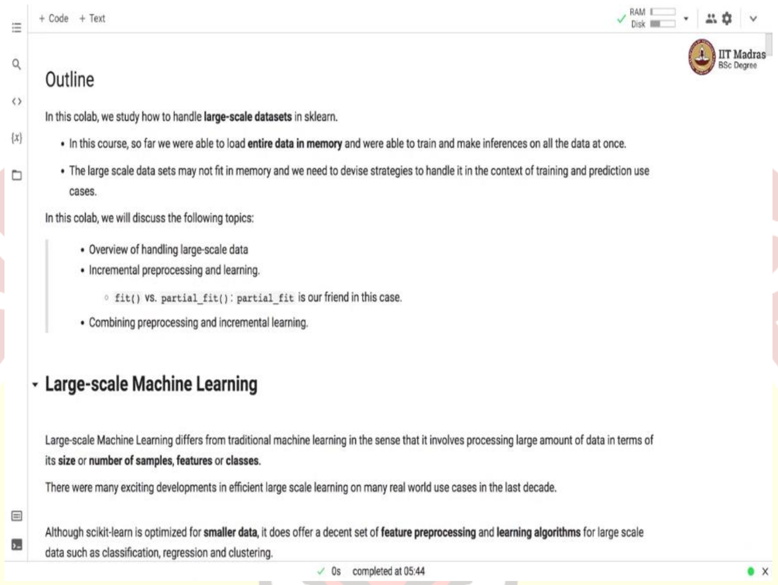
INDIAN INSTITUTE OF TECHNOLOGY MADRAS • सिद्धिर्भवति कर्मजा •

IIT Madras

ONLINE DEGREE

Machine Learning Program
Indian Institute of Technology, Madras
Large-scale Machine Learning

(Refer Slide Time: 0:10)



Outline

In this colab, we study how to handle **large-scale datasets** in sklearn.

- In this course, so far we were able to load **entire data in memory** and were able to train and make inferences on all the data at once.
- The large scale data sets may not fit in memory and we need to devise strategies to handle it in the context of training and prediction use cases.

In this colab, we will discuss the following topics:

- Overview of handling large-scale data
- Incremental preprocessing and learning.
    - `fit()` vs. `partial_fit()`: `partial_fit` is our friend in this case.
- Combining preprocessing and incremental learning.

Large-scale Machine Learning

Large-scale Machine Learning differs from traditional machine learning in the sense that it involves processing large amount of data in terms of its **size** or **number of samples, features** or **classes.**

There were many exciting developments in efficient large scale learning on many real world use cases in the last decade.

Although scikit-learn is optimized for **smaller data**, it does offer a decent set of **feature preprocessing** and **learning algorithms** for large scale data such as classification, regression and clustering.

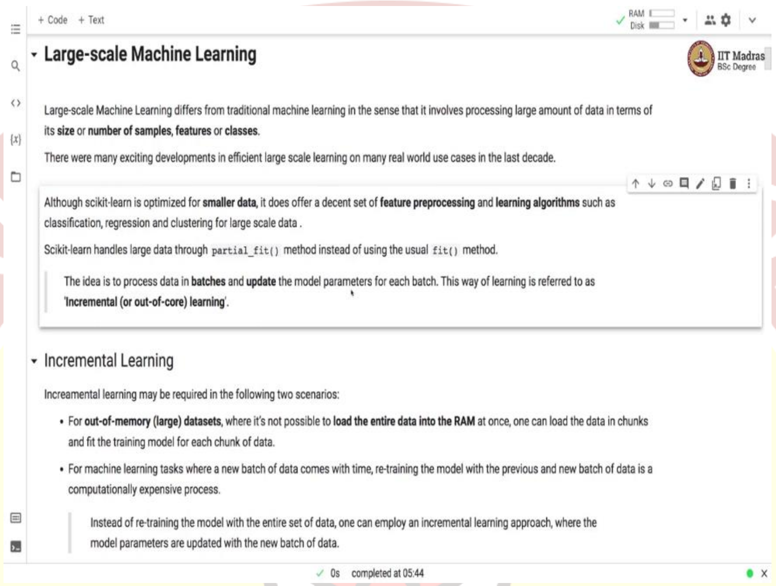
Namaste! Welcome to the next video of Machine Learning Practice Course. In this video, we will study how to handle large-scale datasets in Sklearn. In this course, So, far, we were able to load entire data in memory, and we are able to train and make inferences on all the data at once. The large-scale datasets may not fit in memory. And we need to devise strategies to handle it in context of training and prediction use cases.

Large-scale datasets is what you will encounter in many of the real life applications of this age. There is explosion of data that different companies are collecting from their customers from their processes, from manufacturing units, from different devices. And in order to train machine learning models on this large-scale data, we need special handling of the data, the feature generation process, and retraining process.

Fortunately, sklearn has some functionality for handling large-scale datasets. And the topic of this video is to understand what kind of tools that are available to us in order to perform large- scale learning with sklearn. In this topic, we will discuss tools available in sklearn for large-scale learning. We will discuss overview of handling large-scale data, incremental preprocessing, and learning, basically fit versus partial _fit.

So far, we were using fit function for training the model. Now partial _fit will be our friend. When it comes to large-scale datasets. We will provide a short demonstration for combining preprocessing and incremental learning towards the end of this collab.

(Refer Slide Time: 2:09)



Large-scale Machine Learning

Large-scale Machine Learning differs from traditional machine learning in the sense that it involves processing large amount of data in terms of its **size** or **number of samples**, **features** or **classes**.

There were many exciting developments in efficient large scale learning on many real world use cases in the last decade.

Although scikit-learn is optimized for **smaller data**, it does offer a decent set of **feature preprocessing** and **learning algorithms** such as classification, regression and clustering for large scale data .

Scikit-learn handles large data through `partial_fit()` method instead of using the usual `fit()` method.

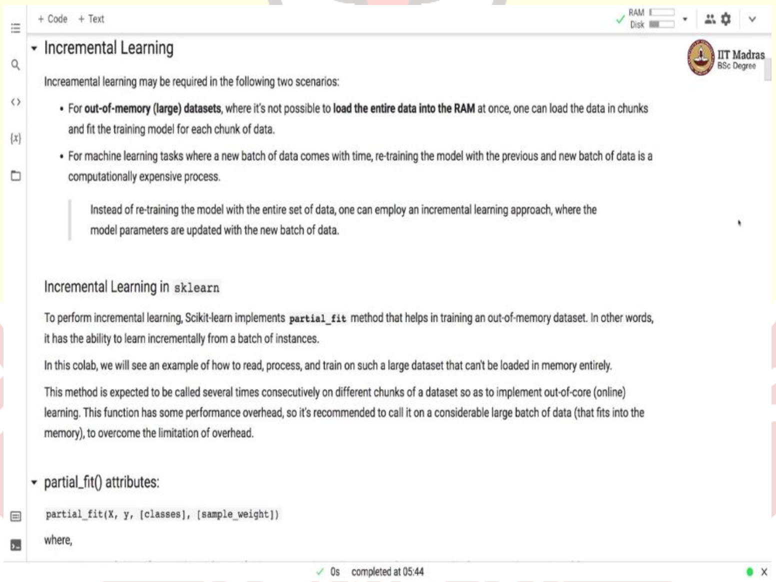
> The idea is to process data in **batches** and **update** the model parameters for each batch. This way of learning is referred to as **'Incremental (or out-of-core) learning'**.

Incremental Learning

Increamental learning may be required in the following two scenarios:

- For **out-of-memory (large) datasets**, where it's not possible to **load the entire data into the RAM** at once, one can load the data in chunks and fit the training model for each chunk of data.
- For machine learning tasks where a new batch of data comes with time, re-training the model with the previous and new batch of data is a computationally expensive process.

> Instead of re-training the model with the entire set of data, one can employ an incremental learning approach, where the model parameters are updated with the new batch of data.



Incremental Learning

Increamental learning may be required in the following two scenarios:

- For **out-of-memory (large) datasets**, where it's not possible to **load the entire data into the RAM** at once, one can load the data in chunks and fit the training model for each chunk of data.
- For machine learning tasks where a new batch of data comes with time, re-training the model with the previous and new batch of data is a computationally expensive process.

> Instead of re-training the model with the entire set of data, one can employ an incremental learning approach, where the model parameters are updated with the new batch of data.

Incremental Learning in `sklearn`

To perform incremental learning, Scikit-learn implements **`partial_fit`** method that helps in training an out-of-memory dataset. In other words, it has the ability to learn incrementally from a batch of instances.

In this colab, we will see an example of how to read, process, and train on such a large dataset that can't be loaded in memory entirely.

This method is expected to be called several times consecutively on different chunks of a dataset so as to implement out-of-core (online) learning. This function has some performance overhead, so it's recommended to call it on a considerable large batch of data (that fits into the memory), to overcome the limitation of overhead.

partial_fit() attributes:

`partial_fit(X, y, [classes], [sample_weight])`

where,

So, large-scale machine learning differs from traditional machine learning, in the sense that it involves processing large amount of data in terms of its size, or number of samples, features and classes. There are many exciting developments in efficient large-scale learning on many real world use cases in the last decade. Although scikit-learn is optimized for smaller data, it does offer a decent set of feature preprocessing and learning algorithms, such as classification regression, and clustering for large-scale data.

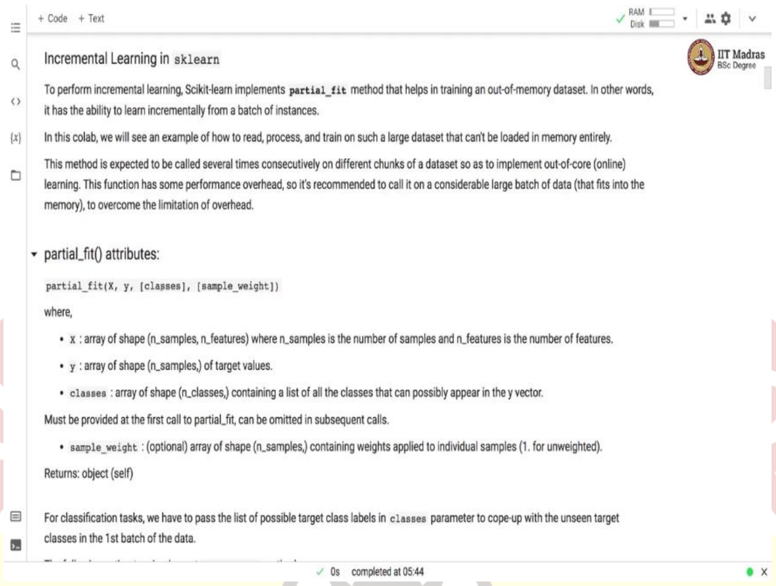
So, one way to check whether there is a support for large-scale dataset in a given estimator or any other preprocessing functions in a scalar, you should check whether that particular API has a partial _fit method. If there is a partial _fit method, there is a good chance that that API will support large-scale learning. So, sklearn handles large-scale data through partial _fit method, instead of using the Fit method.

So, what is the central idea? The central idea is to process data in batches and update the model parameters for every batch. This way of learning is referred to as incremental or out-of-core learning. So, you can see here that we cannot fit the entire data in memory, and call the fit method. So, what we will do is we will load as much of data as we can in memory. And we will use partial _fit method for that.

And what partial _fit method does is it loads a batch of data in memory and performs the parameter estimation, then it loads the next batch in memory and updates the parameters of the model and it keeps doing it until it processes the entire data. So, the incremental learning may be required in the following two scenarios. One is for large datasets or out of memory datasets, where it is not possible to load entire data into the RAM at once. So, one can load data in chunks and fit the machine learning model for each chunk of the data.

And that is another use case for machine learning task where a new batch of data comes with time. So, retraining the model with previous and new batch of data is computationally expensive process. So, instead of returning the model with the entire dataset, one can employ an incremental learning approach where the model parameters are updated with a new batch of data. So, let us look at the incremental learning support in scikit learn.

(Refer Slide Time: 04:58)



Incremental Learning in `sklearn`

To perform incremental learning, Scikit-learn implements `partial_fit` method that helps in training an out-of-memory dataset. In other words, it has the ability to learn incrementally from a batch of instances.

In this colab, we will see an example of how to read, process, and train on such a large dataset that can't be loaded in memory entirely.

This method is expected to be called several times consecutively on different chunks of a dataset so as to implement out-of-core (online) learning. This function has some performance overhead, so it's recommended to call it on a considerable large batch of data (that fits into the memory), to overcome the limitation of overhead.

partial_fit() attributes:

```
partial_fit(X, y, [classes], [sample_weight])
```

where,

- `X` : array of shape (n_samples, n_features) where n_samples is the number of samples and n_features is the number of features.
- `y` : array of shape (n_samples,) of target values.
- `classes` : array of shape (n_classes,) containing a list of all the classes that can possibly appear in the y vector.

Must be provided at the first call to partial_fit, can be omitted in subsequent calls.

- `sample_weight` : (optional) array of shape (n_samples,) containing weights applied to individual samples (1. for unweighted).

