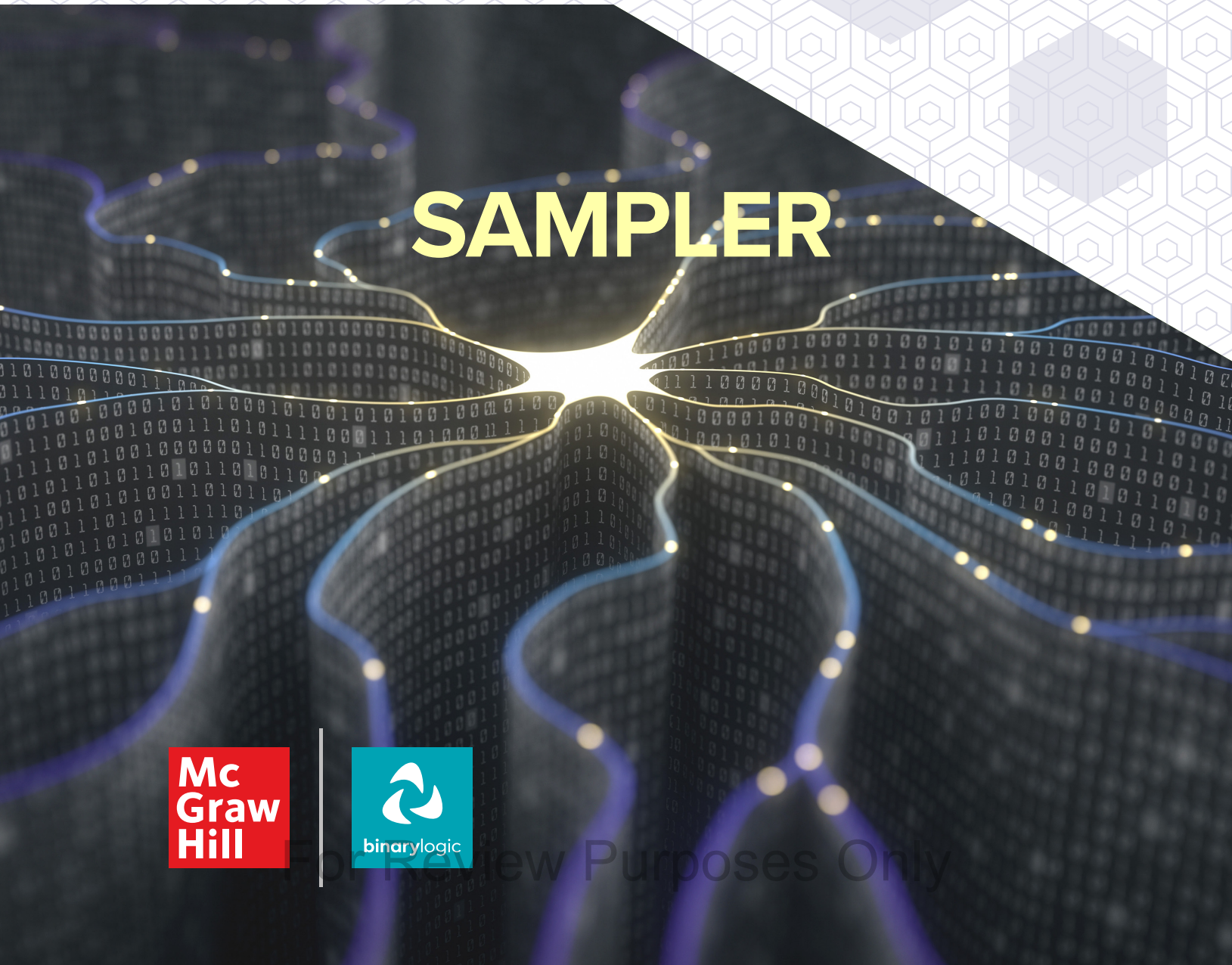
Foundations of AI

Artificial Intelligence 1



Mc Graw Hill | binarylogic

Foundations of AI

Artificial Intelligence 1

Foundations of AI:
Artificial Intelligence 1

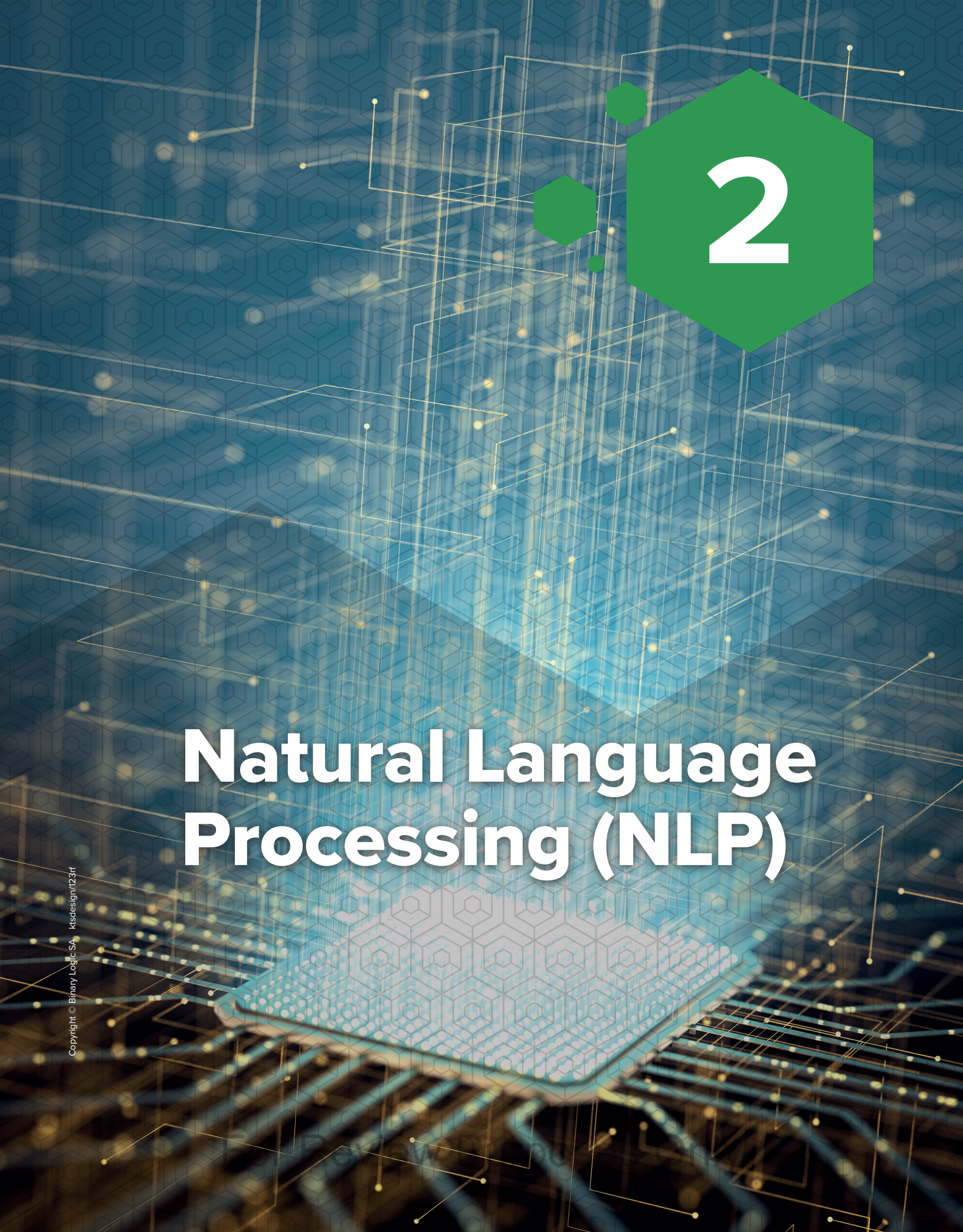
Printed and distributed by McGraw Hill in association with Binary Logic SA.

Cover Credit: © ktsdesign/123rf

 MHID: xxxxxxxxxx ISBN: xxx-xxx-xxx-xxx-x

mheducation.com binarylogic.net

Contents



2

Natural Language Processing (NLP)

INTRODUCTION

Machine learning helps computers understand and analyze text, making tools like chatbots and sentiment analysis possible. This unit focuses on how to train supervised and unsupervised learning models, explore Natural Language Processing (NLP) techniques, and create applications that can understand and generate text.

LEARNING OBJECTIVES

In this unit, you will:

> define supervised learning.

> train a supervised learning model to understand text.

> define unsupervised learning.

> train an unsupervised learning model to understand text.

> create a simple chatbot.

> generate text using the Natural Language Processing (NLP) techniques.

