INDIAN INSTITUTE OF TECHNOLOGY MADRAS • सिद्धिर्भवति कर्मजा •

IIT Madras

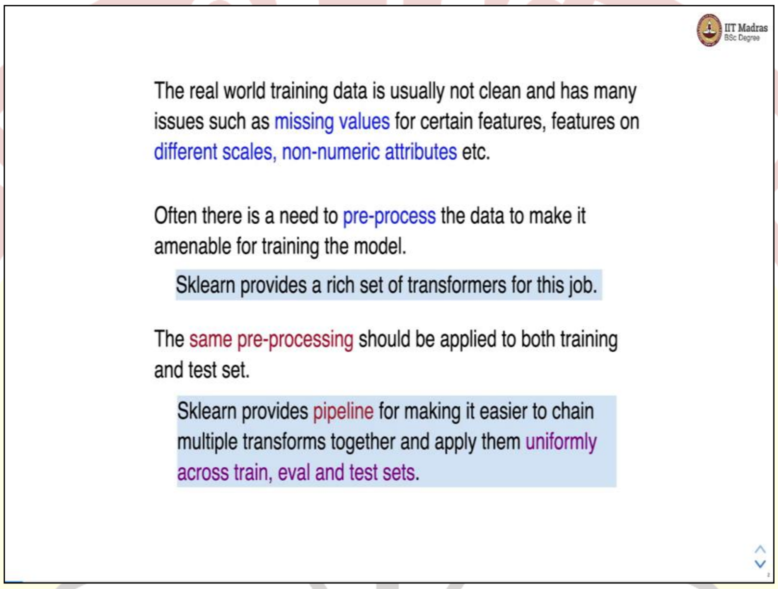
ONLINE DEGREE

Machine Learning Practice
Online Degree Programme
B. Sc in Programming and Data Science
Diploma Level

Dr. Ashish Tendulkar
Indian Institute of Technology – Madras

Data Preprocessing

(Refer Slide Time: 00:20)



The real world training data is usually not clean and has many issues such as missing values for certain features, features on different scales, non-numeric attributes etc.

Often there is a need to pre-process the data to make it amenable for training the model.

Sklearn provides a rich set of transformers for this job.

The same pre-processing should be applied to both training and test set.

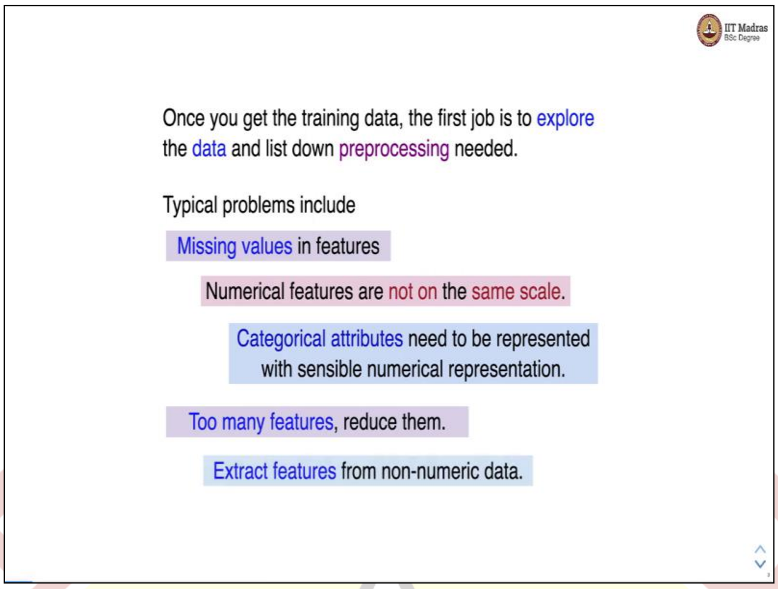
Sklearn provides pipeline for making it easier to chain multiple transforms together and apply them uniformly across train, eval and test sets.

Namaste welcome to the next video of machine learning practice course. In this video we will discuss about data preprocessing. In the real world training data is hardly perfect it is usually not clean and it has got many issues such as missing values for certain features, features on different scales non-numeric attributes etcetera. Often there is a need to pre-process the data to make it amenable for training the model.

Sklearn provides a rich set of transformers for this job. However, we have to make sure that we apply the same set of transformations to both training test as well as on the evaluation set. And this is where most of the real world pipeline sometimes you know if they do not take enough care it may cause some kind of a problem. Fortunately, Sklearn provides a pipeline class for making it easier to chain multiple transformers together.

And pipeline also ensures that we apply these transformations uniformly across training eval and test sets.

(Refer Slide Time: 01:37)


Once you get the training data, the first job is to explore the data and list down preprocessing needed.

Typical problems include

Missing values in features

Numerical features are not on the same scale.

Categorical attributes need to be represented with sensible numerical representation.