Returns: object (self)

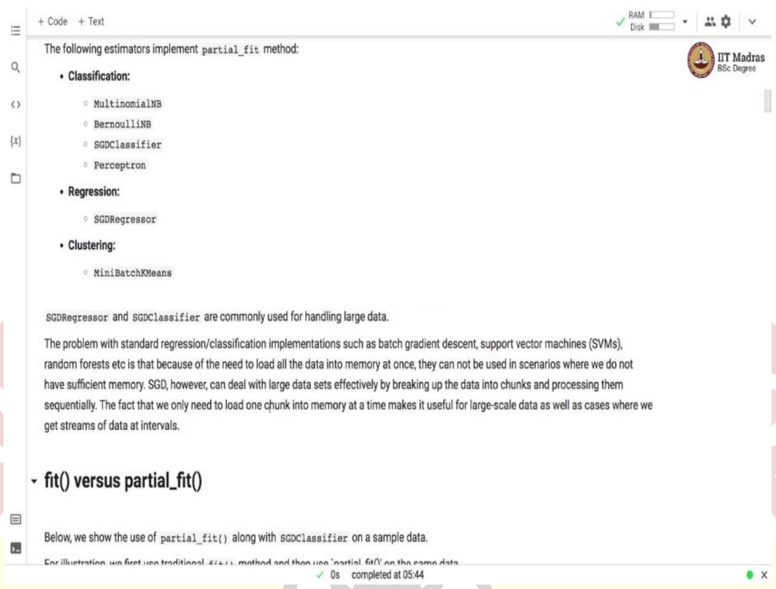
For classification tasks, we have to pass the list of possible target class labels in `classes` parameter to cope-up with the unseen target classes in the 1st batch of the data.

To perform incremental learning scikit learn implements partial _fit method that helps in training and out of memory dataset. So, in other words it has the ability to learn incrementally from the batches of instances. In this collab, we will see an example of how to read process and train on such a large dataset that cannot be loaded in memory entirely.

The partial _fit method is expected to be called several times consecutively on different chunks of dataset so, as to implement out-of-core learning. This function has some performance overhead. So, it is recommended to call it on a considerable large batch of data to overcome the limitations of overhead.

So, let us look at attributes of partial _fit method. So, partial _fit method takes feature matrix label vector, the list of classes and sample weights, as inputs list of classes and sample weights are optional. But, it is recommended that we supply the list of classes very first time when we call the partial _fit method that enables the partial _fit method to have the complete list of classes because since we are processing the data, chunk by chunk, it may happen that in the first chunk certain examples from certain classes might be missing.

(Refer Slide Time: 06:38)



The following estimators implement `partial_fit` method:

- **Classification:**
    - `MultinomialNB`
    - `BernoulliNB`
    - `SGDClassifier`
    - `Perceptron`
- **Regression:**
    - `SGDRegressor`
- **Clustering:**
    - `MiniBatchKMeans`

`SGDRegressor` and `SGDClassifier` are commonly used for handling large data.

The problem with standard regression/classification implementations such as batch gradient descent, support vector machines (SVMs), random forests etc is that because of the need to load all the data into memory at once, they can not be used in scenarios where we do not have sufficient memory. SGD, however, can deal with large data sets effectively by breaking up the data into chunks and processing them sequentially. The fact that we only need to load one chunk into memory at a time makes it useful for large-scale data as well as cases where we get streams of data at intervals.

fit() versus partial_fit()

Below, we show the use of `partial_fit()` along with `SGDClassifier` on a sample data.

For illustration, we first use traditional `fit()` method and then use 'partial_fit()' on the same data.

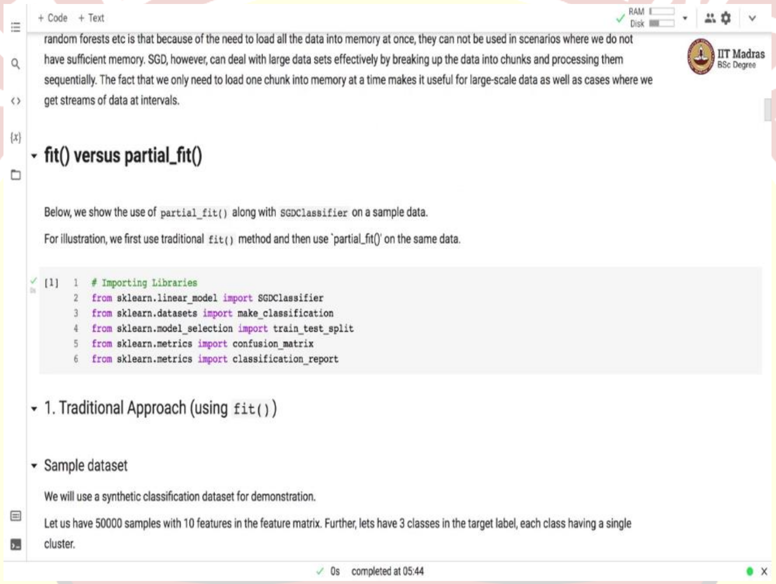
So, the following estimators in sklearn implement partial _fit method. As far as classification is concerned, Multinomial Naive Bayes, Bernoulli Naive Bayes, SGDClassifier and Perceptron implement partial _fit. And you know that with SGDClassifier we can implement different types of classifiers by changing the loss function. SGDRegressor also, implements partial _fit and it can be used for large-scale regression problems.

And for clustering, this is a topic that we will be studying later in the course, there is a Mini BatchKMeans that also, implements partial _fit method. So, SGDRegressor, and SGDClassifier are commonly used for handling large datasets. Now, you may remember our discussion of gradient descent, batch gradient descent and stochastic gradient descent. So, any algorithm that uses let us say full batch gradient descent something like support vector machine random forest, of course, we have to discuss these algorithms they will be coming in subsequent weeks.

So, one of the limitation is that because they are using full batch gradient descent, they need to load all the data in memory before performing any learning. And when we are working with large-scale dataset, loading all the data in memory is not an option because it is just not feasible. On the other hand SGD, if you remember Stochastic gradient descent updates the model parameter with a single example or even mini batch gradient descent uses a small number of examples to update the model parameters.

And the way SGDClassifier and regressors are implemented is that there is a facility to provide the number of examples as argument in SGDClassifier and regressor. So, SGD can deal with large datasets effectively by breaking up the data into chunks and processing them sequentially. So, for SGD we need to load only one chunk into memory at a time and SGD updates the model parameter with that chunk and it goes to the next chunk. So, this is very useful for handling large datasets and also, the data that comes in comes in intervals.

(Refer Slide Time: 09:11)



random forests etc is that because of the need to load all the data into memory at once, they can not be used in scenarios where we do not have sufficient memory. SGD, however, can deal with large data sets effectively by breaking up the data into chunks and processing them sequentially. The fact that we only need to load one chunk into memory at a time makes it useful for large-scale data as well as cases where we get streams of data at intervals.

fit() versus partial_fit()

Below, we show the use of `partial_fit()` along with `SGDClassifier` on a sample data.

For illustration, we first use traditional `fit()` method and then use `partial_fit()` on the same data.

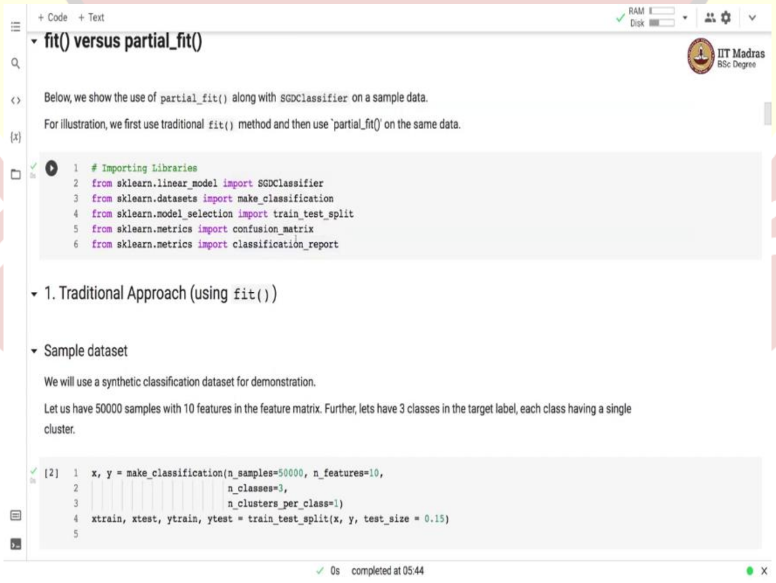
```
# Importing Libraries
from sklearn.linear_model import SGDClassifier
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
```

1. Traditional Approach (using `fit()`)

Sample dataset

We will use a synthetic classification dataset for demonstration.

Let us have 50000 samples with 10 features in the feature matrix. Further, lets have 3 classes in the target label, each class having a single cluster.



fit() versus partial_fit()

Below, we show the use of `partial_fit()` along with `SGDClassifier` on a sample data.

For illustration, we first use traditional `fit()` method and then use `partial_fit()` on the same data.

```
# Importing Libraries
from sklearn.linear_model import SGDClassifier
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
```

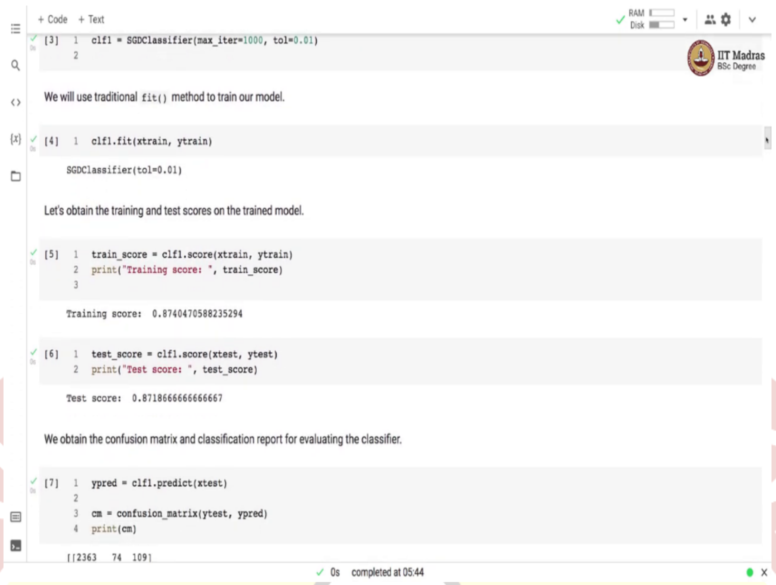
1. Traditional Approach (using `fit()`)

Sample dataset

We will use a synthetic classification dataset for demonstration.

Let us have 50000 samples with 10 features in the feature matrix. Further, lets have 3 classes in the target label, each class having a single cluster.

```
x, y = make_classification(n_samples=50000, n_features=10,
                           n_classes=3,
                           n_clusters_per_class=1)
xtrain, xtest, ytrain, ytest = train_test_split(x, y, test_size = 0.15)
```



```
[3] clf1 = SGDClassifier(max_iter=1000, tol=0.01)
```

We will use traditional `fit()` method to train our model.

```
[4] clf1.fit(xtrain, ytrain)

    SGDClassifier(tol=0.01)
```

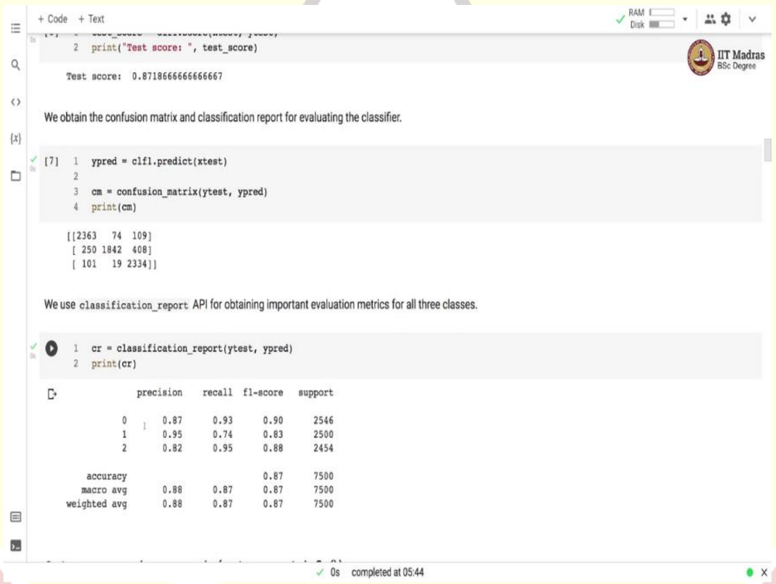
Let's obtain the training and test scores on the trained model.

```
[5] train_score = clf1.score(xtrain, ytrain)
    print("Training score: ", train_score)

    Training score:  0.8740470588235294
```

```
[6] test_score = clf1.score(xtest, ytest)
    print("Test score: ", test_score)

    Test score:  0.8718666666666667
```

We obtain the confusion matrix and classification report for evaluating the classifier.

```
[7] ypred = clf1.predict(xtest)

    cm = confusion_matrix(ytest, ypred)
    print(cm)

    [[2363   74  109]
```



```
    print("Test score: ", test_score)

    Test score:  0.8718666666666667
```

We obtain the confusion matrix and classification report for evaluating the classifier.

```
[7] ypred = clf1.predict(xtest)

    cm = confusion_matrix(ytest, ypred)
    print(cm)

    [[2363   74  109]
     [ 250 1842  408]
     [ 101   19 2334]]
```

We use `classification_report` API for obtaining important evaluation metrics for all three classes.

```
    cr = classification_report(ytest, ypred)
    print(cr)

              precision    recall  f1-score   support

           0       0.87      0.93      0.90      2546
           1       0.95      0.74      0.83      2500
           2       0.82      0.95      0.88      2454

    accuracy                           0.87      7500
   macro avg       0.88      0.87      0.87      7500
weighted avg       0.88      0.87      0.87      7500
```

So, for what we have studied is that there is a partial _fit method that helps us to handle the large datasets and possibly SGD is another powerful method at our disposal for handling large datasets. Let us look at a practical example of fit versus partial _fit. So, here what we do is we will be using a SGDClassifier for a classification problem on a synthetic dataset. And we will show the use of fit and partial _fit and show that both fit and partial _fit leads to similar kind of results.