TOOLS

> **Jupyter Notebook**

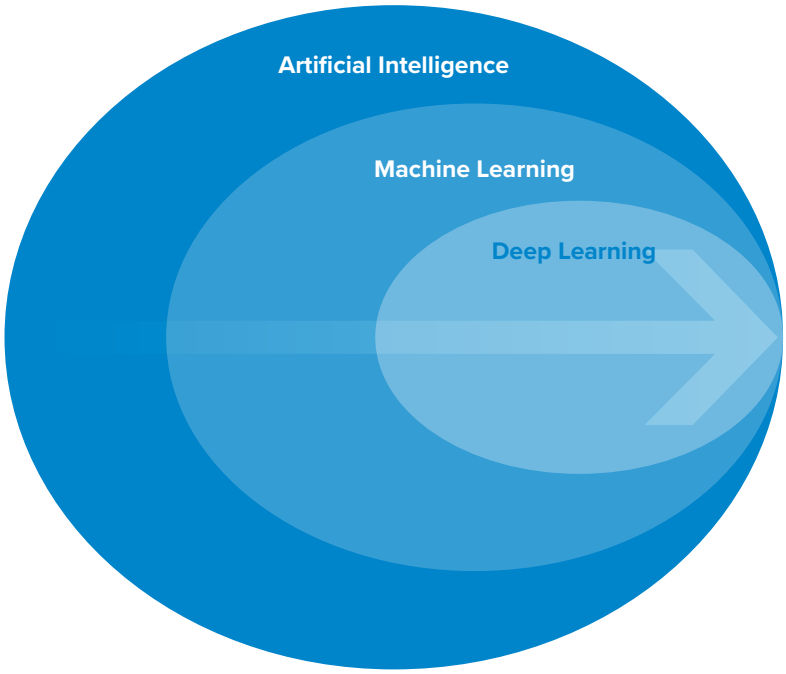
LESSON 1

Supervised Learning

Using Supervised Learning to Understand Text

Natural Language Processing (**NLP**) is a field of AI that focuses on enabling computers to understand, interpret, and generate human language. NLP is concerned with tasks such as text classification, sentiment analysis, machine translation, and question-answering. This lesson will focus specifically on how supervised learning, one of the main types of Machine Learning (ML), can be used to automatically understand and make useful predictions about a text's properties.

AI is an umbrella term that includes Machine Learning and **Deep Learning**, as explained in Unit 1 and illustrated in the figure below. AI is a broad field of computer science that focuses on creating intelligent machines, while machine learning is a subset of AI that focuses on building algorithms and models that allow machines to learn from data without being explicitly programmed.



Deep learning

Deep learning is a type of machine learning that uses deep neural networks to automatically learn from large amounts of data. It allows computers to recognize patterns and make decisions in a more humanlike way, by building complex models of the data.

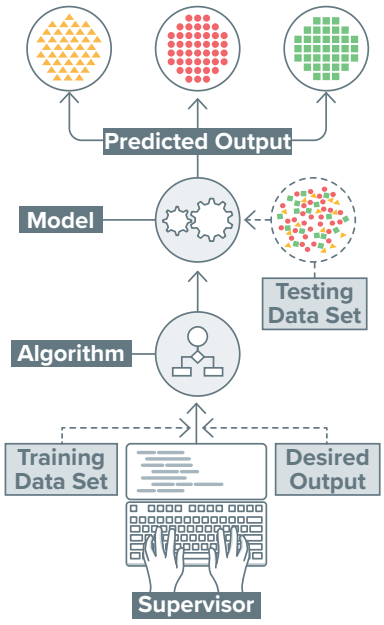
Machine learning (ML)

Machine learning (ML) is a subfield of AI that focuses on developing algorithms that enable computers to learn from data rather than following explicit programming instructions. It involves training computer models to recognize patterns and make predictions based on input data, allowing the model to improve its accuracy over time. This allows machines to perform tasks such as classification, regression, clustering, and recommendation, without being explicitly programmed for each task.

Machine learning can be broadly categorized into three main types:

Supervised learning is a type of Machine Learning where the algorithm learns from labeled training data, with the goal of making predictions on new data, not present in the training or test sets, as illustrated in the figure on the right. Examples:

- Image classification (for example, recognizing objects in photos).
- Fraud detection (for example, identifying suspicious financial transactions).
- Spam filtering (for example, identifying unwanted email messages).



Unsupervised learning is a type of Machine Learning where the algorithm works with unlabeled data, trying to find patterns and relationships in the data. Examples:

- Anomaly detection (for example, detecting unusual patterns in data).
- Clustering (for example, grouping similar data points together).
- Dimensionality reduction (for example, selecting the dimensions that reduce data complexity).

Reinforcement learning is a type of Machine Learning where an agent interacts with its environment and learns by trial and error, receiving rewards or punishments for its actions. Examples:

- Game playing (for example, playing chess or Go).
- Robotics (for example, teaching a robot to navigate its environment).
- Resource allocation (for example, optimizing resource usage in a network).

The table below summarizes the advantages and disadvantages of each type of machine learning.

Advantages and disadvantages of Machine Learning types

Advantages	Disadvantages
Supervised learning	
Well-established and widely used.	Requires labeled data, which can be expensive to obtain.
Easy to understand and implement.	Limited to the task it was trained for, and may not generalize well to new data.
Can handle both linear and non-linear data.	Difficult to adapt to other problems if the model is too complex.
Unsupervised learning	
Does not require labeled data, making it more flexible.	Harder to understand and interpret than supervised learning.
Can discover hidden patterns in data.	Limited to exploratory analysis and may not be suitable for decision-making tasks.
Can handle high-dimensional and complex data.	Difficult to adapt to other problems if the model is too complex.