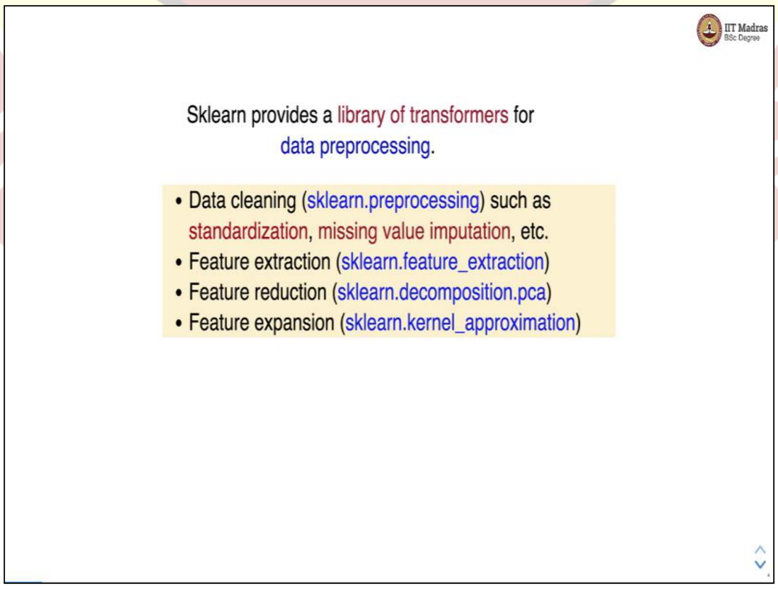
Too many features, reduce them.

Extract features from non-numeric data.

So, usually once you get the training data the first job is to explore the data and list down the preprocessing steps that are needed. Typical problems with training data include missing values in the features, numerical features are often not on the same scale, categorical attributes need to be represented with sensible numerical representation. There are sometimes too many features and not all features are relevant for the learning problem and hence we need to reduce them.

Sometimes we are faced with non-numerical data like images and text and we want to extract features from those non-numerical data.

(Refer Slide Time: 02:29)


Sklearn provides a library of transformers for data preprocessing.

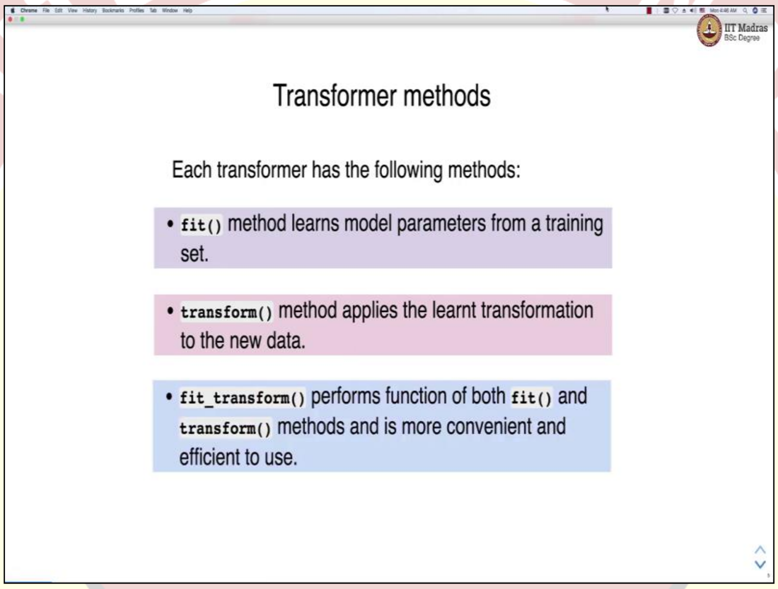
- Data cleaning (sklearn.preprocessing) such as standardization, missing value imputation, etc.
- Feature extraction (sklearn.feature_extraction)
- Feature reduction (sklearn.decomposition.pca)
- Feature expansion (sklearn.kernel_approximation)

So, Sklearn provides a library of transformers for data preprocessing. There are set of transformers for data cleaning they come in the module Sklearn.preprocessing and this

Sklearn_preprocessing module has facility for standardization missing value imputation, feature scaling etcetera. Then we have feature extractors Sklearn.feature_extraction module has bunch of feature extractors to extract features from different data types.

Then we have feature reduction Sklearn.decomposition module has got bunch of feature reduction algorithms implemented one of them is PCA that we will be discussing in this module. And then there are methods for feature expansion that we do especially in kernel methods and we will study this later once we study support vector machines.

(Refer Slide Time: 03:32)



Transformer methods

Each transformer has the following methods:

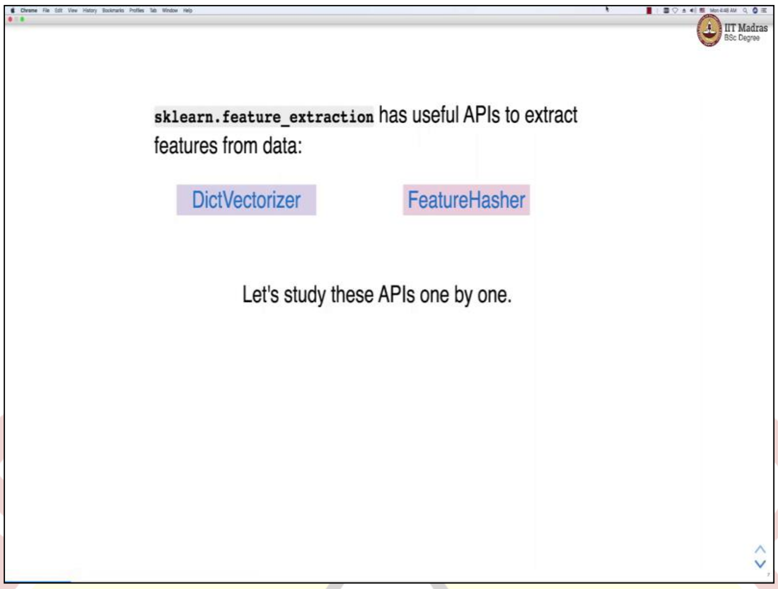
- `fit()` method learns model parameters from a training set.
- `transform()` method applies the learnt transformation to the new data.
- `fit_transform()` performs function of both `fit()` and `transform()` methods and is more convenient and efficient to use.

Each transformer class has the following methods. The fit method learns model parameters from a training set. Transform method applies a learned transformation to the new data and alternatively there is fit_transform method that reforms function of both fit and transform. It is more convenient and efficient to use. Let us understand what fit does.

Let us say we want to scale the features using z score while applying the z score we need mean and standard deviation for the feature and the fit method learns the mean and standard deviation for the features from the training set. It also stores these features and these features are used these parameters are used in the transform method for transforming the dataset from the original set to the transform set.

So, in case of z score the mean and the standard deviation that is learned from the training set is used to transform each feature to its z score using this learned mean and standard deviation.

(Refer Slide Time: 04:48)



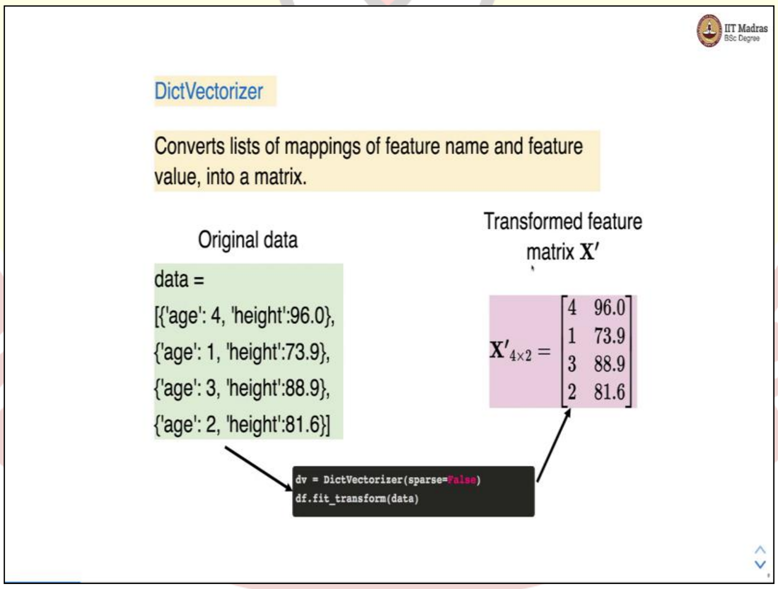
`sklearn.feature_extraction` has useful APIs to extract features from data:

DictVectorizer FeatureHasher

Let's study these APIs one by one.

Let us look at the first part which is feature extraction. Sklearn.feature_extraction module has useful APIs to extract features from the data. There are couple of them which are interesting to us. The first one is dictionary vectorizer or dictatorizer and second one is feature hasher. Let us study these APIs one by one.

(Refer Slide Time: 05:18)



DictVectorizer

Converts lists of mappings of feature name and feature value, into a matrix.

Original data

data =
[{'age': 4, 'height':96.0},
{'age': 1, 'height':73.9},
{'age': 3, 'height':88.9},
{'age': 2, 'height':81.6}]

```
dv = DictVectorizer(sparse=False)
df.fit_transform(data)
```

Transformed feature matrix $\mathbf{X}'$

$$\mathbf{X}'_{4\times2} = \begin{bmatrix} 4 & 96.0 \\ 1 & 73.9 \\ 3 & 88.9 \\ 2 & 81.6 \end{bmatrix}$$

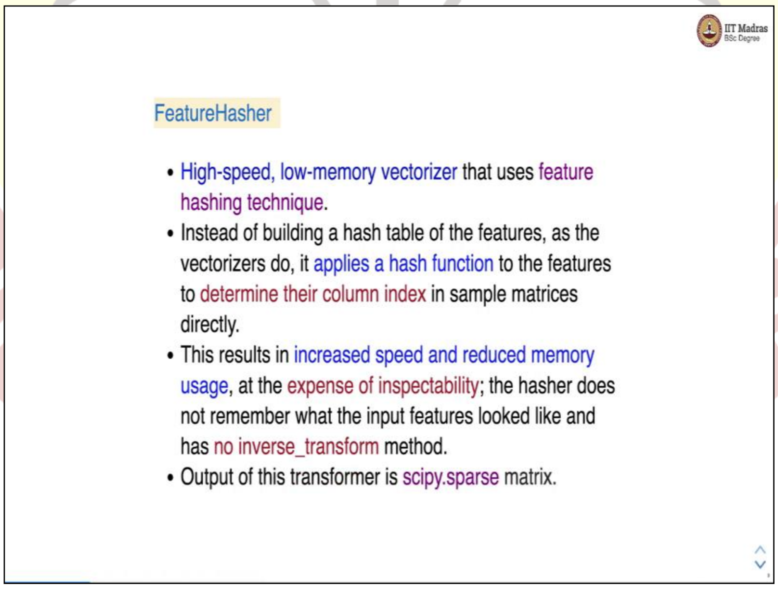
The dictionary vectorizer converts list of mappings of feature names and feature value into a matrix. Let us take a concrete example. Let us say this is the original data which is in form of a data frame. Data frame has got two columns one is age and second is height. And here we have we have four rows or four samples. The first one has age 4 and height of 96 second one has age 1 and height of 73.9 and so on.

