


IIT Madras

ONLINE DEGREE

Online Degree Programme
B. Sc in Programming and Data Science
Diploma Level
Dr. Ashish Tendulkar
Indian Institute of Technology – Madras

Data Preparation

(Refer Slide Time: 00:10)



Step 4: Prepare data for ML algorithm

We often need to preprocess the data before using it for model building due to variety of reasons:

- Due to errors in data capture, data may contain outliers or missing values.
- Different features may be at different scales.
- The current data distribution is not exactly amenable to learning.

Typical steps in data preprocessing are as follows:

1. Separate features and labels.
2. Handling missing values and outliers.
3. Feature scaling to bring all features on the same scale.
4. Applying certain transformations like log, square root on the features.

It's a good practice to make a copy of the data and apply preprocessing on that copy. This ensures that in case something goes wrong, we will at least have original copy of the data intact.

Namaste welcome to the next video of machine learning practice course. In this week we are studying end-to-end machine learning project and we are discussing various steps that are involved in end-to-end machine learning project. So, far we looked at the first three steps the first one being looking at the big picture second is getting the data and the third is data visualization.

Now we are on the fourth step which is about preparation of data for machine learning algorithm. So, in this video we will look at various steps that are involved in data preparation for machine learning algorithm. We often need to pre-process the data before using it for model building due to variety of reasons. There could be errors in data capture. Data may contain outliers or missing values.

Imagine there is a sensor measuring the wind quality and if sensor starts malfunctioning we might see different values than otherwise. In machine learning algorithm generally we use different features to represent an example these different features may be at different scales. The current data distribution is exactly not amenable to learning. So, we apply

certain steps for data processing and these steps are like separating features and labels handling missing values and outliers feature scaling to bring all features on the same scale and in applying certain transformations like log square root on the features.

So, you can see that these four steps that we are talking about are addressing the issues that we discussed a few minutes ago. For example, handling missing values and outliers help us to fix error due to data capture. If there are outliers or missing values in the data, the handling of missing values will take care of you know filling missing values with suitable values.

The feature scaling helps us to bring all features on the same scale and transformations like log and square root might help us to get the data distribution that is more amenable to learning. It is a good practice to make a copy of the data and apply preprocessing on that copy this ensures that if something goes wrong our original copy of the data remains intact. (Refer Slide Time: 03:07)

4.2 Data cleaning

Let's first check if there are any missing values in feature set: One way to find that out is column-wise.

```
1 wine_features.isna().sum() # counts the number of NaN in each column of wine_fe
```

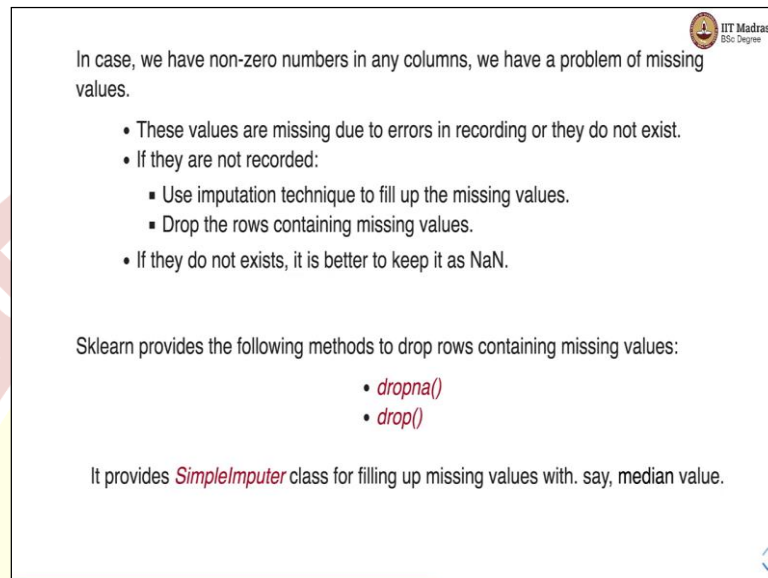
fixed acidity	0
volatile acidity	0
citric acid	0
residual sugar	0
chlorides	0
free sulfur dioxide	0
total sulfur dioxide	0
density	0
pH	0
sulphates	0
alcohol	0
dtype:	int64

In this dataset, we do not have any missing values.