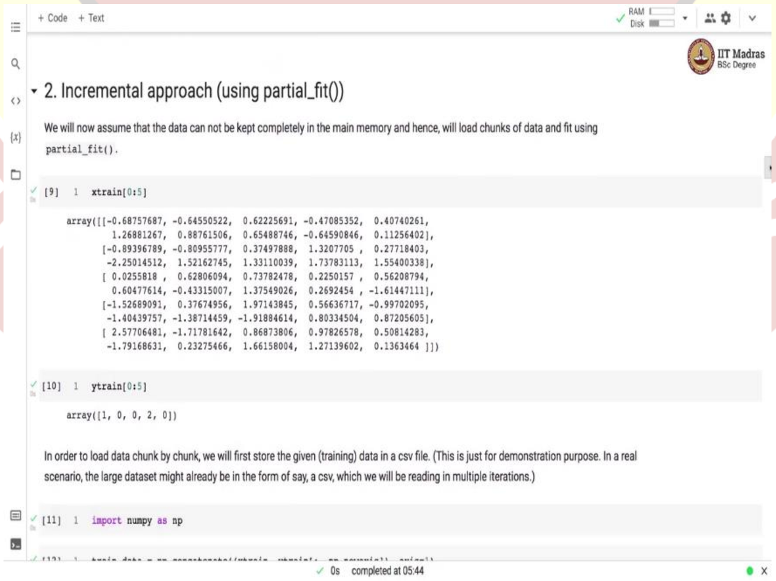
So, you first import the necessary libraries like SGDClassifier, then the dataset creation with make classification API. Then we also, import train _test _split in order to split the dataset into training and test. Then confusion matrix and classification report for evaluation of the model. So, what will first do is we will use first fit but before that, we will generate a sample dataset that has got 50,000 samples with 10 features.

And there are three classes. So, after generating the dataset, we split the data into training and test. So, which we set a size 15% examples for test. We make use of SGDClassifier with maximum number of iterations set to 1000 and tolerance set to 0.01. And you may recall that tolerance is used for deciding the convergence of the SGDClassifier will be using traditional fit method to train our model.

So, here we simply called the fit method with the training feature matrix, and training labels, and we trained our SGDClassifier. So, we obtained the score of 0.87 on the training dataset. And on test data we also obtain very, very, very close test score. As with the training score, both the scores are 0.87. We look at the confusion matrix and classification report on the test set. And we obtained you can see that we obtained their first score of 0.90 for class 0 0.83 for class 1, and 0.88 for class 2. So, overall accuracy is 87%.

Now what we will do is we will use partial _fit method instead of fit method. So, here, we generated a training data and use split method. And this is what we have been doing So, far in this course. Now the change would be that instead of fit, we will be using a partial _fit method.

(Refer Slide Time: 12:26)



2. Incremental approach (using partial_fit())

We will now assume that the data can not be kept completely in the main memory and hence, will load chunks of data and fit using `partial_fit()`.

```
[9]  xtrain[0:5]
```

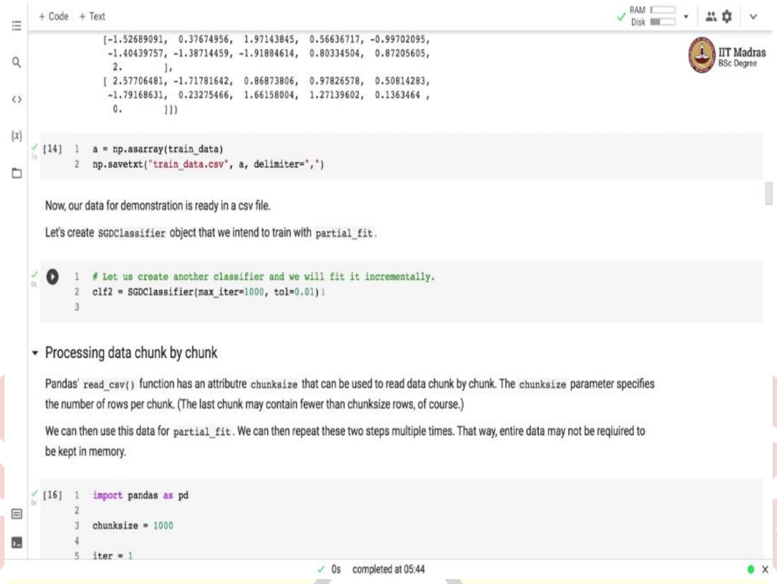
```
array([[-0.68757687, -0.64550522,  0.62225691, -0.47085352,  0.40740261,
         1.26881267,  0.88761506,  0.65488746, -0.64590846,  0.11256402],
       [-0.89396789, -0.80955777,  0.37497888,  1.3207705 ,  0.27718403,
        -2.25014512,  1.52162745,  1.33110039,  1.73783113,  1.55400338],
       [ 0.0255818 ,  0.62806094,  0.73782478,  0.2250157 ,  0.56208794,
         0.60477614, -0.43315007,  1.37549026,  0.2692454 , -1.61447111],
       [-1.52689091,  0.37674956,  1.97143845,  0.56636717, -0.99702095,
        -1.40439757, -1.38714459, -1.91884614,  0.80334504,  0.87205605],
       [ 2.57706481, -1.71781642,  0.86873806,  0.97826578,  0.50814283,
        -1.79168631,  0.23275466,  1.66158004,  1.27139602,  0.1363464 ]])
```

```
[10]  ytrain[0:5]
```

```
array([1, 0, 0, 2, 0])
```

In order to load data chunk by chunk, we will first store the given (training) data in a csv file. (This is just for demonstration purpose. In a real scenario, the large dataset might already be in the form of say, a csv, which we will be reading in multiple iterations.)

```
[11]  import numpy as np
```



```
 [-1.52689091,  0.37674956,  1.97143845,  0.56636717, -0.99702095,
  -1.40439757, -1.38714459, -1.91884614,  0.80334504,  0.87205605,
   2.        ],
 [ 2.57706481, -1.71781642,  0.86873806,  0.97826578,  0.50814283,
  -1.79168631,  0.23275466,  1.66158004,  1.27139602,  0.1363464 ,
   0.        ]])
```

```
[14]
a = np.asarray(train_data)
np.savetxt("train_data.csv", a, delimiter=",")
```

Now, our data for demonstration is ready in a csv file.

Let's create `SGDClassifier` object that we intend to train with `partial_fit`.

```
# Let us create another classifier and we will fit it incrementally.
clf2 = SGDClassifier(max_iter=1000, tol=0.01)

```

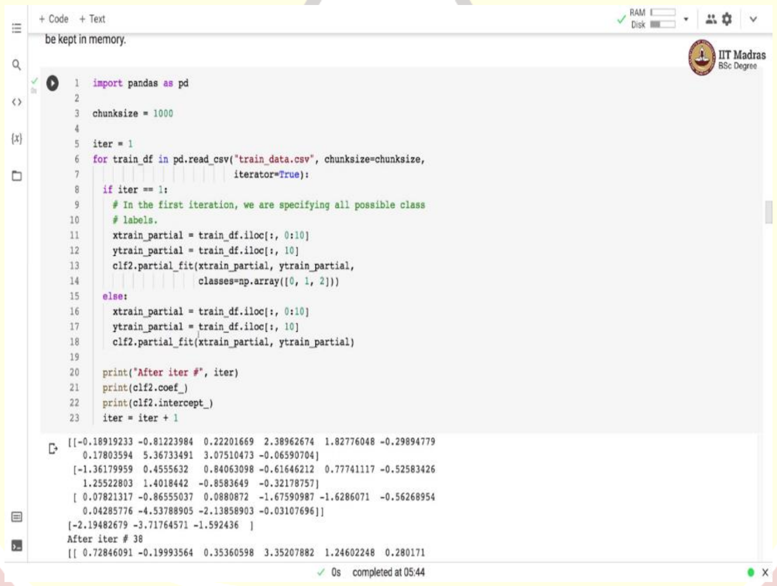
Processing data chunk by chunk

Pandas' `read_csv()` function has an attributre `chunksize` that can be used to read data chunk by chunk. The `chunksize` parameter specifies the number of rows per chunk. (The last chunk may contain fewer than chunksize rows, of course.)

We can then use this data for `partial_fit`. We can then repeat these two steps multiple times. That way, entire data may not be requiired to be kept in memory.

```
[16]
import pandas as pd

chunksize = 1000

iter = 1
```

✓ 0s completed at 05:44

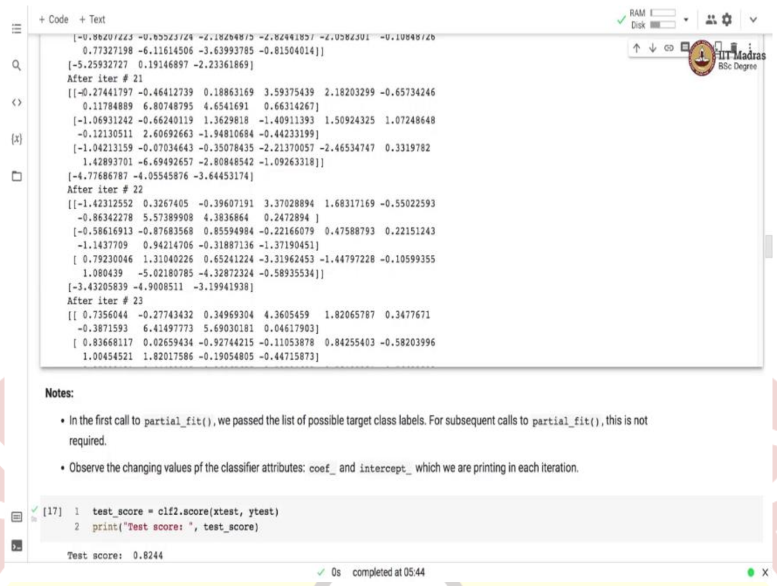


be kept in memory.

```
import pandas as pd

chunksize = 1000

iter = 1
for train_df in pd.read_csv("train_data.csv", chunksize=chunksize,
                            iterator=True):
  if iter == 1:
    # In the first iteration, we are specifying all possible class
    # labels.
    xtrain_partial = train_df.iloc[:, 0:10]
    ytrain_partial = train_df.iloc[:, 10]
    clf2.partial_fit(xtrain_partial, ytrain_partial,
                     classes=np.array([0, 1, 2]))
  else:
    xtrain_partial = train_df.iloc[:, 0:10]
    ytrain_partial = train_df.iloc[:, 10]
    clf2.partial_fit(xtrain_partial, ytrain_partial)

  print("After iter #", iter)
  print(clf2.coef_)
  print(clf2.intercept_)
  iter = iter + 1
```

```
[[-0.18919233 -0.81223984  0.22201669  2.38962674  1.82776048 -0.29894779
   0.17803594  5.36733491  3.07510473 -0.06590704]
 [-1.36179959  0.4555632   0.84063098 -0.61646212  0.77741117 -0.52583426
   1.25522803  1.4018442  -0.8583649  -0.32178757]
 [ 0.07821317 -0.86555037  0.0880872  -1.67590987 -1.6286071  -0.56268954
   0.04285776 -4.53788905 -2.13858903 -0.03107696]]
[-2.19482679 -3.71764571 -1.592436  ]
After iter # 38
[[ 0.72846091 -0.19993564  0.35360598  3.35207882  1.24602248  0.280171
```