When we apply the DictVectorizer, so, we will have to first initialize or we will have to first instantiate the DictVectorizer object and we have to call since it is a transformer it has got fit_transfer method. So, that is the beauty of Sklearn APIs. I mean we studied the design principles of Sklearn and api and these APIs are uniform. Once we know that this is a transformer we can apply fit_transfer method.

So, we instantiate the object we call fit_transfer method on it and we pass the data as an argument. So, this particular data will be transformed by the dictionary vectorizer into a matrix. So, this gets transformed specifically into feature matrix it has got two rows it has got two columns the first column corresponds to the edge and the second column corresponds to height and it has got four rows corresponding to the four samples that were there in the original dataset.

So, you can see that the first sample has got age of 4 and height of 96. The second sample has got age of 1 and height of 73.9 and so on. So, this is how dictionary vectorizer works if you have data present in form of a dictionary or in form of a data frame dictionary vectorizer converts that into the feature matrix.

(Refer Slide Time: 07:26)


IIT Madras
BSc Degree

FeatureHasher

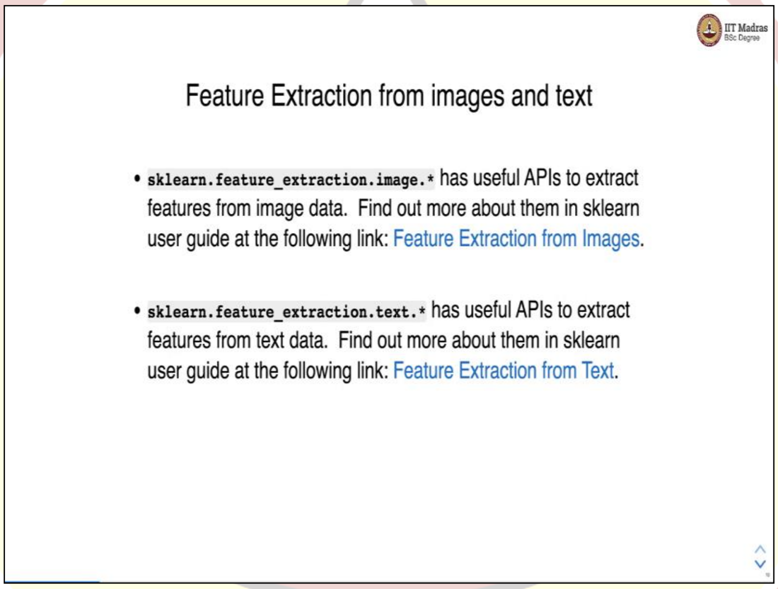
- High-speed, low-memory vectorizer that uses feature hashing technique.
- Instead of building a hash table of the features, as the vectorizers do, it applies a hash function to the features to determine their column index in sample matrices directly.
- This results in increased speed and reduced memory usage, at the expense of inspectability; the hasher does not remember what the input features looked like and has no inverse_transform method.
- Output of this transformer is scipy.sparse matrix.

The second API of our interest is FeatureHasher. FeatureHasher is a high speed low memory vectorizer it uses FeatureHashing technique. So, here instead of building a hash table of features it applies a hash function to the features to determine their column index in the sample matrix directory. This of course results in increased speed and reduced memory usage because now we are not building any kind of hash table.

However there is a cause that we pay and that cost is the in inspectability. We are not able to inspect the features because hasher does not remember what was the original input and it does not have inverse_transfer method to convert the transformed the transform feature back to its original form. So, that is the cost we pay but it is extremely fast and efficient to use especially when we have a very large number of features.

So, FeatureHasher is typically used in cases like text classification where we have a very high dimensional feature vectors. And the FeatureHasher outputs a sparse matrix which is in form of scipy.sparse.