Let us look at the first step where we separate features and labels from the training set. So, in order to separate the feature, we simply apply drop method on the training set. So, in this case we dropped this label which is quality before applying data cleaning we first check if there are any missing values in the feature set one way of it one way of finding it out is as follows. So, we do column wise operation we can we calculate we apply is in a function on each column and then sum it up.

This counts the number of isna's in each columns or number of missing values in each column. In this data set you can see that there are no missing values because the values in the column are the values you know that we obtained after this operation are all zero. So, there are no missing values.

(Refer Slide Time: 04:18)



In case, we have non-zero numbers in any columns, we have a problem of missing values.

- These values are missing due to errors in recording or they do not exist.
- If they are not recorded:
 - Use imputation technique to fill up the missing values.
 - Drop the rows containing missing values.
- If they do not exists, it is better to keep it as NaN.

Sklearn provides the following methods to drop rows containing missing values:

- `dropna()`
- `drop()`


It provides `SimpleImputer` class for filling up missing values with. say, median value.

In case we have non-zero numbers in any column we have problem of missing values. These values are missing due to errors in recording or these values simply do not exist. If the values are not recorded, we can use imputation technique to fill up the missing values. The second option is to drop the rows containing the missing values. The values dot exist it is better to keep them as NaN.

Sklearn provides a couple of methods for dropping rows containing missing values. We can we can call drop in a method or drop method to draw to draw pros containing the missing values but what happens is in case of machine learning the data is scarce and throwing away any data like this could be costly. Hence we try to use imputation technique to fill up the missing values.

Sklearn provides simple imputer class for filling up missing values and his missing values can be filled up with let us say median value or average value or maybe lowest value or highest value I mean depending on the feature and the context.

(Refer Slide Time: 05:45)



```

1 from sklearn.impute import SimpleImputer
2 imputer = SimpleImputer(strategy="median")

```

The strategy contains instructions as how to replace the missing values. In this case, we specify that the missing value should be replaced by the median value.

```

1 imputer.fit(wine_features)


```

SimpleImputer(add_indicator=False, copy=True, fill_value=None, missing_values=nan, strategy='median', verbose=0)

In case, the features contains non-numeric attributes, they need to be dropped before calling the fit method on imputer object.

In this case we import the simple computer class from sklearn and we apply the median strategy for filling up the missing values. In this case every missing value will be replaced by the median of that particular feature. We first called the fit method on computer for that particular feature. The fit method learns the strategy it will basically calculate the median value in this particular case.

(Refer Slide Time: 06:36)



Let's check the statistics learnt by the imputer on the training set:

```

1 imputer.statistics_

```

array([7.9 , 0.52 , 0.26 , 2.2 , 0.08 , 14. , 39. , 0.99675, 3.31 , 0.62 , 10.2])

Note that these are median values for each feature. We can cross-check it by calculating median on the feature set:

```

1 wine_features.median()

```

fixed acidity	7.00000
volatile acidity	0.52000
citric acid	0.26000
residual sugar	2.20000
chlorides	0.08000
free sulfur dioxide	14.00000
total sulfur dioxide	39.00000
density	0.99675
pH	3.31000
sulphates	0.62000
alcohol	10.20000

dtype: float64

We can learn the statistics learned by computer with simple statistics underscore member variable. And here you can see on median values for each feature in our dataset. We can cross check it by actually calculating median on the feature set. And you can see that these values match exactly is 7.9 for fixed acidity the imputer also learned 7.9 as the value of median then 0.52 for volatile acidity and so on.

You can check each of these values over here and the values that are learned by imputer and we find that these values are matching and note that we have already separated the label which is quality which is not present in the feature set.