✓ 0s completed at 05:44



```
 [-0.86207223 -0.65523724 -2.18264875 -2.82441857 -2.0582301  -0.10848726
   0.77327198 -6.11614506 -3.63993785 -0.81504014]]
[-5.25932727  0.19146897 -2.23361869]
After iter # 21
[[-0.27441797 -0.46412739  0.18863169  3.59375439  2.18203299 -0.65734246
   0.11784889  6.80748795  4.6541691   0.66314267]
 [-1.06931242 -0.66240119  1.3629818  -1.40911393  1.50924325  1.07248648
  -0.12130511  2.60692663 -1.94810684 -0.44233199]
 [-1.04213159 -0.07034643 -0.35078435 -2.21370057 -2.46534747  0.3319782
   1.42893701 -6.69492657 -2.80848542 -1.09263318]]
[-4.77686787 -4.05545876 -3.64453174]
After iter # 22
[[-1.42312552  0.3267405  -0.39607191  3.37028894  1.68317169 -0.55022593
  -0.86342278  5.57389908  4.3836864   0.2472894 ]
 [-0.58616913 -0.87683568  0.85594984 -0.22166079  0.47588793  0.22151243
  -1.1437709   0.94214706 -0.31887136 -1.37190451]
 [ 0.79230046  1.31040226  0.65241224 -3.31962453 -1.44797228 -0.10599355
   1.080439   -5.02180785 -4.32872324 -0.58935534]]
[-3.43205839 -4.9008511  -3.19941938]
After iter # 23
[[ 0.7356044  -0.27743432  0.34969304  4.3605459   1.82065787  0.3477671
  -0.3871593   6.41497773  5.69030181  0.04617903]
 [ 0.83668117  0.02659434 -0.92744215 -0.11053878  0.84255403 -0.58203996
   1.00454521  1.82017586 -0.19054805 -0.44715873]
```

Notes:

- In the first call to `partial_fit()`, we passed the list of possible target class labels. For subsequent calls to `partial_fit()`, this is not required.
- Observe the changing values pf the classifier attributes: `coef_` and `intercept_` which we are printing in each iteration.

```
[17]
test_score = clf2.score(xtest, ytest)
print("Test score: ", test_score)
```

```
Test score:  0.8244
```

So, you may recall from our discussion, that partial _fit basically loads a small chunk of data in memory, processes them. And in processing, what it does is it actually trains the model on that chunk of data. Then it loads the next chunk and update the model parameters. So, this is how partial _fit method works. So, what we will do is, we will pretend that this dataset that we have generated, which has got 50,000 sample is we just cannot load the entire data in memory.

So, what we will do is for that, we will load this data in chunks in memory, and then call the partial _fit method for training the model. So, let us see how to do that. So, we first concatenate the entire training feature metrics and training label into what is called as train data. And we store this train data into a CSV file. Now the data is in the file. And what we will do is we will read this file chunk by chunk and call the partial _fit method.

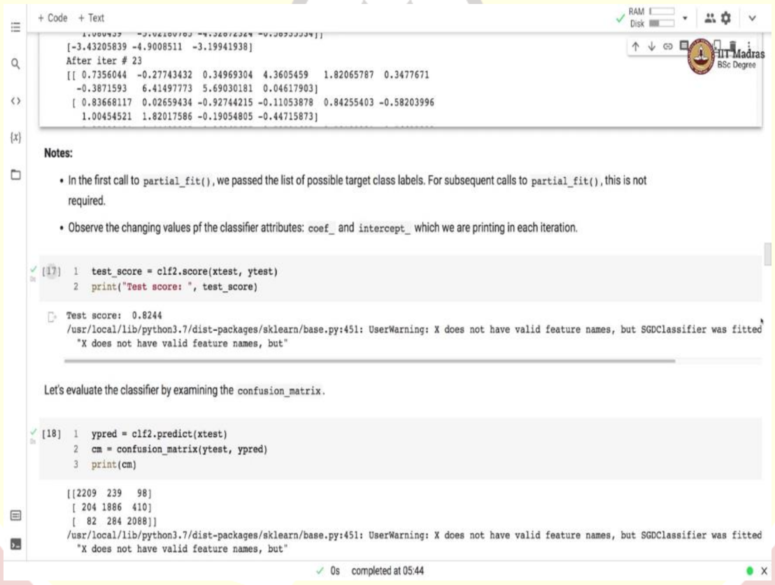
So, we instantiate an SGDClassifier with maximum iteration, set to 1000 and tolerance set to 0.01. And this is where we are processing data chunk by chunk. So, we are going to use the read _csv function in pandas that has got attribute called chunksize. And the chunksize parameter specifies the number of rows per chunk that we want to that we want to read. So, here we set the chunk size 1000.

So, we are going to read 1000 examples at a time and then call the partial _fit method on those 1000 examples. So, remember that while performing or while calling the partial _fit method for the first time, we need to also supply the list of classes that we have done here. That is why we have a conditional statement here. So, if the iteration number is 1, then we call partial _fit with the list of classes.

And for the subsequent iterations, we do not need to specify the list of classes and we simply specify the feature matrix and label vector. And after every iteration we are printing the coefficients and the intercepts that are obtained by the partial _fit method. So, here there are you can see that after every iteration we are printing the intercept. So, there are three classes. So, there are three intercepts, and there are three weight vectors one per each class.

So, you can see that after every iteration the weight vector is kind of changing and also, the intercepts are changing.

(Refer Slide Time: 15:38)



```
 [-3.43205839 -4.9008511  -3.19941938]
After iter # 23
[[ 0.7356044  -0.27743432  0.34969304  4.3605459   1.82065787  0.3477671
  -0.3871593   6.41497773  5.69030181  0.04617903]
 [ 0.83668117  0.02659434 -0.92744215 -0.11053878  0.84255403 -0.58203996
   1.00454521  1.82017586 -0.19054805 -0.44715873]
```

Notes:

- In the first call to `partial_fit()`, we passed the list of possible target class labels. For subsequent calls to `partial_fit()`, this is not required.
- Observe the changing values pf the classifier attributes: `coef_` and `intercept_` which we are printing in each iteration.

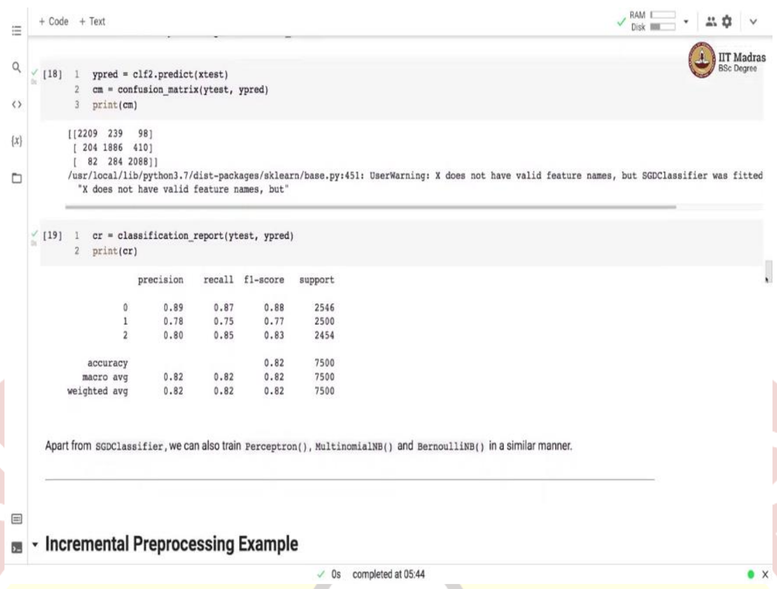
```
[17]
test_score = clf2.score(xtest, ytest)
print("Test score: ", test_score)
```

```
Test score:  0.8244
/usr/local/lib/python3.7/dist-packages/sklearn/base.py:451: UserWarning: X does not have valid feature names, but SGDClassifier was fitted
  "X does not have valid feature names, but"
```

Let's evaluate the classifier by examining the `confusion_matrix`.

```
[18]
ypred = clf2.predict(xtest)
cm = confusion_matrix(ytest, ypred)
print(cm)
```

```
[[2209  239   98]
 [ 204 1886  410]
 [  82  284 2088]]
/usr/local/lib/python3.7/dist-packages/sklearn/base.py:451: UserWarning: X does not have valid feature names, but SGDClassifier was fitted
  "X does not have valid feature names, but"
```



```
[18] ypred = clf2.predict(xtest)
     cm = confusion_matrix(ytest, ypred)
     print(cm)

[[2209  239   98]
 [ 204 1886  410]
 [  82  284 2088]]
/usr/local/lib/python3.7/dist-packages/sklearn/base.py:451: UserWarning: X does not have valid feature names, but SGDClassifier was fitted
  "X does not have valid feature names, but"
```

```
[19] cr = classification_report(ytest, ypred)
     print(cr)

              precision    recall  f1-score   support

           0       0.89      0.87      0.88      2546
           1       0.78      0.75      0.77      2500
           2       0.80      0.85      0.83      2454

    accuracy                           0.82      7500
   macro avg       0.82      0.82      0.82      7500
weighted avg       0.82      0.82      0.82      7500
```

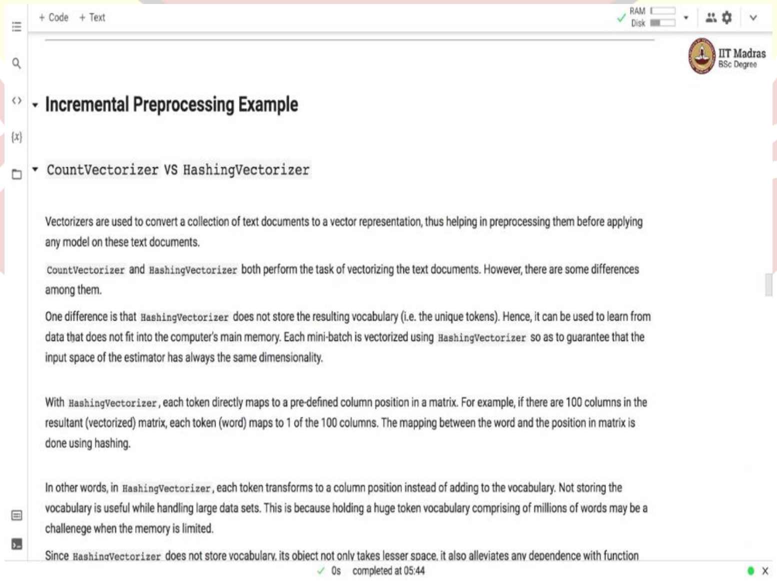
Apart from `SGDClassifier`, we can also train `Perceptron()`, `MultinomialNB()` and `BernoulliNB()` in a similar manner.

Incremental Preprocessing Example

Now, after the training is complete we obtain the score on the test set it is 0.82 which is lower than what we obtained from the fit method earlier and f1-scores are also, slightly lower than what we obtained earlier we obtained f1-score on the class 0 the f1-score 0.88 for class 1 it is 0.77 and for class 2 it is 0.83 the overall accuracy is 82%.

So, this was a small demonstration of how to use partial _fit method instead of the fit method. So, apart from SGDClassifier, we can also, train perceptron Multinomial Naive Bayes and Bernoulli Naive Bayes in a similar manner.

(Refer Slide Time: 16:28)



Incremental Preprocessing Example

`CountVectorizer` VS `HashingVectorizer`

Vectorizers are used to convert a collection of text documents to a vector representation, thus helping in preprocessing them before applying any model on these text documents.

`CountVectorizer` and `HashingVectorizer` both perform the task of vectorizing the text documents. However, there are some differences among them.

One difference is that `HashingVectorizer` does not store the resulting vocabulary (i.e. the unique tokens). Hence, it can be used to learn from data that does not fit into the computer's main memory. Each mini-batch is vectorized using `HashingVectorizer` so as to guarantee that the input space of the estimator has always the same dimensionality.

With `HashingVectorizer`, each token directly maps to a pre-defined column position in a matrix. For example, if there are 100 columns in the resultant (vectorized) matrix, each token (word) maps to 1 of the 100 columns. The mapping between the word and the position in matrix is done using hashing.

In other words, in `HashingVectorizer`, each token transforms to a column position instead of adding to the vocabulary. Not storing the vocabulary is useful while handling large data sets. This is because holding a huge token vocabulary comprising of millions of words may be a challenege when the memory is limited.

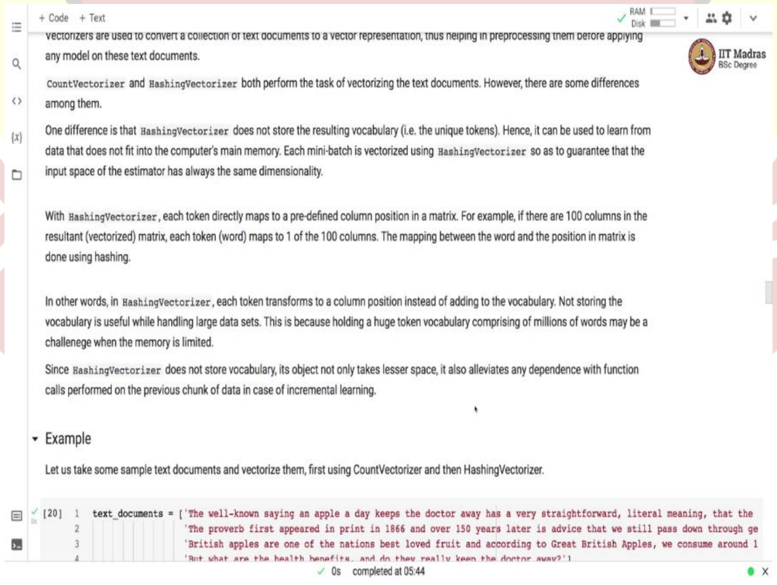
Since `HashingVectorizer` does not store vocabulary, its object not only takes lesser space, it also alleviates any dependence with function

So, far we have seen how to train a model in stepwise manner. Now, we will see how to perform preprocessing in stepwise or in incremental manner. So, here we show a couple of preprocessing techniques, which is CountVectorizer and HashingVectorizer. So, vectorizer used to convert a collection of text documents into a vector representation. So, this is a preprocessing step mainly in case of text data.