(Refer Slide Time: 09:00)

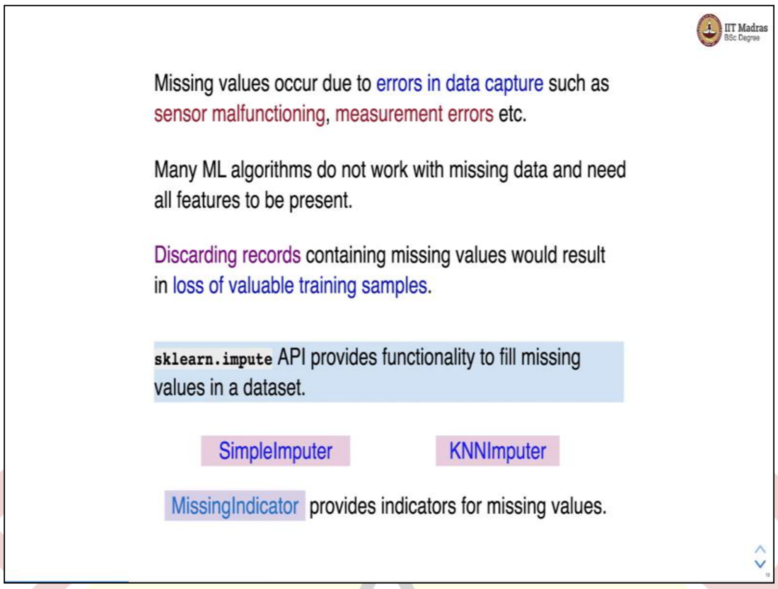


Feature Extraction from images and text

- sklearn.feature_extraction.image.* has useful APIs to extract features from image data. Find out more about them in sklearn user guide at the following link: Feature Extraction from Images.
- sklearn.feature_extraction.text.* has useful APIs to extract features from text data. Find out more about them in sklearn user guide at the following link: Feature Extraction from Text.

We also have sometimes non-numerical data like images and text and Sklearn provides specialized APIs for extracting features from such kind of datasets. In order to extract features from images Sklearn has a Sklearn.feature_extraction.image class and Sklearn also has a corresponding class for extracting features from text. We will not be covering these two modules in detail in this course because they are very specialized but we have provided links here in the Sklearn user guide for you to study these topics further.

(Refer Slide Time: 09:49)



Missing values occur due to errors in data capture such as sensor malfunctioning, measurement errors etc.

Many ML algorithms do not work with missing data and need all features to be present.

Discarding records containing missing values would result in loss of valuable training samples.

`sklearn.impute` API provides functionality to fill missing values in a dataset.

SimpleImputer KNNImputer

MissingIndicator provides indicators for missing values.

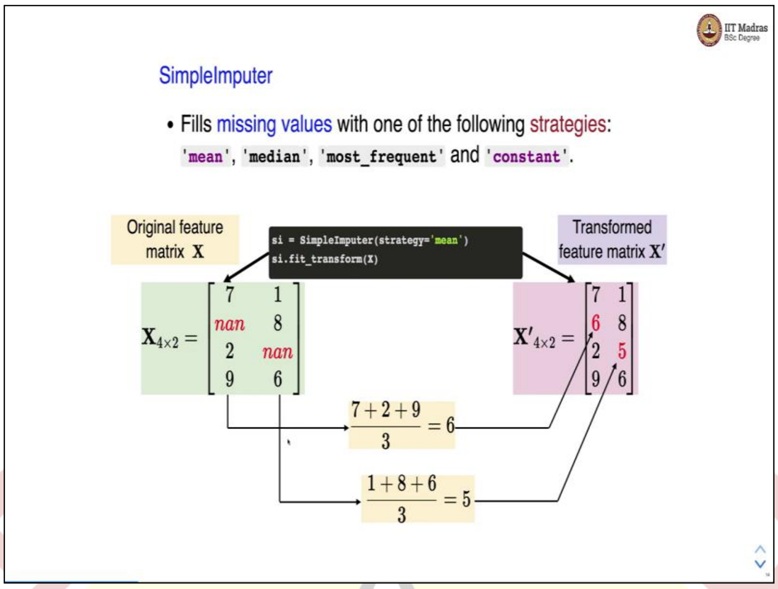
Now that we have extracted the features the next task is cleaning up the data. Let us look at the first line of operation which is handling missing values. So, we had discussed this in the first week when we were learning ml end to end pipeline. We said that missing values occur due to errors in data capture and these errors could be because of sensor malfunctioning, measurement error.

There could be some human components also involved in having the missing values. Unfortunately, many machine learning algorithms do not work with missing data and they need all features to be present. Discarding records that contains missing value is not an option because when you discard the records we are throwing away precious training data hence we need a mechanism to handle these missing values.

So, basically we want mechanism where we can take these on this feature these rows with missing features and you know and fill it up using certain methods. So, a Sklearn impute module provides functionality to fill missing values in the dataset. There are a couple of classes implemented in this module one is SimpleImputer and second is KNNImputer.

These two imputers will help us to fill up the missing values and there is MissingIndicator class that provides indicators for missing values this will be very useful if you want to later examine the data and also find out how the classifiers performance classifier or regressors performance is on these kind of missing features.