Advantages	Disadvantages
Reinforcement learning	
Flexible and can handle complex and dynamic environments.	More complex than supervised or unsupervised learning.
Can learn from experience and improve over time.	Challenging to design reward functions that accurately capture the desired behavior.
Suitable for decision-making tasks, such as game playing and robotics.	May require large amounts of training data and computational resources.

Jupyter Notebook

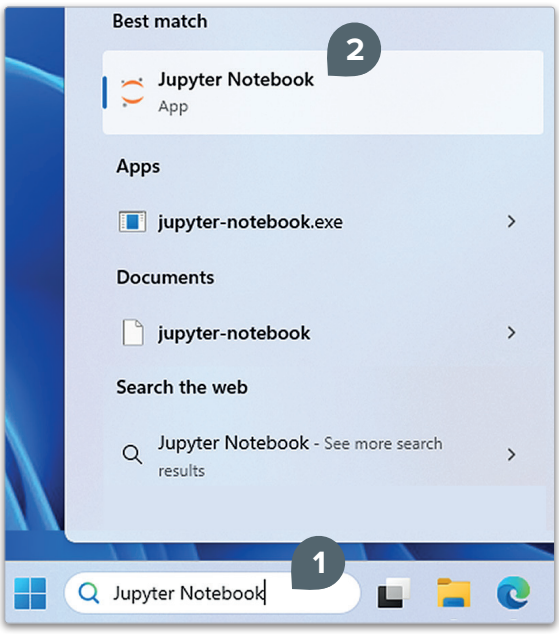
In this unit, you will write Python code using **Jupyter Notebook**. Jupyter Notebook is an online web application to create and share computational documents. Each document, called a notebook, includes your code, comments, raw and processed data, and data visualizations. You will use the offline version of Jupyter Notebook.

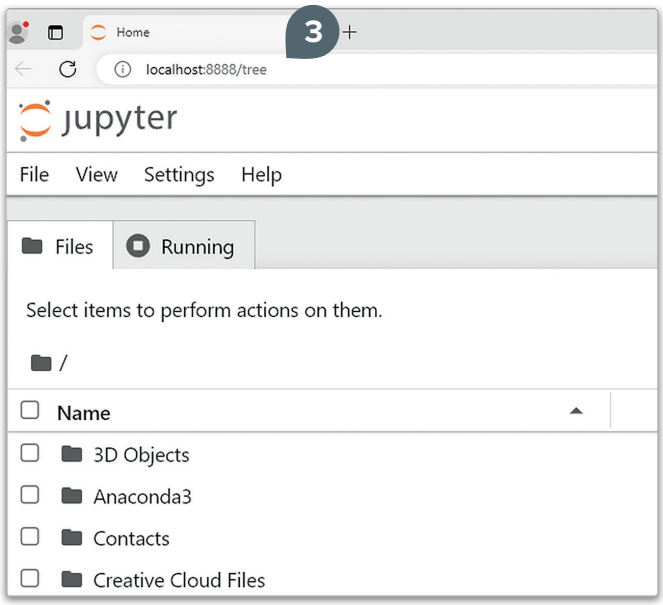
The easiest way to install it locally is through Anaconda, an open-source distribution platform, which is free for students and hobbyists. Download and install Anaconda from here:
https://www.anaconda.com/products/distribution

Python and Jupyter Notebook will be installed automatically.

To open Jupyter Notebook:

> On the Windows **Search** bar type "**Jupyter Notebook**". 1
> Open **Jupyter Notebook**. 2
> The Jupyter Notebook home page opens in the browser. 3