So, CountVectorizer and HashingVectorizer both perform task of vectorizing the text document there is some differences between them. So, one difference is that HashingVectorizer does not have or does not store the resulting vocabulary. Hence, it is used to learn from the data that does not fit into the memory. So, each mini batch is vectorized using HashingVectorizer. So, as to guarantee that the input space of the estimator has always the same number of dimensions.

With HashingVectorizer each token directly maps to a predefined column position in the matrix. So, for example, if there are 100 columns in the resultant matrix, each token would get mapped to one of the 100 columns, the mapping between the word and the position in the matrix is done using hashing. So, as the name suggests, it is HashingVectorizer. So, hashing, which is a technique that you must have studied in the data structure is used for generating a vector representation for the text data.

(Refer Slide Time: 18:15)



Vectorizers are used to convert a collection of text documents to a vector representation, thus helping in preprocessing them before applying any model on these text documents.

`CountVectorizer` and `HashingVectorizer` both perform the task of vectorizing the text documents. However, there are some differences among them.

One difference is that `HashingVectorizer` does not store the resulting vocabulary (i.e. the unique tokens). Hence, it can be used to learn from data that does not fit into the computer's main memory. Each mini-batch is vectorized using `HashingVectorizer` so as to guarantee that the input space of the estimator has always the same dimensionality.

With `HashingVectorizer`, each token directly maps to a pre-defined column position in a matrix. For example, if there are 100 columns in the resultant (vectorized) matrix, each token (word) maps to 1 of the 100 columns. The mapping between the word and the position in matrix is done using hashing.

In other words, in `HashingVectorizer`, each token transforms to a column position instead of adding to the vocabulary. Not storing the vocabulary is useful while handling large data sets. This is because holding a huge token vocabulary comprising of millions of words may be a challenege when the memory is limited.

Since `HashingVectorizer` does not store vocabulary, its object not only takes lesser space, it also alleviates any dependence with function calls performed on the previous chunk of data in case of incremental learning.

Example

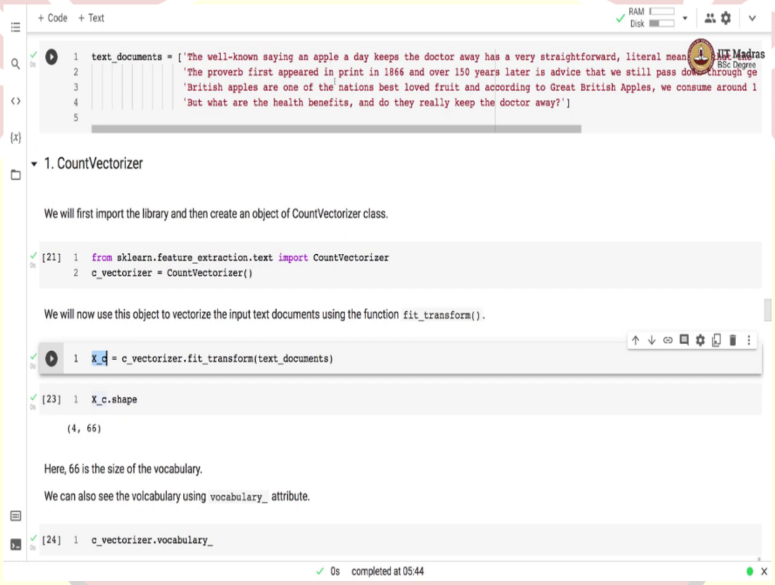
Let us take some sample text documents and vectorize them, first using CountVectorizer and then HashingVectorizer.

```
[20] text_documents = ['The well-known saying an apple a day keeps the doctor away has a very straightforward, literal meaning, that the
                      'The proverb first appeared in print in 1866 and over 150 years later is advice that we still pass down through ge
                      'British apples are one of the nations best loved fruit and according to Great British Apples, we consume around 1
                      'But what are the health benefits, and do they really keep the doctor away?']
```

So, HashingVectorizer, what happens is each token gets transformed into a column position. And we do not really meant in a vocabulary. And not storing vocabulary is very useful while handling large datasets, because we just cannot hold a huge token vocabulary comprising of millions of words into memory since the memory is limited. So, since HashingVectorizer does not store vocabulary, it does not take much space, unlike the CountVectorizer.

So, it also, alleviates any dependency with function calls performed on the previous chunk of data while performing incremental learning. So, HashingVectorizer is kind of another friend of ours when processing large-scale text datasets.

(Refer Slide Time: 19:06)



```
text_documents = ['The well-known saying an apple a day keeps the doctor away has a very straightforward, literal mean
                  'The proverb first appeared in print in 1866 and over 150 years later is advice that we still pass down through ge
                  'British apples are one of the nations best loved fruit and according to Great British Apples, we consume around 1
                  'But what are the health benefits, and do they really keep the doctor away?']
```

1. CountVectorizer

We will first import the library and then create an object of CountVectorizer class.

```
[21] from sklearn.feature_extraction.text import CountVectorizer
     c_vectorizer = CountVectorizer()
```

We will now use this object to vectorize the input text documents using the function `fit_transform()`.

```
X_c = c_vectorizer.fit_transform(text_documents)
```

```
[23] X_c.shape
```

```
(4, 66)
```

Here, 66 is the size of the vocabulary.

We can also see the volcabulary using `vocabulary_` attribute.

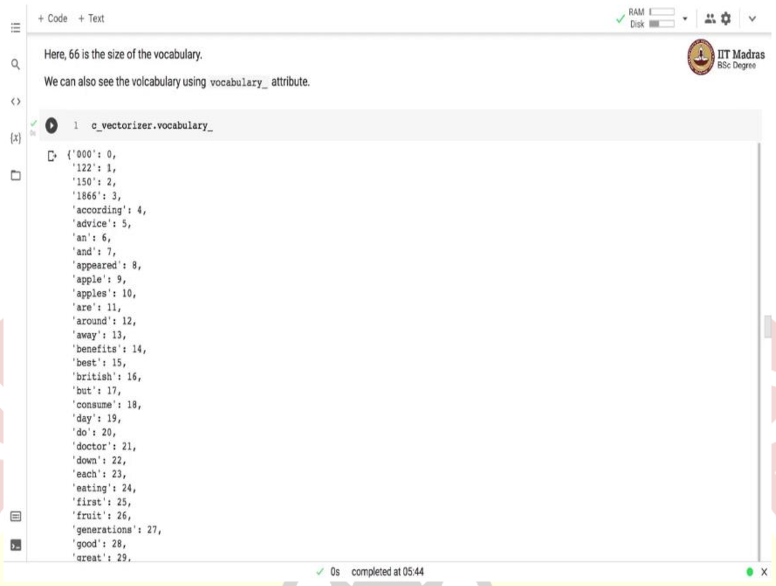
```
[24] c_vectorizer.vocabulary_
```

So, now what we will do is we will take some sample text documents and vectorize them using both CountVectorizer and HashingVectorizer. So, that you get an idea of how both these vectorizer work. So, these are 4 small text snippets. And what we will be doing is we will be first using a CountVectorizer and CountVectorizer is a preprocessing technique. And it is part of sklearn.feature _extraction.text module.

So, we instantiate a count vector a CountVectorizer and then called fit _transform method. Now remember that this is a preprocessing technique. And each processing API has fit _transform method implemented. So, we pass the list of text documents to the fit _transform method. And we obtain we obtained a transformed representations or the vector representation of the documents.

So, you can see that the shape of the, the feature, the feature matrix is $\frac{4}{66}$. So, 4 is the number of text document and 66 is the size of vocabulary, there are 66 unique tokens in these 4 documents.

(Refer Slide Time: 20:25)

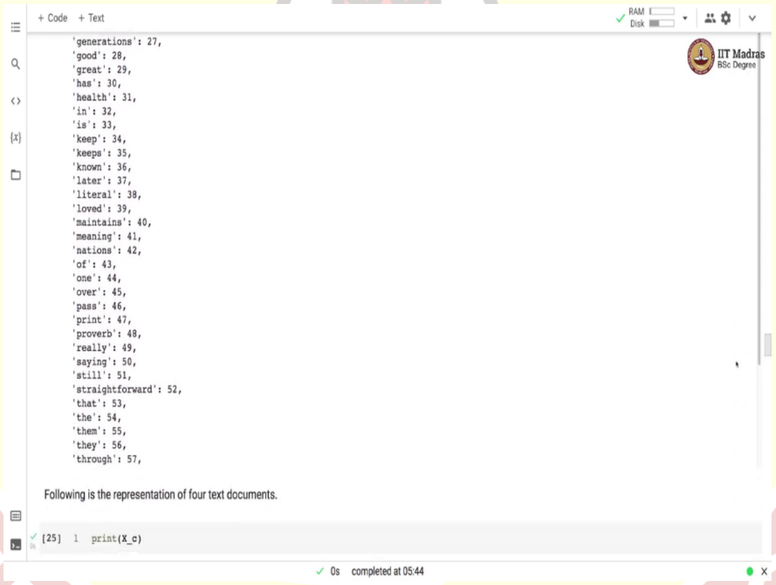


Here, 66 is the size of the vocabulary.

We can also see the volcabulary using `vocabulary_` attribute.

```
c_vectorizer.vocabulary_
```

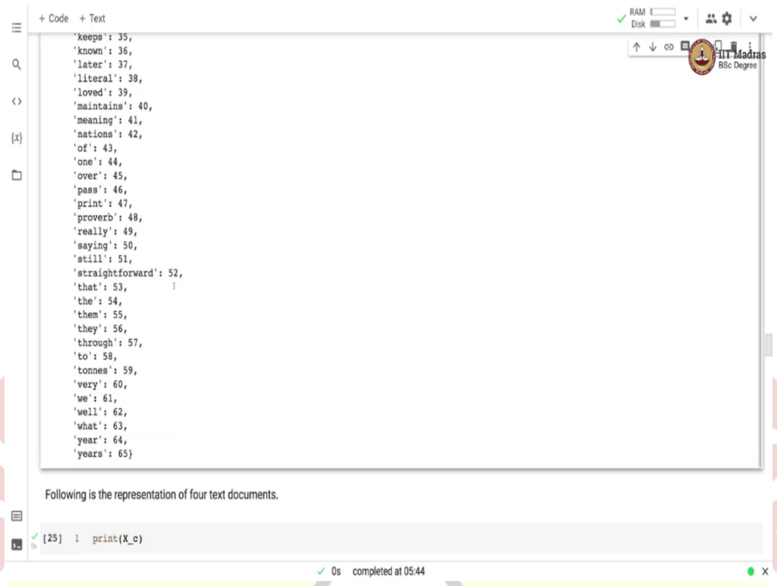
```
{'000': 0,
 '122': 1,
 '150': 2,
 '1866': 3,
 'according': 4,
 'advice': 5,
 'an': 6,
 'and': 7,
 'appeared': 8,
 'apple': 9,
 'apples': 10,
 'are': 11,
 'around': 12,
 'away': 13,
 'benefits': 14,
 'best': 15,
 'british': 16,
 'but': 17,
 'consume': 18,
 'day': 19,
 'do': 20,
 'doctor': 21,
 'down': 22,
 'each': 23,
 'eating': 24,
 'first': 25,
 'fruit': 26,
 'generations': 27,
 'good': 28,
 'great': 29,
```



```
 'generations': 27,
 'good': 28,
 'great': 29,
 'has': 30,
 'health': 31,
 'in': 32,
 'is': 33,
 'keep': 34,
 'keeps': 35,
 'known': 36,
 'later': 37,
 'literal': 38,
 'loved': 39,
 'maintains': 40,
 'meaning': 41,
 'nations': 42,
 'of': 43,
 'one': 44,
 'over': 45,
 'pass': 46,
 'print': 47,
 'proverb': 48,
 'really': 49,
 'saying': 50,
 'still': 51,
 'straightforward': 52,
 'that': 53,
 'the': 54,
 'them': 55,
 'they': 56,
 'through': 57,
```