(Refer Slide Time: 07:30)

Let's check the statistics learnt by the imputer on the training set:

```
1 imputer.statistics_  
array([ 7.9 , 0.52 , 0.26 , 2.2 , 0.08 , 14. , 39. , 0.99675, 3.31 ,  
0.62 , 10.2 ])
```

Note that these are median values for each feature. We can cross-check it by calculating median on the feature set:

```
1 wine_features.median()  
fixed acidity      7.90000  
volatile acidity   0.52000  
citric acid        0.26000  
residual sugar     2.20000  
chlorides          0.08000  
free sulfur dioxide 14.00000  
total sulfur dioxide 39.00000  
density            0.99675  
pH                 3.31000  
sulphates          0.62000  
alcohol            10.20000  
dtype: float64
```

Finally, we use a trend imputer to transform the training set such that the missing values are replaced by the medians. We simply call the transform method on imputer. The transform method returns a numpy array and we can convert it into data frame if needed. We can look at the shape of the of the transformed feature set and you can see that it has got 1279 examples with 11 features.

(Refer Slide Time: 08:14)

4.3 Handling text and categorical attributes

4.3.1 Converting categories to numbers:

```
1 from sklearn.preprocessing import OrdinalEncoder  
2 ordinal_encoder = OrdinalEncoder()
```

- Call `fit_transform()` method on `ordinal_encoder` object to convert text to numbers.
- The list of categories can be obtained via `categories_` instance variable.

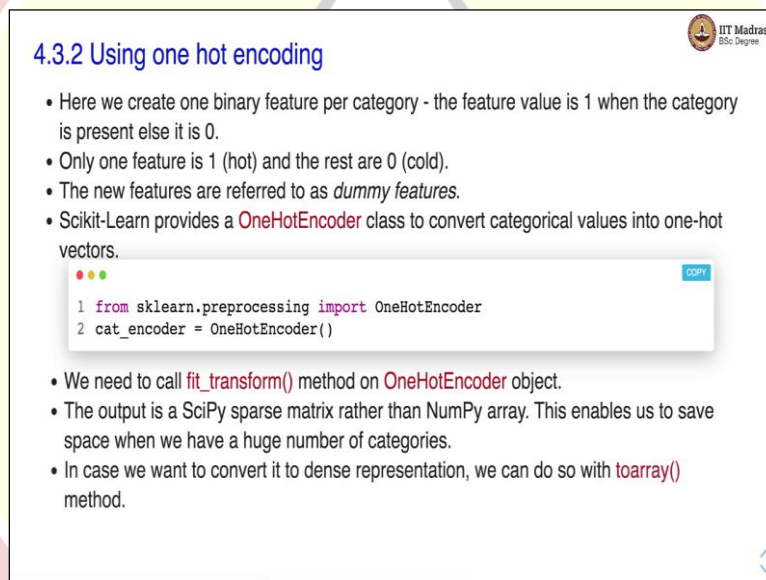
One issue with this representation is that the ML algorithm would assume that the two nearby values are closer than the distinct ones.

We can convert this numpy array into the data frame sometimes we might have text or categorical features in the dataset and we need special handling for these features. Since

machine learning algorithms prefer numerical data what we basically do here is we take this text or categorical attributes and convert them into numbers. Let us look at how to do it. First option is to convert the categories to numbers. We can simply use ordinal encoder transformation for that ordinal and encoder transformation called the fit transfer method on ordinal encoder objects to convert text to numbers and text is also an example of a category.

The list of categories can be obtained via categories underscore instance variable. One issue with this representation is that ML algorithm would assume that the two nearby values are closer than the distinct ones that could be a real problem if there is no systematic ordering in the categories.

(Refer Slide Time: 09:27)



4.3.2 Using one hot encoding

- Here we create one binary feature per category - the feature value is 1 when the category is present else it is 0.
- Only one feature is 1 (hot) and the rest are 0 (cold).
- The new features are referred to as *dummy features*.
- Scikit-Learn provides a **OneHotEncoder** class to convert categorical values into one-hot vectors.

```
1 from sklearn.preprocessing import OneHotEncoder
2 cat_encoder = OneHotEncoder()
```

- We need to call **fit_transform()** method on **OneHotEncoder** object.
- The output is a SciPy sparse matrix rather than NumPy array. This enables us to save space when we have a huge number of categories.
- In case we want to convert it to dense representation, we can do so with **toarray()** method.