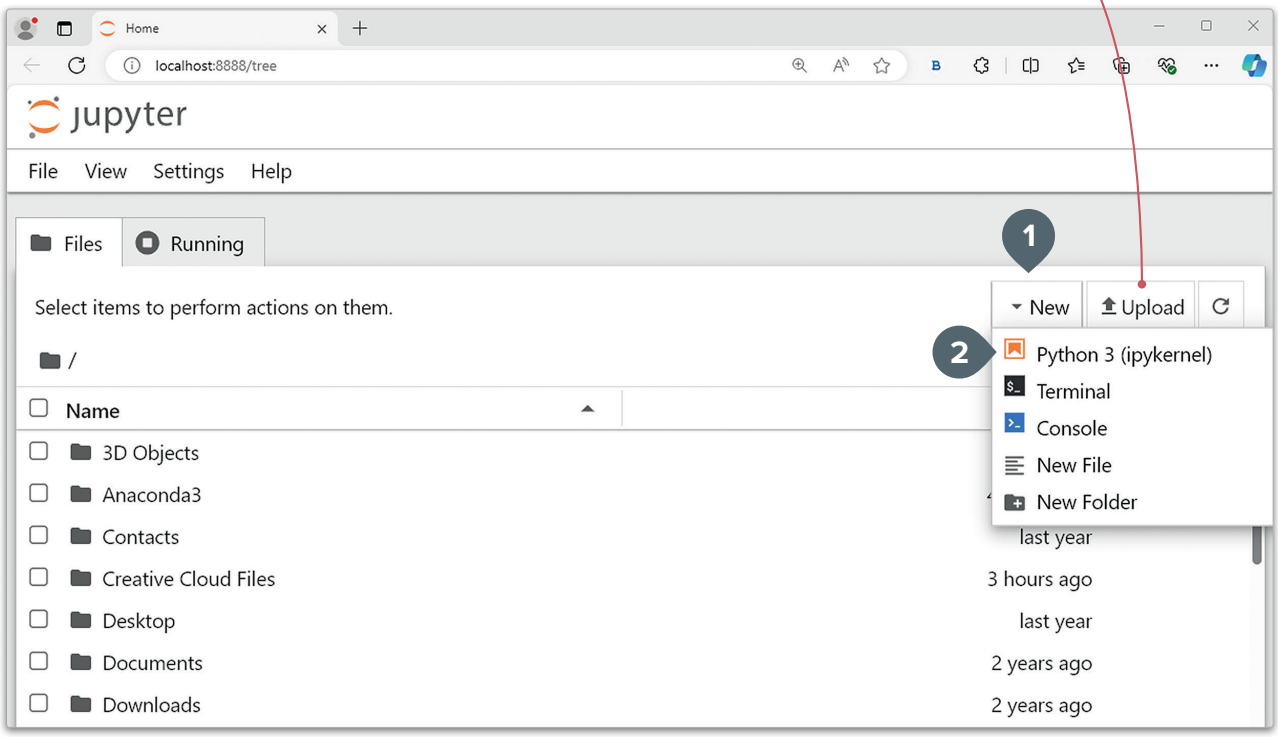




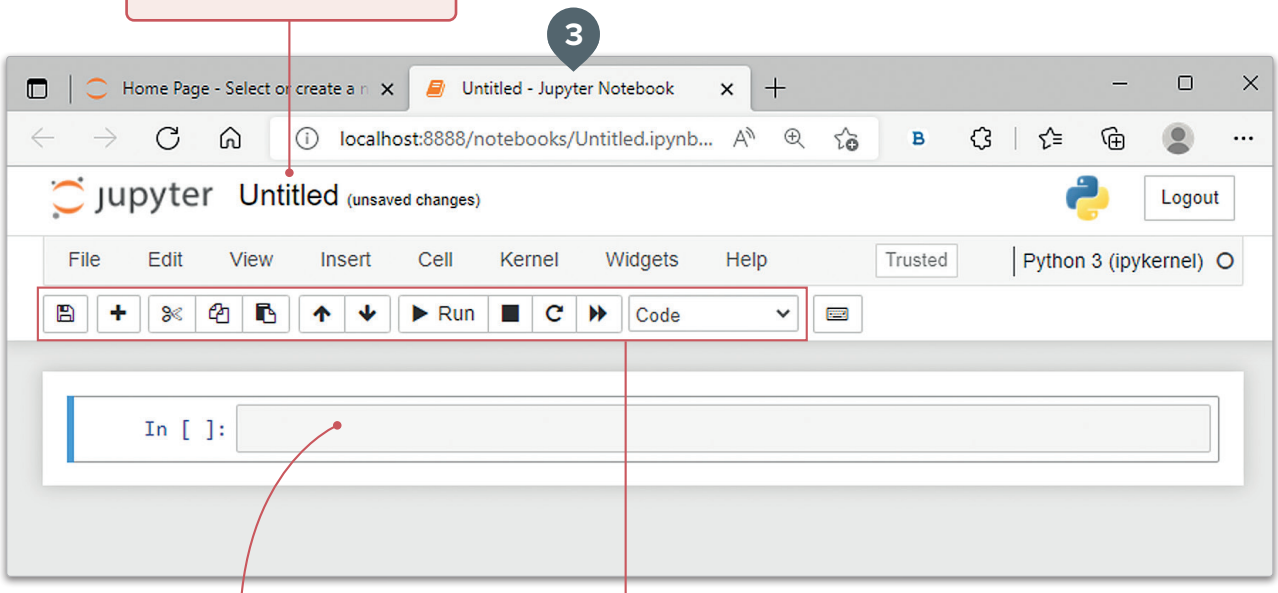
To create a new Jupyter Notebook:

> At the top-right corner of your screen, click **New**. 1
> Select **Python 3 (ipykernel)**. 2
> Your Notebook opens in a new tab in your browser. 3

You can **Upload** a notebook from your computer.



The default name of the notebook is **Untitled**.



Code cell. You can type text, a math expression, or a Python command.