(Refer Slide Time: 11:50)



SimpleImputer

- Fills missing values with one of the following strategies: `'mean'`, `'median'`, `'most_frequent'` and `'constant'`.

```
si = SimpleImputer(strategy='mean')
si.fit_transform(X)
```

Original feature matrix X
Transformed feature matrix X′
$\mathbf{X}_{4\times 2} = \begin{bmatrix} 7 & 1 \\ nan & 8 \\ 2 & nan \\ 9 & 6 \end{bmatrix}$
$\mathbf{X'}_{4\times 2} = \begin{bmatrix} 7 & 1 \\ 6 & 8 \\ 2 & 5 \\ 9 & 6 \end{bmatrix}$
$\frac{7+2+9}{3} = 6$
$\frac{1+8+6}{3} = 5$

Let us look at the SimpleImputer class it fills up missing values with one of the following strategies. Now the idea is that we have a feature and in that particular feature there are certain samples that are missing. So, we are we are going to handle this missing values let us say feature by feature. And in one feature let us say there are few samples that are missing then we need to basically come up with a sensible value for those missing values.

So, you know we can either use the mean of that particular feature or we can use median value. We might use most frequent value or we can replace missing value with some kind of with some constant. So, these are the strategies that would be used in SimpleImputer. Let us take a concrete example. Here we have four samples and two features. So, we have feature matrix x with shape (4,2).

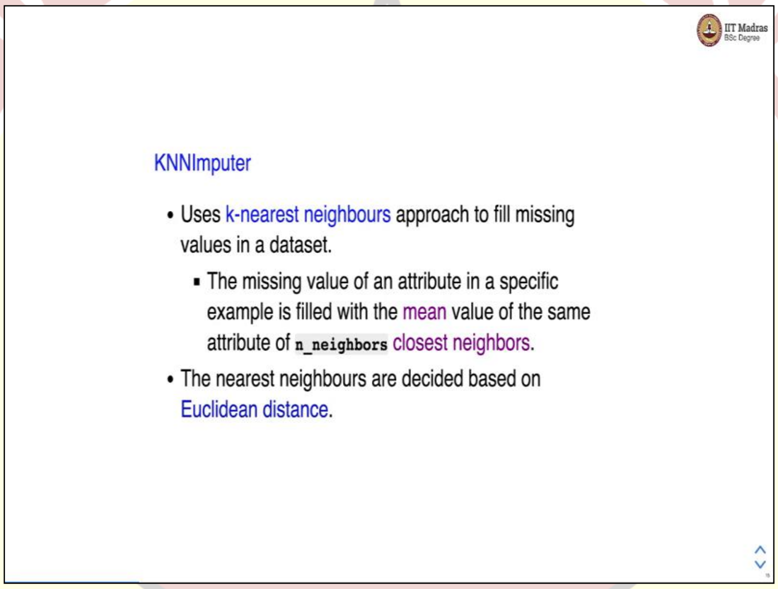
So, here in the first feature there is one missing value and in second feature there is also one missing value. Here the missing value is present in the second sample and in the second column the feature the missing value is present in the third sample. Now what we do is we apply a SimpleImputer on it with strategy mean and this is how we instantiate the SimpleImputer object by specifying the strategy for filling up the missing values.

Then we simply call the fit transform method on this particular feature matrix. So, you can see that x which is our original feature matrix is the argument to fit_transfer method. And we get a transform feature matrix with value 6 which is added which is replacing the nan in the second sample and in the third sample value 5 replaces the nan or the missing values.

Let us see how these values are computed. So, since we are using mean strategy for imputation. The mean of the first column of the first feature is basically 6 and you know how to calculate mean but for your convenience we have shown it we have shown the explicit computation of the mean over here. So, while calculating mean we ignore the value which is missing and we calculate mean with the other values that are present.

So, in this case the mean is 6 and that value is replaced that that value replaces the nan or the missing value. In the second column the mean is 5 and hence we fill up this particular missing value with the mean which is 5.

(Refer Slide Time: 15:00)

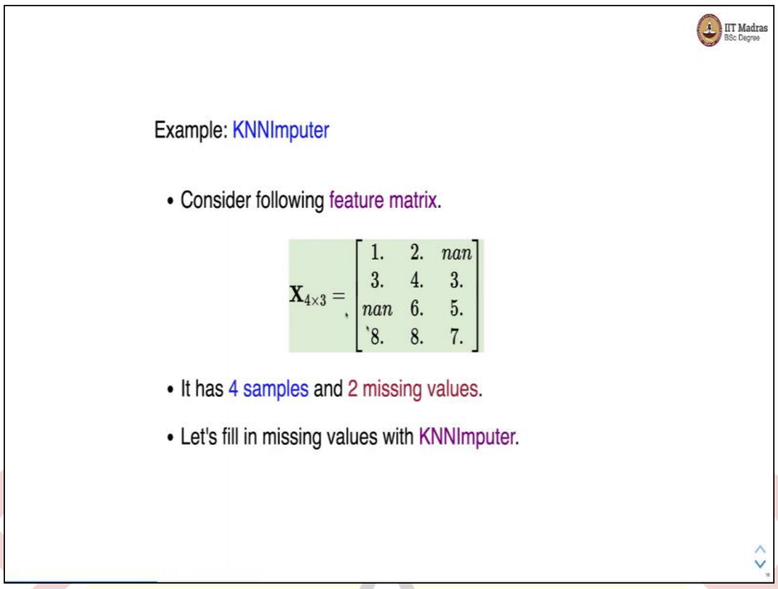


KNNImputer

- Uses k-nearest neighbours approach to fill missing values in a dataset.
  - The missing value of an attribute in a specific example is filled with the mean value of the same attribute of `n_neighbors` closest neighbors.
- The nearest neighbours are decided based on Euclidean distance.