In that case we use one hot encoding as an option. Here we create one binary feature per category the feature value is 1 when the category is present otherwise it is 0. Only one feature is 1 which is hot and the rest of them are 0 which is cold. The new features are referred to as dummy features a sklearn provides one hot encoder class to convert the categorical values into one hot vectors.

And again one hot encoder is a transformation so that it has got fit underscore transfer method it returns scifi sparse matrix as an output rather than numpy array because the new representation is pretty sparse many features are actually zero and only few features are one. So, this enables us to save a huge amount of space for this new representation in case you want to convert it into dense representation.

We can do it with two array method. Again the list of categories can be obtained via categories underscore member variable.

(Refer Slide Time: 10:55)

- As we observed that when the number of categories are very large, the one-hot encoding would result in a very large number of features.
- This can be addressed with one of the following approaches:
 - Replace with categorical numerical features
 - Convert into low-dimensional learnable vectors called *embeddings*



As we observe that when the number of categories are very large the one hot encoding would result in a very large number of features. This can be addressed with one of the following approaches we can either replace one hot encoding representation with the categorical numerical features or we can convert it into a lower dimensional learnable vector called embeddings.

(Refer Slide Time: 11:20)

4.4 Feature Scaling

- Most ML algorithms do not perform well when input features are on very different scales.
- Scaling of target label is generally not required.

4.5.1 Min-max scaling or Normalization

- We subtract minimum value of a feature from the current value and divide it by the difference between the minimum and the maximum value of that feature.
- Values are shifted and scaled so that they range between 0 and 1.
- Scikit-Learn provides *MinMaxScaler* transformer for this.
- One can specify hyperparameter *feature_range* to specify the range of the feature.

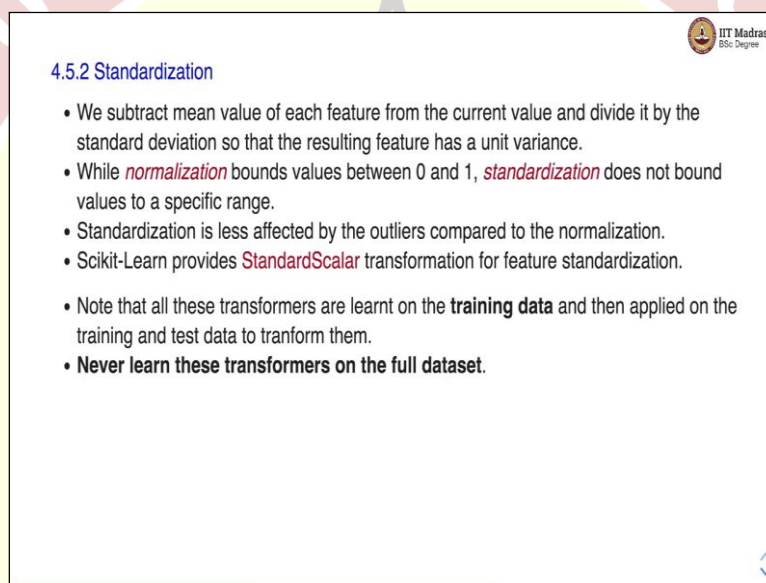


So, it is often observed that in different features that are used for representing examples are at different scales and most of the machine learning algorithms do not perform well

when input features are on different scales. Scaling of target label is generally not required. We generally do scaling only on the features. There are variety of scaling approaches that can be used the first one is mean max scaling or normalization.

In mean max scaling we subtract minimum value of the feature from the current value and then divide it by the difference between the minimum and maximum value of that feature. The values are shifted and scaled so, that they range between 0 and 1. Scikit-learn provide min max scalar transformer for this. One can specify the hyper parameter of feature range to specify the range of the feature.