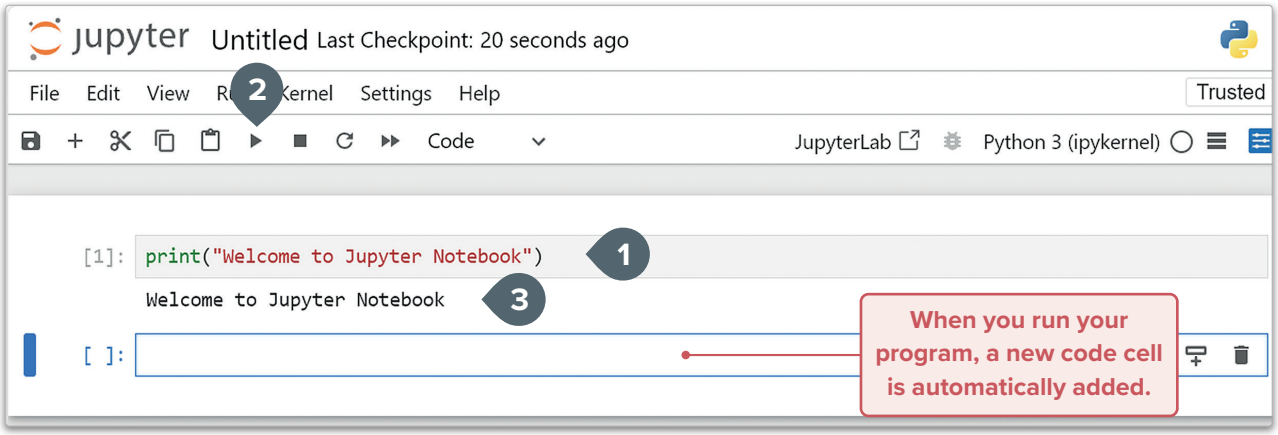
Notebook toolbar

Now that your notebook is ready, it's time to write and run your first program in Jupyter Notebook.

To create a program in Jupyter Notebook:

> Type the commands inside the code cell. 1
> Click the **Run** button. 2
> The result is displayed under the commands. 3

You can have as many different cells as you need in the same Notebook. Each cell contains its own code.

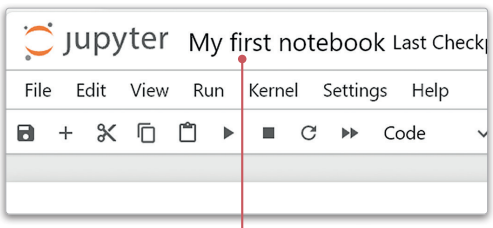


You can run your program by pressing Shift ⇧ + Enter ↵.

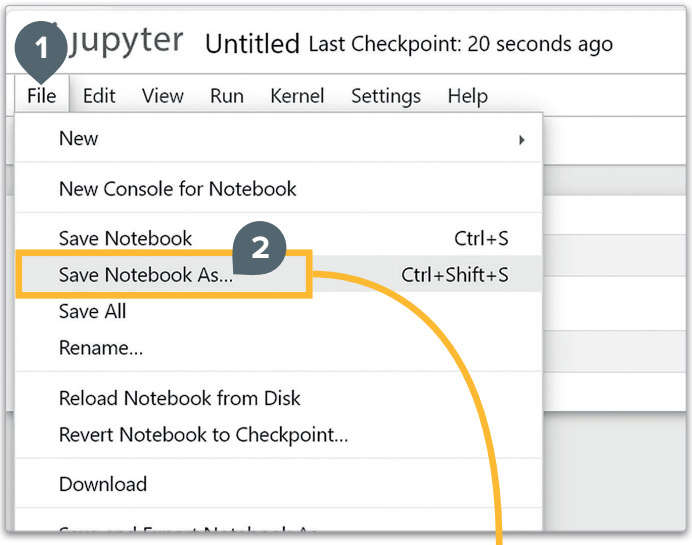
It's time to save your Notebook.

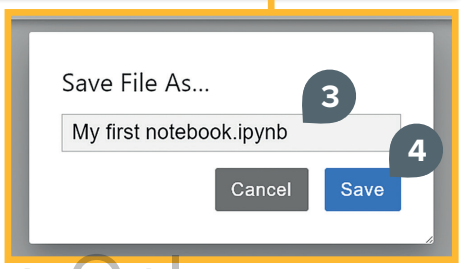
To save your Notebook:

> Click **File**. 1
> Select **Save Notebook As.** 2
> Type a name for your Notebook. 3
> Press **Save**. 4



The name of the notebook has changed.





When you are working, the Notebook is autosaved.

Supervised learning

Supervised learning is a type of ML that uses labeled data to train an algorithm to make predictions. The algorithm is trained on a labeled dataset and then tested on a hidden dataset. Supervised learning is commonly used in NLP for tasks such as text classification, sentiment analysis, and named entity recognition. In these tasks, the algorithm is trained on a labeled dataset where each example is labeled with the correct category or sentiment. If the labels represent continuous numbers, the task is called **regression**, which aims to predict values within a range. If the labels are discrete, the task is referred to as **classification**, which predicts specific categories or groups.

Supervised learning

In supervised learning, you use manually curated and labeled datasets to train computer algorithms to predict new values.

