are Numpy, Matplotlib, and Pandas. Numpy is the library you’ll need for all things mathematical. Since your code is going to run on math, you’re going to use this one. Matplotlib (specifically Matplotlib.pyplot) is the library you’ll want if you’re going to make charts. Pandas is the best tool available for importing and managing datasets. Pandas and Numpy are basically essential for data preprocessing.

(If you’re new to all of this, you might want to check out the ultimate beginner’s guide to NumPy!)

It makes the most sense to import these libraries with a shortcut alias so that you can save a little time later. That’s simple and you can do it like this:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

Now you can read in your dataset by typing

```
dataset = pd.read_csv('my_data.csv')
```

This tells Pandas (pd) to read in your dataset. These are the first few lines of the dataset I put together for this tutorial:

Now we have our dataset, but we need to create a matrix of dependent variables and a vector of independent variables. You can create the matrix of dependent variables by typing:

```
X = dataset.iloc[:, :-1].values
```

That first colon (:)means that we want to grab all of the lines in our dataset. :-1 means that we want to grab all of the columns of data except the last column. The .values on the end means that we want to grab all of the values.

Now we want a vector of dependent variable with only the data from the last column, so we can type

```
y = dataset.iloc[:, 3].values
```

Remember when you’re looking at your dataset, the index starts at 0. If you’re trying to count the columns, start counting at 0, not 1. [:, 3] gets you the animal, age, and worth columns. 0 is the animal column, 1 is the age column, and 2 is the worth. You will get used to this counting system if you aren’t already!

What happens if we have missing data?

This actually happens all the time.

GIF via GIPHY

We could just remove the lines where data are missing, but that’s a really not the smartest idea. That could easily cause problems. We need to find a better idea! The most common solution is to take the mean of the columns to fill in the missing data point.

You can easily do this with the imputer class from scikit-learn’s preprocessing model. If you don’t know about it already, scikit-learn contains amazing machine learning models and I strongly suggest you check it out!)

You might not be comfortable with terms like "method," "class," and "object" as they apply to machine learning. Not a problem!

A class is the model of something that we want to build. If we're going to build a shed, the construction plan for the shed is the class.

An object is an instance of the class. The object in this example is the shed we built by following the construction plan. There can be many objects of the same class. That's like saying that you can make lots of sheds from the construction plan.

A method is a tool that we can use on the object, or a function that's applied to the object that takes some inputs and returns some output. This is like a handle that we can use to open the window when our shed is starting to get a little stuffy.

Photo by Roman Kraft on Unsplash

To use the imputer, we would run something like this

```
from sklearn.preprocessing import Imputer
imputer = Imputer(missing_values = np.nan, strategy = 'mean', axis = 0)
```

Mean is the default strategy, so you don't actually need to specify that, but it's here so you can get a sense of what information you want to include. The default values for missing_values is nan. If your data set has missing values that are called "NaN," you'll stick with np.nan. Check out the official documentation here!

Now to fit this imputer, we type

```
imputer = imputer.fit(X[:, 1:3])
```

We only want to fit the imputer to the columns where data are missing. The first colon means that we want to include all of the lines, while 1:3 means that we're taking column indexes 1 and 2. Don't worry. You'll get used to the way Python counts in no time!

Now we want to use the method that will actually replace the missing data. You'll set that up by typing

```
X[:, 1:3] = imputer.transform(X[:, 1:3])
```

Try this out with other strategies! You might find that it makes more sense for your project to fill in the missing values with the median of the column. Or the mode! Decisions like these seem small, but they actually hold a lot of importance.

Just because something is popular doesn't necessarily make it the right choice. The average (mean) of your data points isn't necessarily the best choice for your model.

After all, nearly everyone reading this article has an above average number of arms…

Photo by Matthew Henry on Unsplash

What if you have categorical data?

Great question! You can't exactly take the mean of cat, dog, and moose. What can we do? We can encode the categorical values as numbers! You'll want to grab the Label Encoder class from sklearn.preprocessing.

Start with one column where you want to encode the data and call the label encoder. Then fit it onto your data

```
from sklearn.preprocessing import LabelEncoder
labelencoder_X = LabelEncoder()
X[:, 0] = labelencoder_X.fit_transform(X[:, 0])
```

(Remember how the numbers in the brackets work? : means that we want to work with all of the lines and 0 means that we want to grab the first column.)

That's all it takes to replace the categorical variables in your first column with numbers. For example, instead of moose, you'll have "0," instead of "dog" you'll have "2," and instead of "cat," you'd have "3."

Do you see the potential problem?