(Refer Slide Time: 12:29)



4.5.2 Standardization

- We subtract mean value of each feature from the current value and divide it by the standard deviation so that the resulting feature has a unit variance.
- While *normalization* bounds values between 0 and 1, *standardization* does not bound values to a specific range.
- Standardization is less affected by the outliers compared to the normalization.
- Scikit-Learn provides **StandardScaler** transformation for feature standardization.
- Note that all these transformers are learnt on the **training data** and then applied on the training and test data to transform them.
- **Never learn these transformers on the full dataset.**

The second approach for feature scaling is standardization. In standardization we subtract mean value of each feature from the current value and then divide it by the standard deviation. So, that the resulting feature has a unit variance. While normalization bounds when normalization bounds values between 0 and 1 standardization does not bound values to a specific range.

Standardization is less affected by outliers compared to normalization. A Scikit-learn provides standard scalar transformation for feature trans standardization. Note that all these transformers are learnt on the training set and then apply it on the training and test set to transform them. Remember that we should apply the same kind of transformation to training and test in order to get in order to have the correct machine learning pipeline.

These make sure that we are applying the same kind of transformation on the test as that as that was applied on the training set. We never learn these transformers on the full data set.

(Refer Slide Time: 13:53)

Transformation Pipeline

- Scikit-Learn provides a Pipeline class to line up transformations in an intended order.
- Here is an example pipeline:

```
1 from sklearn.pipeline import Pipeline
2 from sklearn.preprocessing import StandardScaler
3 transform_pipeline = Pipeline([
4     ('imputer', SimpleImputer(strategy="median")),
5     ('std_scaler', StandardScaler())])
6 wine_features_tr = transform_pipeline.fit_transform(wine_features)
```

Let's understand what is happening here:


- **Pipeline** has a sequence of transformations - missing value imputation followed by standardization.
- Each step in the sequence is defined by **name, estimator** pair.
- Each name should be unique and **should not contain** __ (double underscore).

So, depending on the nature of the feature set you can decide your transformation. So, for example if your data set has outliers then standardization might be the transformation that you want to use rather than normalization. So, in order to make sure that we apply uniform preprocessing to both training and test set we use a pipeline class provided by Scikit-learn. Pipeline class helps us to line up transformation in an intended order.

Here is an example of pipeline. The pipeline class we specify a sequence of transformation. Here we are we have two transformations one is the imputer transformation and second is standard scalar transformation. So, the imputer transformation helps us to fill up the missing values whereas standard scalar helps us in standardization. You can note that each step is defined by the name and the estimator pair.

For imputer we are going to use simple computer estimator and for standard scalar we are going to use standard scalar estimator. Each name should be unique and should not contain double underscore.

(Refer Slide Time: 15:34)




```
1 from sklearn.pipeline import Pipeline
2 from sklearn.preprocessing import StandardScaler
3 transform_pipeline = Pipeline([
4     ('imputer', SimpleImputer(strategy="median")),
5     ('std_scaler', StandardScaler())])
6 wine_features_tr = transform_pipeline.fit_transform(wine_features)
```

- The output of one step is passed on the next one in sequence until it reaches the last step.
 - Here the pipeline first performs imputation of missing values and its result is passed for standardization.
- The pipeline exposes the same method as the final estimator.
 - Here **StandardScaler** is the last estimator and since it is a transformer, we call **fit_transform()** method on the **Pipeline** object.

The output of one step in this case of the simple imputer is passed to the standard scalar. So, what really happens is whatever data that we get we first fill up the missing values with simple imputer and then we apply standard scaling on that on the resulting data set from the simple imputer. Now in this transform underscore pipeline the standard scalar is the last transformation that we are using.

So, the pipeline would expose the method of the last estimator. In this case the last estimator is standard scalar and since it is a transformer we call a fit transfer method on the pipeline object.

(Refer Slide Time: 16:26)



How to transform mixed features?