Regression

In regression tasks, for example, the focus can be on predicting the sale price of a house based on its size, location, and number of bedrooms. It can also be used to predict the demand for a product based on historical sales data and advertising expenditure. In an NLP context, regression can use the available text to predict the sentiment score of a movie review or the popularity of a social media post.

Classification

Classification, on the other hand, can be used in applications such as diagnosing a medical condition based on symptoms and test results. When it comes to understanding text, supervised learning can be used to classify or predict categories or labels based on the words and phrases within a document. For example, a supervised learning model might be trained to classify an email as spam or not spam based on the words and phrases used in the email. Another popular application is sentiment classification, which focuses on predicting whether the overall sentiment of a given document is negative or positive. This application is used as a working example in this unit to demonstrate all the steps in the end-to-end process of building and using a supervised learning model.

In this lesson, you will use a dataset of movie reviews. The dataset has already been split into two parts, one to be used for training the model and one to be used for testing. Download the train and test csv files for the Large Movie Review Dataset:

https://huggingface.co/datasets/jahjinx/IMDb_movie_reviews/resolve/main/IMDB_test.csv

https://huggingface.co/datasets/jahjinx/IMDb_movie_reviews/resolve/main/IMDB_train.csv

available from **http://ai.stanford.edu/~amaas/data/sentiment/**

To load the data into a DataFrame, you will use the **pandas** Python library. The pandas library is a popular tool for manipulating spreadsheet data. The following code is used to import the library into your program and then load the two datasets:

```
# capture is used to suppress the installation output
%%capture

# install the pandas library, if it is missing
!pip install pandas
import pandas as pd
```

Pandas is a popular library used to read and process spreadsheet-like data.

Dataset credit: Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. (2011). Learning Word Vectors for Sentiment Analysis. The 49th Annual Meeting of the Association for Computational Linguistics (ACL 2011).

```
# load the train and testing data
movie_train_reviews=pd.read_csv('imdb_train.csv')
movie_test_reviews=pd.read_csv('imdb_test.csv')

movie_train_reviews
```

The data files should be placed on the same folder that the Jupyter Notebook is running.

	text	label
0	Beautifully photographed and ably acted, gener...	0
1	Well, where to start describing this celluloid...	0
2	I first caught the movie on its first run on H...	1
3	I love Umberto Lenzi's cop movies -- ROME ARME...	0
4	I generally won't review movies I haven't seen...	0
...	...	...
35995	speaking solely as a movie, i didn't really li...	0
35996	This film plays like a demented episode of VH1...	0
35997	A couple of teenagers have a little sex on the...	0
35998	Good things out of the way first: U...	0
35999	I saw this in the summer of 1990. I'm still an...	0

36000 rows × 2 columns

The DataFrame dataset has two columns:

- text review.
- label.

A "0" label represents the negative review, while a "1" label represents the positive one.

positive review

negative review

The next step is to assign the text and label columns to separate variables, from the training and testing examples in the DataFrame dataset:

```
# extract the text from the 'text' column for both training and testing
X_train_text=movie_train_reviews['text']
X_test_text=movie_test_reviews['text']

# extract the labels from the 'label' column for both training and testing
Y_train=movie_train_reviews['label']
Y_test=movie_test_reviews['label']
X_train_text # training data in text format
```

The X, Y notations are typically used in supervised learning to represent the input data used to make the prediction (X) and the target labels (Y).

```
0        Beautifully photographed and ably acted, gener...
1        Well, where to start describing this celluloid...
2        I first caught the movie on its first run on H...
3        I love Umberto Lenzi's cop movies -- ROME ARME...
4        I generally won't review movies I haven't seen...
                              ...
35995    speaking solely as a movie, i didn't really li...
35996    This film plays like a demented episode of VH1...
35997    A couple of teenagers have a little sex on the...
35998    Good things out of the way first:<br /><br />U...
35999    I saw this in the summer of 1990. I'm still an...
Name: text, Length: 36000, dtype: object
```

Data Preparation and Pre-Processing

Even though this raw text format in the previous figure is intuitive to the human reader, it is unusable by supervised learning algorithms. Instead, algorithms require such documents to be converted into a numeric vector format. The **vectorization** process can be implemented in multiple different methods, and it has a great impact on the performance of the trained model.