That system of labeling implies a hierarchical value to the data that could affect your model. 3 has a higher value than 0, but cat is not (necessarily…) greater than moose.

Photo by [Cel Lisboa](#) on [Unsplash](#)

We need to create dummy variables!

We can create one column for cat, one for moose, and so on. Then we'll fill the columns in with 1s and 0s (think 1=yes and 0=no.) That means that if you had cat in your original column, now you'd have a 0 in the moose column, a 0 in the dog column, and a 1 in the cat column.

That sounds complicated. Enter One Hot Encoder!

Import the encoder and then specify the index of the column

```
from sklearn.preprocessing import OneHotEncoder
onehotencoder = OneHotEncoder(categorical_features = [0])
```

Now a little fit and transform

```
X = onehotencoder.fit_transform(X).toarray()
```

Voila! Your single column has been replaced by one column for each of the categorical variables that you had in your original column and it has 1s and 0s replacing the categorical variables.

Pretty sweet, right?

We can go ahead and use label encoder for our y column if we have categorical variables like "yes" and "no."

```
labelencoder_y = LabelEncoder()
y = labelencoder_y.fit_transform(y)
```

This will go ahead and fit and transform y into an encoded variable with 1 for yes and 0 for no.

Train test split

At this point, you can go ahead and split your data into training and testing sets. I know I already said this in the image classification tutorial, but always separate your data into training and testing sets and never use your testing data for training! You need to avoid overfitting. (You can think of overfitting like memorizing super specific details before a test without understanding the information. When you memorize details, you'll do a great job with your flashcards at home. You'll fail any real test, though, where you're presented with new information.)

Right now, we have a machine that needs to learn something. It needs to train on data and see how well it understands what it's learned on separate data. Memorizing the training set is not the same thing as learning! The better your model learns on the training set, the better it will be at predicting the results for the testing set. You never want to overfit your model. You really want it to learn!

Photo by Janko Ferlič on Unsplash

First, we import

```
from sklearn.model_selection import train_test_split
```

Now we can create X_train and X_test and y_train and y_test sets.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_sta
```

It's very common to do an 80/20 split of your data, with 80% of your data going to training and 20% to testing. That's why we specified a test_size of 0.2. You can split it however you need to. You don't need to set a random state, but I like to do that so that we can exactly reproduce our results.

Now for feature scaling.

What is feature scaling? Why do we need it?

Well, look at our data. We have one column with animal ages from 4–17 and we have animal worth that ranges from $48,000-$83,000. Not only is the worth column made up of much higher numbers than the age column, but the variables also cover a much wider range of data. That means that the Euclidean distance will be dominated by worth and will wind up dominating the age data.

What if Euclidean distance doesn't play a part in your specific machine learning model? Scaling the features will still make the model much faster, so you might want to include this step when you're preprocessing your data.

There are many ways to do feature scaling. They all mean that we're putting all of our features into the same scale so that none are dominated by another.

Start with the import (you must be getting used to that)

```
from sklearn.preprocessing import StandardScaler
```

Then create an object that we'll scale and call the standard scaler

```
sc_X = StandardScaler()
```

Now we directly fit and transform our dataset. Grab the object and apply the methods.

```
X_train = sc_X.fit_transform(X_train)
X_test = sc_X.transform(X_test)
```

We don't need to fit it to our test set, we just need a transform.

```
sc_y = StandardScaler()
y_train = sc_y.fit_transform(y_train)
```

What about the dummy variables? Do you need to scale them?

Well, some people say yes and some say no. It's a question of how much you want to hang on to your interpretation. It is good to have all of our data at the same scale. But if we scale our data, we lose our ability to easily interpret which observations belong to which variable.

What about y? If you have a dependent variable like 0 and 1, you really don't need to apply feature scaling. It's a classification problem with a categorically dependent value. But if you have a large range of feature values, then yes! You do want to apply the scaler!

You did it!

That's it!

Photo by Roven Images on Unsplash

With just a handful of lines of code, you've taken care of the basics of data cleaning and preprocessing! You can see the code here if want to take a look.

There will definitely be a ton of thought that you'll need to put into this step. You want to think about exactly how you're going to fill in your missing data. Consider whether you need to scale your features and how you want to do it. Dummy variables or no? Are you going to encode your data? Will you encode your dummy variables? There are a ton of details to consider here.

That said, you've got this!

Now get out there and get that data ready!

As always, if you’re doing anything cool with this information, let people know about it in the responses below or reach out any time on LinkedIn @annebonnerdata!

If you liked this, you might be interested in some of my other articles as well:

Thanks for reading!