- The real world data has both categorical as well as numerical features and we need to apply different transformations to them.
- Scikit-Learn introduced **ColumnTransformer** for this purpose.

```
1 from sklearn.compose import ColumnTransformer
```

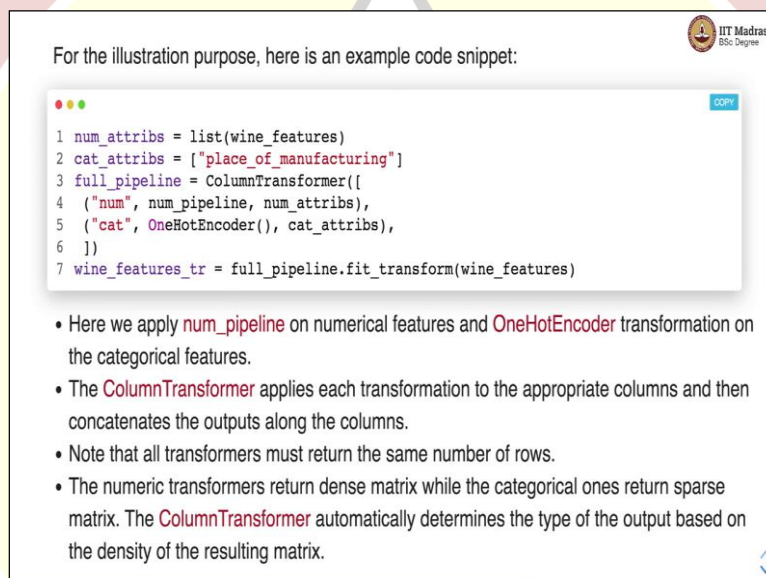
In our dataset, we do not have features of mixed types. All our features are numeric.

So, this works well when we have features of the of the same type. What if we have features of different types and it is very common in real world situation that features will be of

different types. Some features will be numerical features some features might be categorical features and so on. So, transforming this mixed feature can be a challenging thing. But a Scikit-learn has a way of Scikit-learn has provided a way for us to transform these mixed features.

Sklearn provides column transformer for exactly this particular purpose. We can import column transformer class from sklearn dot compose package. In our current data set for wine prediction wine quality prediction we do not have features of mixed types. All our features are numeric. So, it was easier for us to apply you know the same kind of transformer on all features.

(Refer Slide Time: 17:36)



For the illustration purpose, here is an example code snippet:

```
1 num_attribs = list(wine_features)
2 cat_attribs = ["place_of_manufacturing"]
3 full_pipeline = ColumnTransformer([
4     ("num", num_pipeline, num_attribs),
5     ("cat", OneHotEncoder(), cat_attribs),
6 ])
7 wine_features_tr = full_pipeline.fit_transform(wine_features)
```

- Here we apply `num_pipeline` on numerical features and `OneHotEncoder` transformation on the categorical features.
- The `ColumnTransformer` applies each transformation to the appropriate columns and then concatenates the outputs along the columns.
- Note that all transformers must return the same number of rows.
- The numeric transformers return dense matrix while the categorical ones return sparse matrix. The `ColumnTransformer` automatically determines the type of the output based on the density of the resulting matrix.

So, here for the purpose of illustration we have a code snippet using the column transformer. Here we are defining what kind of transformers that should be applied on different type of attributes. For numerical features we will use num underscore pipeline and for categorical features we will use OneHotEncoder. The column transformer applies each transformation to the appropriate column and then concatenates output along the columns.

And the columns are specified over here. Here we are specified numerical attributes that are coming from this particular list of fine features. So, this is a list of attributes on which this numpy pipeline transformation will be you have applied. Categorical attribute which is place of manufacturing and this OneHotEncoder will be applied on this place of manufacturing which is in the list of categorical attributes.

Here there is a single categorical attribute but if we can also specify multiple categorical attributes if they exist in our dataset. So, this is a very, very useful transformer to learn and this will be very useful in real life situations. However, there is only one restriction all transformers must return the same number of rows. If that does not happen we will have problem while concatenating the outputs along different columns.

In this case the numerical transformer returns a tens matrix while the categorical transformer returns a sparse matrix. The column transformer automatically determines the type of the output based on the density of the resulting matrix. That is all from the data preprocessing steps. We looked at how to handle the missing values how to standardize the features and how to apply the pre-processing uniformly to training and test set using pipeline class.

This content will be very useful for you while solving the real-world. In problem next step we will look at how to train the machine learning model.