So, KNNimputer is another class that handles the missing values that fills up the missing values it uses k-nearest neighbour approach for filling the missing values. Here what we do is we calculate. So, missing value is there for a specific example. We calculate the n-nearest neighbour for this particular example and then we fill up the missing values of an attribute in that example with the mean value of the same attributes of its nearest neighbours and the nearest neighbours are decided based on euclidean distance.

(Refer Slide Time: 15:49)


Example: KNNImputer

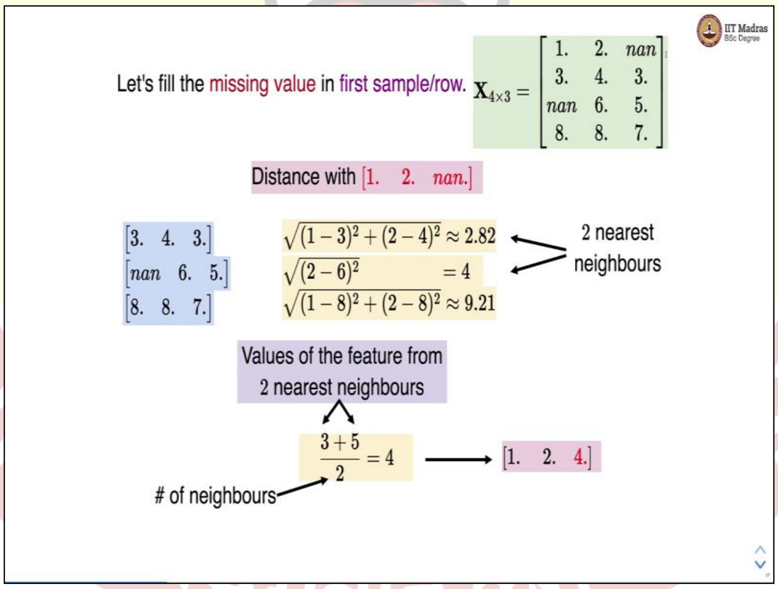
- Consider following feature matrix.

$$\mathbf{X}_{4\times3} = \begin{bmatrix} 1. & 2. & nan \\ 3. & 4. & 3. \\ nan & 6. & 5. \\ 8. & 8. & 7. \end{bmatrix}$$

- It has 4 samples and 2 missing values.
- Let's fill in missing values with KNNImputer.

Let us look at example of KNNImputer. So, let us say we have the following feature matrix it has got four samples and three features. There are two missing values there is a missing value in the first sample and another missing value in the third sample. Now we will fill up these missing values with KNtNImputers.

(Refer Slide Time: 16:20)


Let's fill the missing value in first sample/row.

$$\mathbf{X}_{4\times3} = \begin{bmatrix} 1. & 2. & nan \\ 3. & 4. & 3. \\ nan & 6. & 5. \\ 8. & 8. & 7. \end{bmatrix}$$

Distance with $[1. \quad 2. \quad nan.]$

$[3. \quad 4. \quad 3.]$ $\quad \sqrt{(1-3)^2+(2-4)^2} \approx 2.82$

$[nan \quad 6. \quad 5.]$ $\quad \sqrt{(2-6)^2} = 4$

$[8. \quad 8. \quad 7.]$ $\quad \sqrt{(1-8)^2+(2-8)^2} \approx 9.21$

2 nearest neighbours

Values of the feature from 2 nearest neighbours

$$\frac{3+5}{2} = 4 \longrightarrow [1. \quad 2. \quad 4.]$$

# of neighbours

So, this is the original feature matrix and let us look at the missing value filling for the first sample and the first sample is 1 2 and nan. So, what we will do is we will calculate the distance of this first sample with other samples because the key step here is to find out the nearest neighbour of the sample containing the missing value let us look at the nearest neighbour computation.

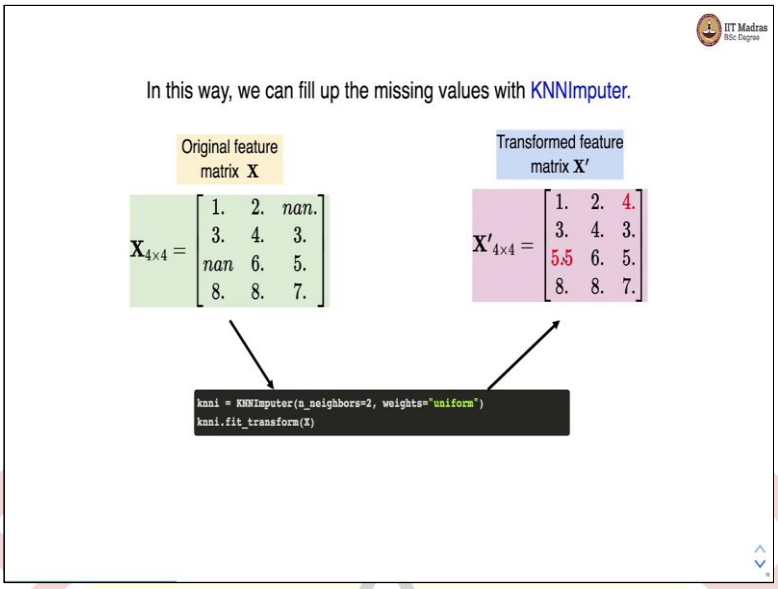
So, first we calculate the distance between this particular sample with let us say the second sample with using the Euclidean distance formula. So, since the third value is nan we do not consider it consider it for the distance computation. So, you know the Euclidean formula is square root of x the feature x1 of the first example one is feature x one of the second example and we square up the difference plus the difference between the second features of these two examples and we square up that difference and we get the value of 2.82.