Following is the representation of four text documents.

```
[25] print(X_c)
```



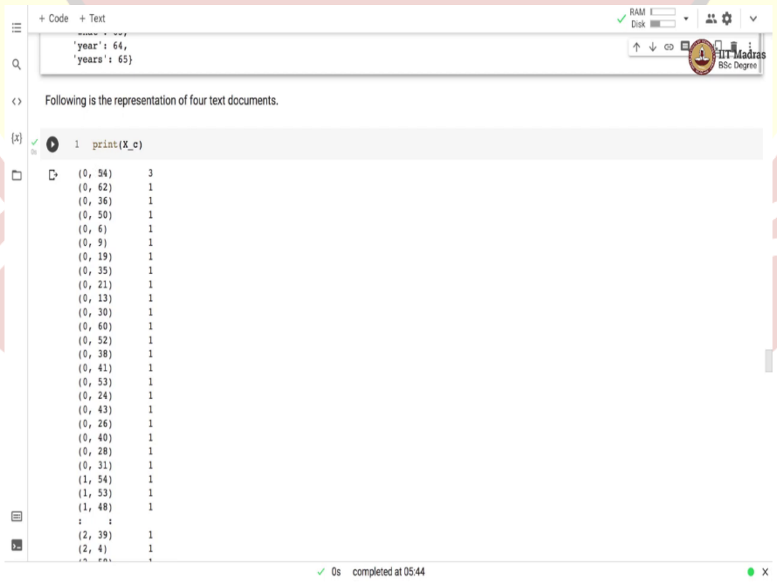
```
 'keeps': 35,
 'known': 36,
 'later': 37,
 'literal': 38,
 'loved': 39,
 'maintains': 40,
 'meaning': 41,
 'nations': 42,
 'of': 43,
 'one': 44,
 'over': 45,
 'pass': 46,
 'print': 47,
 'proverb': 48,
 'really': 49,
 'saying': 50,
 'still': 51,
 'straightforward': 52,
 'that': 53,
 'the': 54,
 'them': 55,
 'they': 56,
 'through': 57,
 'to': 58,
 'tonnes': 59,
 'very': 60,
 'we': 61,
 'well': 62,
 'what': 63,
 'year': 64,
 'years': 65}
```

Following is the representation of four text documents.

```
[25] print(X_c)
```

So, here, the vocabulary is present in the dictionary format, where the key of the dictionary is a token. And the value is the ID assigned to this particular token. So, 000 has the ID 0, 122 has ID of 1, 150 is has the ID of 2, and so, on. So, we have 66 such kind of tokens. And we will assign IDs from 0 to 65. And we can also, see that these tokens are in alphabetical order. So, let us see how the text documents are represented. So, this is our text documents for text documents, and we will see how they are represented with CountVectorizer.

(Refer Slide Time: 21:17)

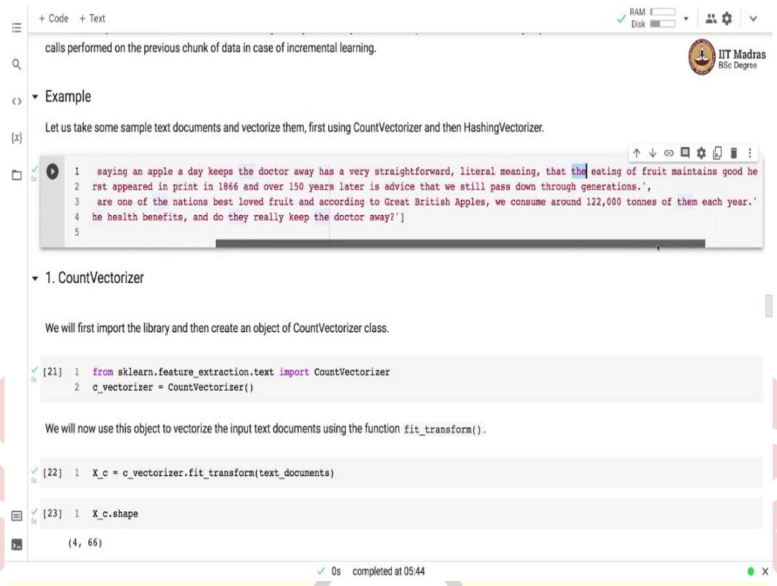


```
 'year': 64,
 'years': 65}
```

Following is the representation of four text documents.

```
print(X_c)
```

```
  (0, 54)	3
  (0, 62)	1
  (0, 36)	1
  (0, 50)	1
  (0, 6)	1
  (0, 9)	1
  (0, 19)	1
  (0, 35)	1
  (0, 21)	1
  (0, 13)	1
  (0, 30)	1
  (0, 60)	1
  (0, 52)	1
  (0, 38)	1
  (0, 41)	1
  (0, 53)	1
  (0, 24)	1
  (0, 43)	1
  (0, 26)	1
  (0, 40)	1
  (0, 28)	1
  (0, 31)	1
  (1, 54)	1
  (1, 53)	1
  (1, 48)	1
  :	:
  (2, 39)	1
  (2, 4)	1
```


calls performed on the previous chunk of data in case of incremental learning.

Example

Let us take some sample text documents and vectorize them, first using CountVectorizer and then HashingVectorizer.

```
 saying an apple a day keeps the doctor away has a very straightforward, literal meaning, that the eating of fruit maintains good he
rst appeared in print in 1866 and over 150 years later is advice that we still pass down through generations.',
 are one of the nations best loved fruit and according to Great British Apples, we consume around 122,000 tonnes of them each year.'
he health benefits, and do they really keep the doctor away?']

```

1. CountVectorizer

We will first import the library and then create an object of CountVectorizer class.

```
[21] from sklearn.feature_extraction.text import CountVectorizer
     c_vectorizer = CountVectorizer()
```

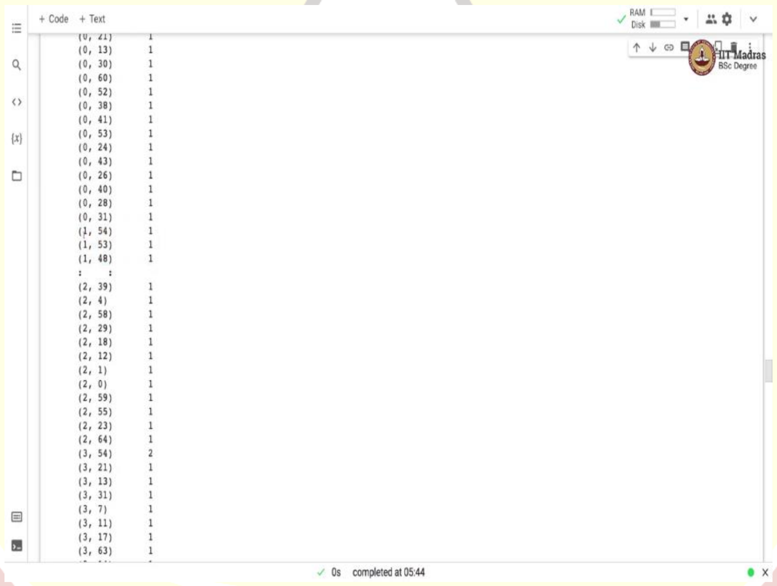
We will now use this object to vectorize the input text documents using the function `fit_transform()`.

```
[22] X_c = c_vectorizer.fit_transform(text_documents)
```

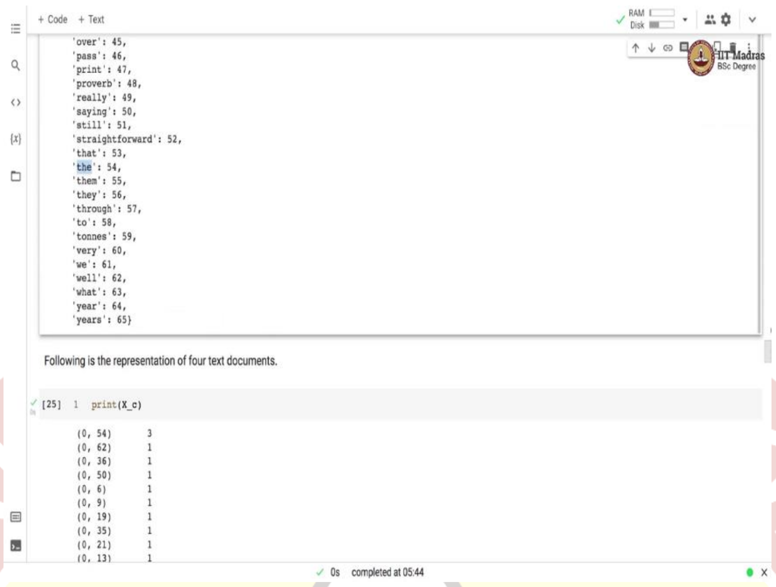
```
[23] X_c.shape
```

```
(4, 66)
```

✓ 0s completed at 05:44


```
  (0, 21)	1
  (0, 13)	1
  (0, 30)	1
  (0, 60)	1
  (0, 52)	1
  (0, 38)	1
  (0, 41)	1
  (0, 53)	1
  (0, 24)	1
  (0, 43)	1
  (0, 26)	1
  (0, 40)	1
  (0, 28)	1
  (0, 31)	1
  (1, 54)	1
  (1, 53)	1
  (1, 48)	1
  :	:
  (2, 39)	1
  (2, 4)	1
  (2, 58)	1
  (2, 29)	1
  (2, 18)	1
  (2, 12)	1
  (2, 1)	1
  (2, 0)	1
  (2, 59)	1
  (2, 55)	1
  (2, 23)	1
  (2, 64)	1
  (3, 54)	2
  (3, 21)	1
  (3, 13)	1
  (3, 31)	1
  (3, 7)	1
  (3, 11)	1
  (3, 17)	1
  (3, 63)	1
```

✓ 0s completed at 05:44


```
 'over': 45,
 'pass': 46,
 'print': 47,
 'proverb': 48,
 'really': 49,
 'saying': 50,
 'still': 51,
 'straightforward': 52,
 'that': 53,
 'the': 54,
 'them': 55,
 'they': 56,
 'through': 57,
 'to': 58,
 'tonnes': 59,
 'very': 60,
 'we': 61,
 'well': 62,
 'what': 63,
 'year': 64,
 'years': 65}
```

Following is the representation of four text documents.

```
[25] print(X_c)
```

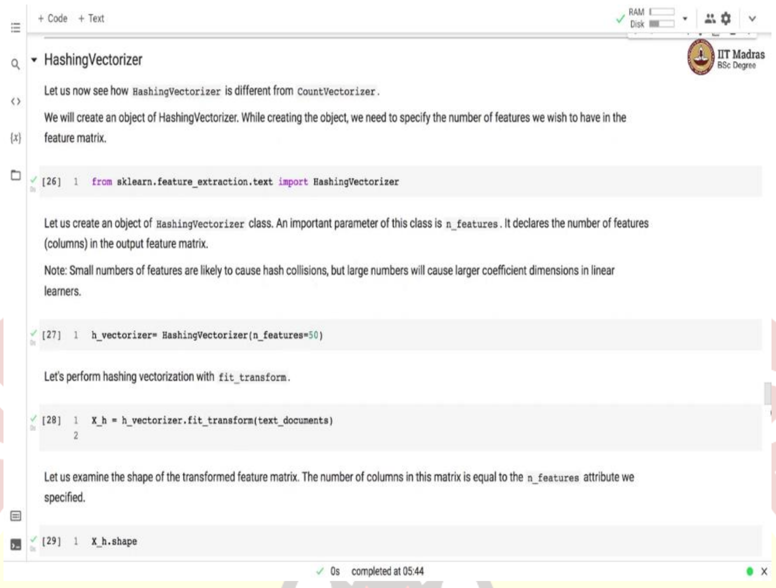
```
  (0, 54)	3
  (0, 62)	1
  (0, 36)	1
  (0, 50)	1
  (0, 6)	1
  (0, 9)	1
  (0, 19)	1
  (0, 35)	1
  (0, 21)	1
  (0, 13)	1
```

So, we have what is called as a sparse matrix representation. So, we do not we do not show or we do not store all the 65 dimensions for each tokens, because not all 65 tokens will be present in a document. So, we only store the index of the tokens or index of the words that are present in the document for example, document 1 has got word or token with index 54. We can check it out what is the 54 token it is the it is a word 'the'.

So, this word 'the' has come three times we can go here and actually check whether 'the' has come three times there is one third here second 'the' here, and there is a third 'the' over here. So, indeed, 'the' has come three times in the first document. And hence, you can So, this is the count of the word 'the'. In the same way the word 62 the word ID 62, which is word 'well' has come once in the document. Which we can go back and check again. So, there is word 'well', that has come once in the first document. So, you can see that count over here and so, on.

So, this is how the documents are represented this is a sparse matrix representation, where we only store the indices of words as part of this representation. Similarly, we have the representation for the second document, then for the third document, as well as for the first document. And so we are not actually showing the entire presentation, because this listing would be too big. So, only few words in the second documents are shown here.

(Refer Slide Time: 23:30)



HashingVectorizer

Let us now see how `HashingVectorizer` is different from `CountVectorizer`.

We will create an object of HashingVectorizer. While creating the object, we need to specify the number of features we wish to have in the feature matrix.

```
[26]  from sklearn.feature_extraction.text import HashingVectorizer
```

Let us create an object of `HashingVectorizer` class. An important parameter of this class is `n_features`. It declares the number of features (columns) in the output feature matrix.