Next we calculate the distance between the first sample and the third sample. So, we will be doing it we will be calculating distance of first sample with every other sample in the in the dataset. Here there is nan. So, we do not consider that. So, we only have the second feature to consider for calculating the distance and it is 4. In the same way we calculate distance with the last example and that comes out to be 9.21.

So, if you want to use let us say two nearest neighbour. The first the second and the third samples are two nearest neighbors for the first sample or for the first row. So, now what we will do is we will fill up this particular missing value using let us say a mean strategy. So, we calculate the mean of this particular feature based on these two nearest neighbors.

So, in this case we just consider these two values because these are two nearest neighbors and we get the value of 4. So, here 2 is the number of nearest neighbour 3 and 5 are the value of the third feature where we have a missing value in the first example. So, we calculate mean of mean of the values of this particular third feature for these two examples and it comes out to be 4. So, we fill up this particular missing value with 4. So, we get value 1, 2 and 4 after transformation.

(Refer Slide Time: 19:22)


In this way, we can fill up the missing values with KNNImputer.

Original feature matrix $\mathbf{X}$

$$\mathbf{X}_{4\times4} = \begin{bmatrix} 1. & 2. & nan. \\ 3. & 4. & 3. \\ nan & 6. & 5. \\ 8. & 8. & 7. \end{bmatrix}$$

Transformed feature matrix $\mathbf{X}'$

$$\mathbf{X}'_{4\times4} = \begin{bmatrix} 1. & 2. & 4. \\ 3. & 4. & 3. \\ 5.5 & 6. & 5. \\ 8. & 8. & 7. \end{bmatrix}$$

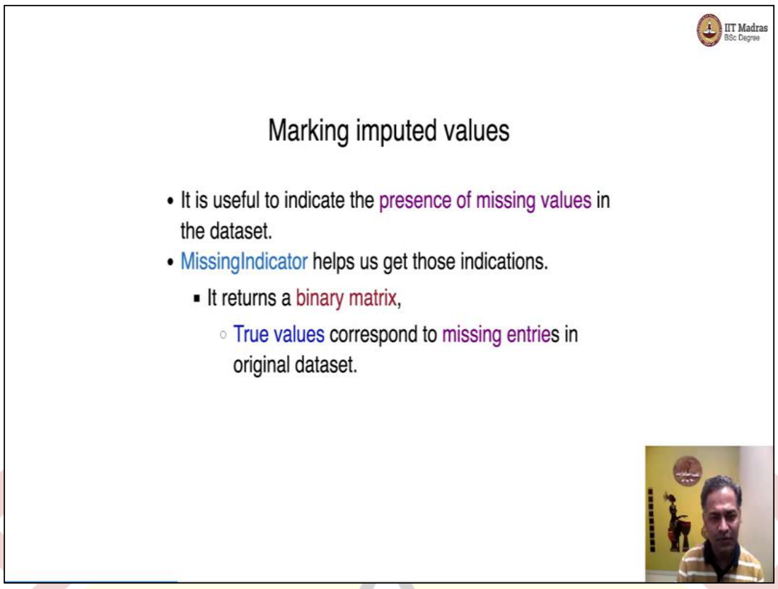
```
knni = KNNImputer(n_neighbors=2, weights="uniform")
knni.fit_transform(X)
```

So, in this way we can fill up the missing values with KNNImputer. So, let us look at the syntax for KNNImputer. We instantiate KNNImputer object by specifying number of nearest neighbors and the weight that we want to apply weight that we want to give to the nearest neighbour. Here we give uniform weight. So, we are basically weighing each neighbors uniformly.

And by default the strategies mean we can change it to other values. And here and then we call fit_transform again notices fit_transform KNNImputer is also a transformer hence there is an it implements fit_transform and then we pass the original feature matrix and we get a transform feature matrix like this. We have already seen how this 4 was computed.

So, you can also see how this 5.5 is computed by doing the same computation that we did in the previous slide.

(Refer Slide Time: 20:35)



Marking imputed values

- It is useful to indicate the presence of missing values in the dataset.
- MissingIndicator helps us get those indications.
  - It returns a binary matrix,
    - True values correspond to missing entries in original dataset.

It is always useful to indicate the presence of missing values in the dataset and missing indicator helps us get those indications. Missing indicator returns a binary matrix where a true value corresponds to missing entries in the original dataset.