Note: Small numbers of features are likely to cause hash collisions, but large numbers will cause larger coefficient dimensions in linear learners.

```
[27]  h_vectorizer= HashingVectorizer(n_features=50)
```

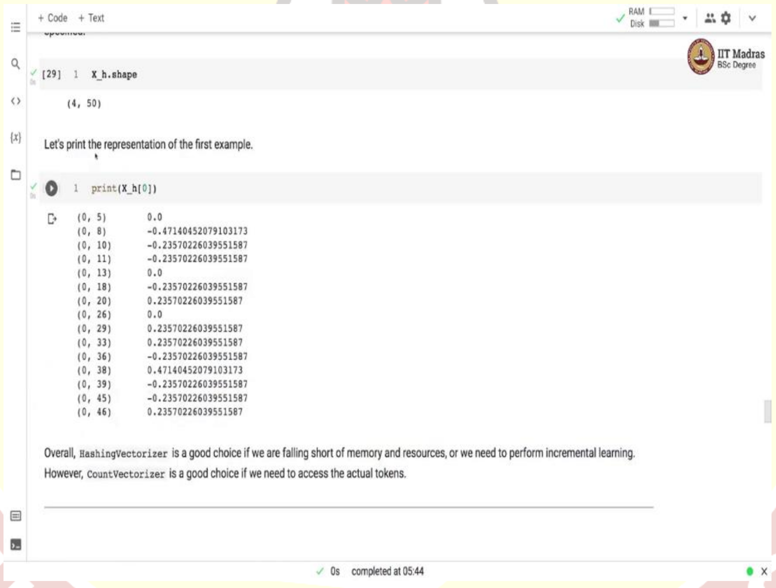
Let's perform hashing vectorization with `fit_transform`.

```
[28]  X_h = h_vectorizer.fit_transform(text_documents)
```

Let us examine the shape of the transformed feature matrix. The number of columns in this matrix is equal to the `n_features` attribute we specified.

```
[29]  X_h.shape
```

0s completed at 05:44



```
[29]  X_h.shape

(4, 50)
```

Let's print the representation of the first example.

```
print(X_h[0])

  (0, 5)	0.0
  (0, 8)	-0.47140452079103173
  (0, 10)	-0.23570226039551587
  (0, 11)	-0.23570226039551587
  (0, 13)	0.0
  (0, 18)	-0.23570226039551587
  (0, 20)	0.23570226039551587
  (0, 26)	0.0
  (0, 29)	0.23570226039551587
  (0, 33)	0.23570226039551587
  (0, 36)	-0.23570226039551587
  (0, 38)	0.47140452079103173
  (0, 39)	-0.23570226039551587
  (0, 45)	-0.23570226039551587
  (0, 46)	0.23570226039551587
```

Overall, `HashingVectorizer` is a good choice if we are falling short of memory and resources, or we need to perform incremental learning. However, `CountVectorizer` is a good choice if we need to access the actual tokens.

0s completed at 05:44

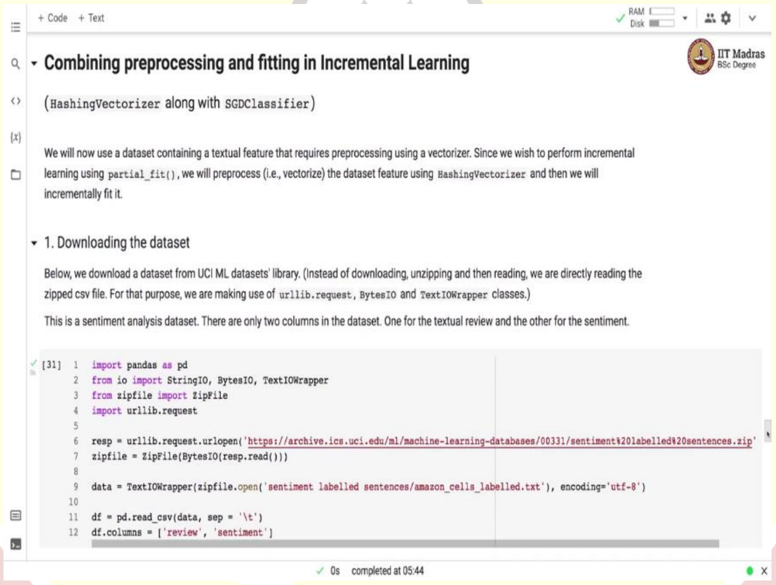
Now, let us use HashingVectorizer instead of CountVectorizer for vectorizing the documents. So, we here we import the HashingVectorizer from sklearn.feature _extraction.text module. So, we instantiate an object of HashingVectorizer class. There is an important parameter which is n _features that declares a number of features in the output feature matrix we set it to 50.

Now remember that do not set too small or too large number over here. Too small number will cause hash collisions and which will pollute the representation. But, if you set this to a very large number, it will cause large number of feature dimensions for each text document. Now let us perform HashingVectorizer with fit _transform method. So, we call fit _transform method on HashingVectorizer by supplying the text documents.

Let us examine the shape of transform feature matrix. So, there are 4 documents and each document is represented with 50 values and this 50 is equal to the n _feature attribute that we had specified. So, this is a sample representation for first text document. So, for every text document, we get a score from the from the HashingVectorizer.

So, overall HashingVectorizer is a good choice if we are falling short of memory and resources or we need to perform incremental learning. However, CountVectorizer is a good choice if we need to access the actual tokens.

(Refer Slide Time: 25:29)



Combining preprocessing and fitting in Incremental Learning

(`HashingVectorizer` along with `SGDClassifier`)

We will now use a dataset containing a textual feature that requires preprocessing using a vectorizer. Since we wish to perform incremental learning using `partial_fit()`, we will preprocess (i.e., vectorize) the dataset feature using `HashingVectorizer` and then we will incrementally fit it.

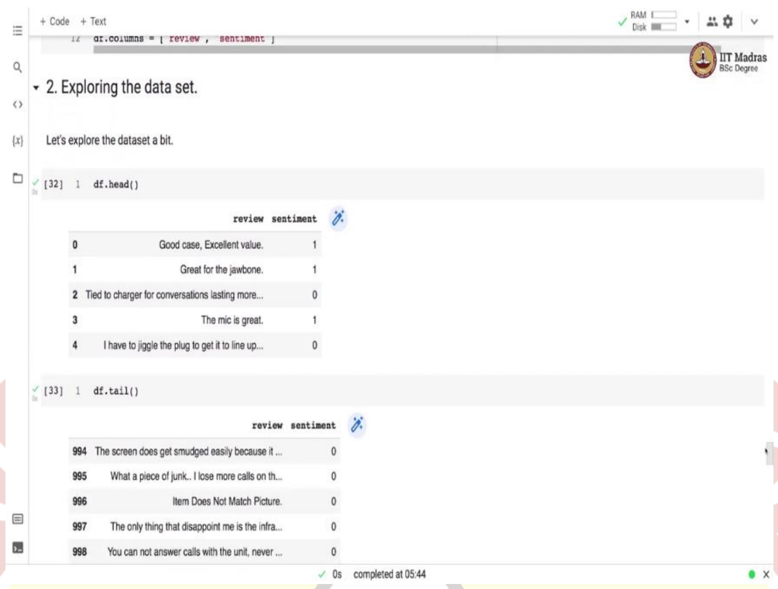
1. Downloading the dataset

Below, we download a dataset from UCI ML datasets' library. (Instead of downloading, unzipping and then reading, we are directly reading the zipped csv file. For that purpose, we are making use of `urllib.request`, `BytesIO` and `TextIOWrapper` classes.)

This is a sentiment analysis dataset. There are only two columns in the dataset. One for the textual review and the other for the sentiment.

```
[31]
import pandas as pd
from io import StringIO, BytesIO, TextIOWrapper
from zipfile import ZipFile
import urllib.request

resp = urllib.request.urlopen('https://archive.ics.uci.edu/ml/machine-learning-databases/00331/sentiment%20labelled%20sentences.zip'
zipfile = ZipFile(BytesIO(resp.read()))

data = TextIOWrapper(zipfile.open('sentiment labelled sentences/amazon_cells_labelled.txt'), encoding='utf-8')

df = pd.read_csv(data, sep = '\t')
df.columns = ['review', 'sentiment']
```

✓ 0s completed at 05:44



```
df.columns = ['review', 'sentiment']
```

2. Exploring the data set.

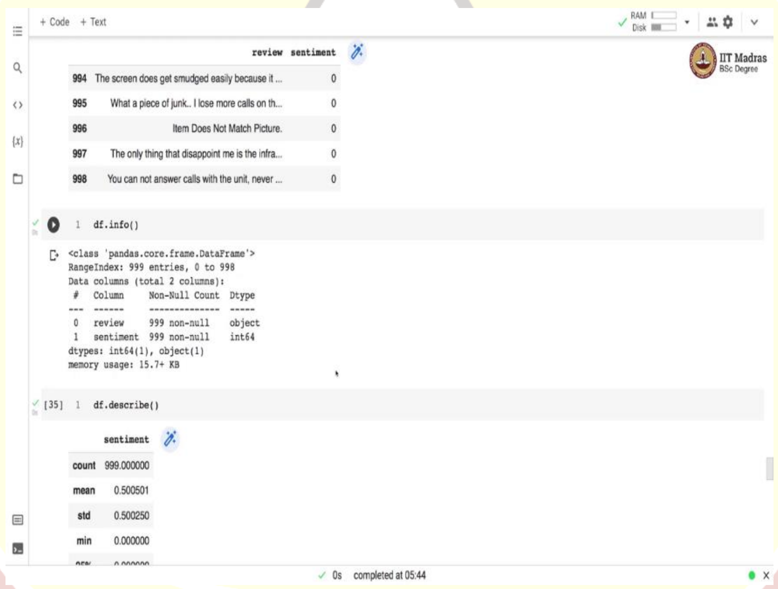
Let's explore the dataset a bit.

```
[32] df.head()
```

	review	sentiment
0	Good case, Excellent value.	1
1	Great for the jawbone.	1
2	Tied to charger for conversations lasting more...	0
3	The mic is great.	1
4	I have to jiggle the plug to get it to line up...	0

```
[33] df.tail()
```

	review	sentiment
994	The screen does get smudged easily because it ...	0
995	What a piece of junk.. I lose more calls on th...	0
996	Item Does Not Match Picture.	0
997	The only thing that disappoint me is the infra...	0
998	You can not answer calls with the unit, never ...	0



	review	sentiment
994	The screen does get smudged easily because it ...	0
995	What a piece of junk.. I lose more calls on th...	0
996	Item Does Not Match Picture.	0
997	The only thing that disappoint me is the infra...	0
998	You can not answer calls with the unit, never ...	0

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 999 entries, 0 to 998
Data columns (total 2 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   review     999 non-null    object
 1   sentiment  999 non-null    int64
dtypes: int64(1), object(1)
memory usage: 15.7+ KB
```

```
[35] df.describe()
```

	sentiment
count	999.000000
mean	0.500501
std	0.500250
min	0.000000

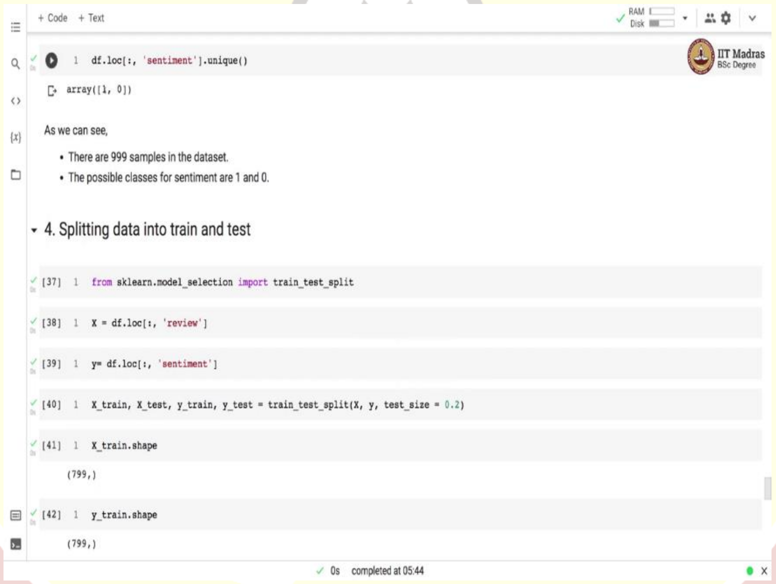
Now in the final section of the collab, what we will do is we will combine the preprocessing and fitting in incremental learning. So, here we will be using HashingVectorizer, along with SGDClassifier. So, now what we will do is we will work on a small text classification task, which is sentiment analysis or sentiment prediction. So for that, what we will do is we will represent each document with CountVectorizer.

And then, we will be using a SGDClassifier to detect or to classify the sentiment in the text document. And we will perform both HashingVectorizer an SGDClassifier in incremental manner. So, here we download a dataset from UCI machine learning library. So, instead of downloading unzipping and reading, we directly read the zip csv file for that we are using urllib.request.url open. And then we also, use BytesIO and TextIOWrapper class to get the data.

So, there are two columns in the data, which is the review under sentiment. So, let us explore the dataset a bit. So, you can see that there is a sentence in the review, which is Good case Excellent value under sentiment is 1. Then the second statement is Great for the jawbone sentiment is 1 third one is Tied to charger for conversations lasting more and this sentence is not complete here right now.

And the sentiment is 0. The mic is great, the sentiment is fine and so, on. So, there are in all 999 reviews, and 999 sentiment 1 sentiment per review.

(Refer Slide Time: 27:46)



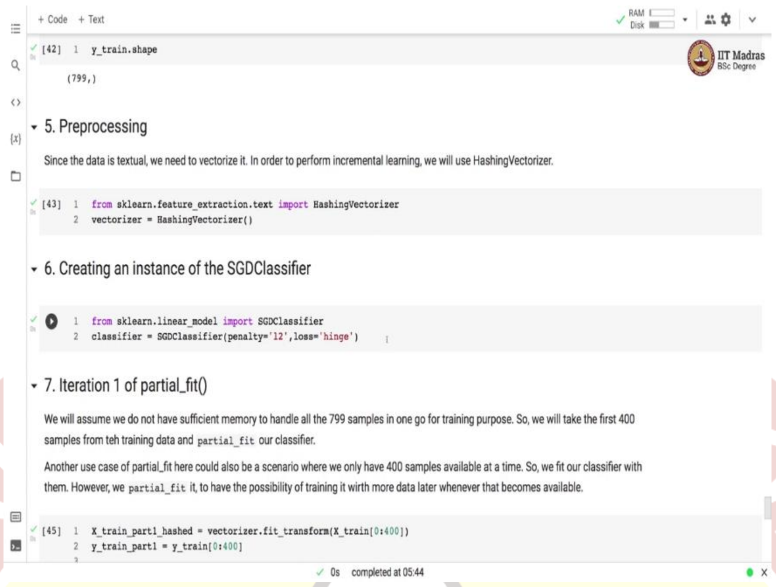
```
df.loc[:, 'sentiment'].unique()
array([1, 0])
```

As we can see,

- There are 999 samples in the dataset.
- The possible classes for sentiment are 1 and 0.

4. Splitting data into train and test

```
[37] from sklearn.model_selection import train_test_split
[38] X = df.loc[:, 'review']
[39] y= df.loc[:, 'sentiment']
[40] X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2)
[41] X_train.shape
(799,)
[42] y_train.shape
(799,)
```



```
[42] y_train.shape

(799,)
```

5. Preprocessing

Since the data is textual, we need to vectorize it. In order to perform incremental learning, we will use HashingVectorizer.

```
[43] from sklearn.feature_extraction.text import HashingVectorizer
     vectorizer = HashingVectorizer()
```

6. Creating an instance of the SGDClassifier

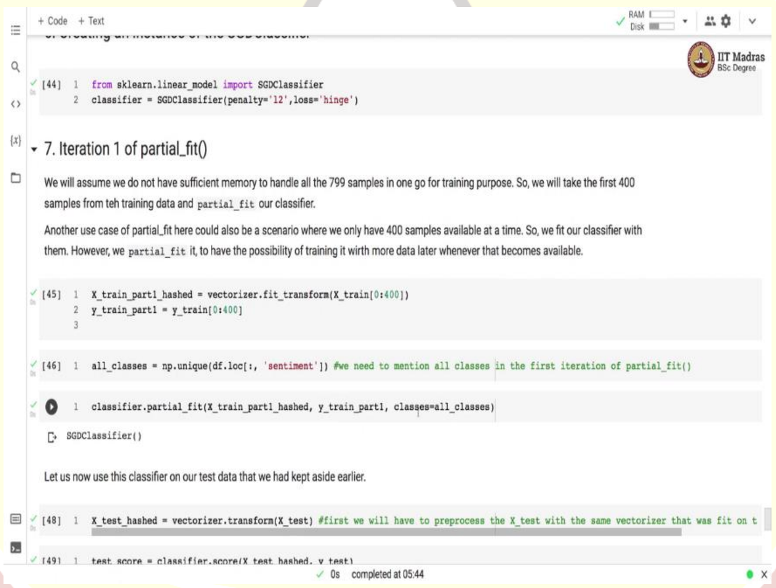
```
from sklearn.linear_model import SGDClassifier
classifier = SGDClassifier(penalty='l2',loss='hinge')
```

7. Iteration 1 of partial_fit()

We will assume we do not have sufficient memory to handle all the 799 samples in one go for training purpose. So, we will take the first 400 samples from teh training data and `partial_fit` our classifier.

Another use case of partial_fit here could also be a scenario where we only have 400 samples available at a time. So, we fit our classifier with them. However, we `partial_fit` it, to have the possibility of training it wirth more data later whenever that becomes available.

```
[45] X_train_part1_hashed = vectorizer.fit_transform(X_train[0:400])
     y_train_part1 = y_train[0:400]
```



```
[44] from sklearn.linear_model import SGDClassifier
     classifier = SGDClassifier(penalty='l2',loss='hinge')
```

7. Iteration 1 of partial_fit()

We will assume we do not have sufficient memory to handle all the 799 samples in one go for training purpose. So, we will take the first 400 samples from teh training data and `partial_fit` our classifier.

Another use case of partial_fit here could also be a scenario where we only have 400 samples available at a time. So, we fit our classifier with them. However, we `partial_fit` it, to have the possibility of training it wirth more data later whenever that becomes available.

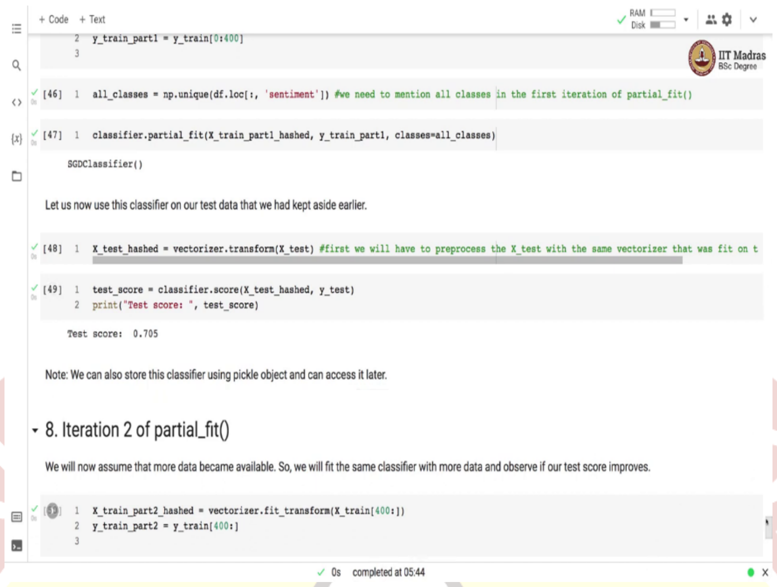
```
[45] X_train_part1_hashed = vectorizer.fit_transform(X_train[0:400])
     y_train_part1 = y_train[0:400]
```

```
[46] all_classes = np.unique(df.loc[:, 'sentiment']) #we need to mention all classes in the first iteration of partial_fit()
```

```
classifier.partial_fit(X_train_part1_hashed, y_train_part1, classes=all_classes)

SGDClassifier()
```

Let us now use this classifier on our test data that we had kept aside earlier.

```
[48] X_test_hashed = vectorizer.transform(X_test) #first we will have to preprocess the X_test with the same vectorizer that was fit on t

[49] test_score = classifier.score(X_test_hashed, y_test)
```



```
y_train_part1 = y_train[0:400]
```

```
[46] all_classes = np.unique(df.loc[:, 'sentiment']) #we need to mention all classes in the first iteration of partial_fit()
```

```
[47] classifier.partial_fit(X_train_part1_hashed, y_train_part1, classes=all_classes)
```

```
SGDClassifier()
```

Let us now use this classifier on our test data that we had kept aside earlier.

```
[48] X_test_hashed = vectorizer.transform(X_test) #first we will have to preprocess the X_test with the same vectorizer that was fit on t
```

```
[49] test_score = classifier.score(X_test_hashed, y_test)
     print("Test score: ", test_score)
```

```
Test score:  0.705
```

Note: We can also store this classifier using pickle object and can access it later.

8. Iteration 2 of partial_fit()

We will now assume that more data became available. So, we will fit the same classifier with more data and observe if our test score improves.

```
X_train_part2_hashed = vectorizer.fit_transform(X_train[400:])
y_train_part2 = y_train[400:]
```

And the sentiment is either 0 or 1, 0 meaning the negative sentiment and 1 meaning the positive sentiment. Now, we split the data into training and test set, we have 799 examples in the training. Since the data is in the in the text form, we need to vectorize it. And in order to perform the vectorization in an incremental manner, we use HashingVectorizer. So, we instantiate an object of HashingVectorizer.

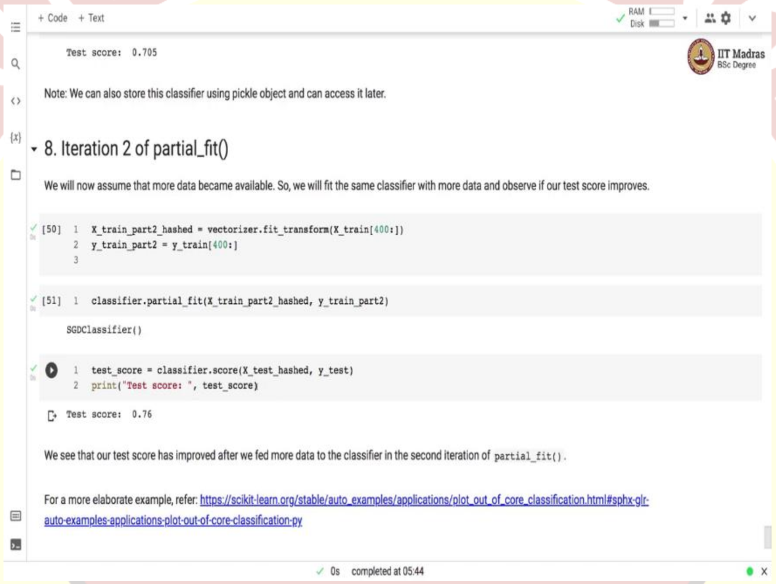
And then, we create an instance of a SGDClassifier with L2 regularization. And here we are using hinge-loss. So, this way of using SGDClassifier with hinge-loss, we basically get the SVM, which is the technique that we are going to learn in the next week. Now what we will do is we will first perform the vectorization of the documents. So, here we have 799 samples, we can very well use the fit method.

But since you are trying to demonstrate the partial _fit method or incremental learning, we will assume that we do not have sufficient memory to handle all 799 samples in one go. So, what we will do is we will first take 400 samples from the training data and perform partial _fit on those 400 samples. Alternatively, we can also, think that we only have 400 samples available at a time and you fit our classifier with these 400 samples.

And then whenever more samples will be available, we will basically update our model. So, partial _fit is also, useful in such a scenario. Then we first perform the transformation on the text document using CountVectorizer And then we call the partial _fit on the on the transformed representation. So, we perform the transform feature matrix and the labels along with the list of classes.

So, while performing the evaluation, we first transform the test set using the same CountVectorizer and call the score method on the classifier object by supplying the transform test representation along with test labels. And we obtained the score of 0.70 on the test set. So, this is using first 400 examples.

(Refer Slide Time: 30:45)



```
Test score:  0.705
```

Note: We can also store this classifier using pickle object and can access it later.

8. Iteration 2 of partial_fit()

We will now assume that more data became available. So, we will fit the same classifier with more data and observe if our test score improves.

```
[50]
X_train_part2_hashed = vectorizer.fit_transform(X_train[400:])
y_train_part2 = y_train[400:]

[51]
classifier.partial_fit(X_train_part2_hashed, y_train_part2)

SGDClassifier()

test_score = classifier.score(X_test_hashed, y_test)
print("Test score: ", test_score)

Test score:  0.76
```

We see that our test score has improved after we fed more data to the classifier in the second iteration of `partial_fit()`.

For a more elaborate example, refer: https://scikit-learn.org/stable/auto_examples/applications/plot_out_of_core_classification.html#sphx-glr-auto-examples-applications-plot-out-of-core-classification-py

Now, what we will do is we will use the remaining 399 examples and perform the second round of partial _fit. So, here, this is the second part of the training data, we transform our documents with HashingVectorizer and then perform the partial _fit with SGDClassifier. After training the data, we evaluate it on the test set. And now you can see that our test accuracy has gone up to 0.76.

In this video, we discussed how to train machine learning models on large datasets. We mainly use technique of incremental learning to partial _fit method. Any estimator and transformer implementing partial _fit method can be used for large-scale learning. We demonstrated working of large-scale learning with sentiment classification example. Now you have tools for solving real world large-scale learning problems.