sklearn library

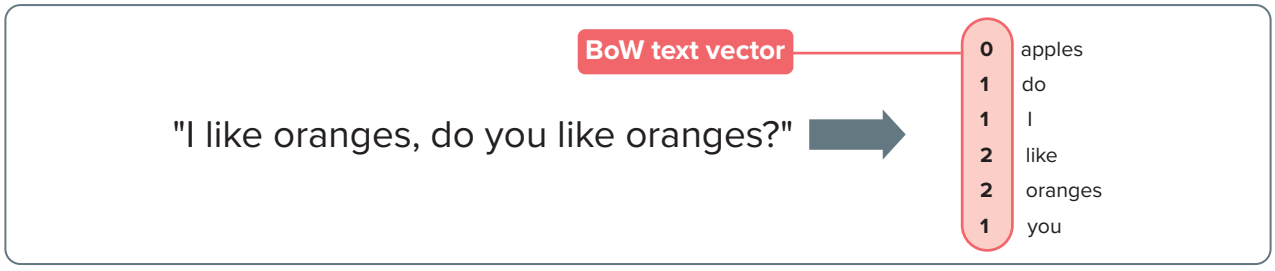
The supervised model will be built with sklearn (also known as "scikit-learn"), a popular Python library for machine learning. It provides a range of tools and algorithms for tasks such as classification, regression, clustering, and dimensionality reduction. One useful tool within sklearn is the CountVectorizer, which can be used to preprocess and vectorize text data.

CountVectorizer

The CountVectorizer converts a collection of text documents into a matrix of token counts, where each row represents a entry and each column represents a particular token. Tokens can be individual words, phrases, or even more complex constructs that capture various patterns in the underlying text data. The matrix entries (cells) represent the number of times each token exists in each entry. This is also known as Bag-of-Words (BoW) representation, as the order of the words is not preserved and only the counts of the words are retained. Even though the BoW representation is an oversimplification of human language, it can achieve very competitive results in practice.

Vectorization

Vectorization is the process of converting strings of words or phrases (text) to a corresponding vector of real numbers that is used to encode properties of the text using a format that ML algorithms can understand.



The following code uses the **CountVectorizer** tool to vectorize the movie training dataset:

```
from sklearn.feature_extraction.text import CountVectorizer

# the min_df parameter is used to ignore terms that exist in less than 10 reviews
vectorizer_v1 = CountVectorizer(min_df=10)

vectorizer_v1.fit(X_train_text) # fit the vectorizer on the training data
# use the fitted vectorizer to vectorize the data
X_train_v1 = vectorizer_v1.transform(X_train_text)

X_train_v1
```

Fit is like teaching the model to recognize the important words in the training data. This step helps the vectorizer "learn" which words to search for.

```
<36000x22161 sparse matrix of type '<class 'numpy.int64'>'
      with 4770650 stored elements in Compressed Sparse Row format>
```

```
# expand the sparse data into a sparse matrix format, where each column represents a different word
X_train_v1_dense=pd.DataFrame(X_train_v1.toarray(),
                 columns=vectorizer_v1.get_feature_names_out())
X_train_v1_dense
```

	00	000	007	01	02	04	05	06	08	09	...	zones	zoo	zoom	zooming	zooms	zorro	zu	zucco	zucker	zulu
0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
35995	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
35996	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
35997	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
35998	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
35999	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0

36000 rows × 22161 columns

This dense matrix format represents the 36,000 reviews in the training data. It also has a column for each of the words that presents in at least 10 reviews (enforced via the min_df parameter). As mentioned above, this creates a total of 22,161 columns, sorted in alphanumeric order. The matrix entry in position [i,j] represents the number of times that the j_th word presents in the i_th review.

Even though this matrix could directly be used by a supervised learning algorithm, it is highly inefficient in terms of memory usage. This is due to the fact that the vast majority of the entries in this matrix are equal to 0. This happens because only a very small percentage of the 22,161 possible words will actually be found in each review. To address this inefficiency, the **CountVectorizer** tool stores the vectorized data in a sparse format, which only remembers the non-zero entries in each column.

The code below uses the **getsizeof()** function, which returns the size of a Python object in bytes to demonstrate the memory savings of the sparse format for the movie data:

```
from sys import getsizeof
print('\nMegaBytes of RAM memory used by the raw text format:',
       getsizeof(X_train_text)/1000000)
print('\nMegaBytes of RAM memory used by the dense matrix format:',
       getsizeof(X_train_v1_dense)/1000000)
print('\nMegaBytes of RAM memory used by the sparse format:',
       getsizeof(X_train_v1)/1000000)
```

```
MegaBytes of RAM memory used by the raw text format: 49.506212

MegaBytes of RAM memory used by the dense matrix format: 6382.368144

MegaBytes of RAM memory used by the sparse format: 4.8e-05
```

As expected, the sparse format requires far less memory, more specifically 0.000048 megabytes. The dense matrix occupies 7 gigabytes. This matrix will not be used again and can thus be deleted to free up this significant amount of memory:

```
# delete the dense matrix
del X_train_v1_dense
```

Build a Prediction Pipeline

Once the training data has been vectorized, the next step is to build a prediction pipeline. A Machine Learning (ML) pipeline is a sequence of steps designed to process data and generate predictions. It typically begins by transforming raw data into a format that the model can interpret and then applies a trained model to make predictions. Organizing these steps into a pipeline streamlines the process, making it repeatable and ensuring consistent results when applying the same sequence to new data.

One commonly used classifier for document prediction is the Naive Bayes classifier. This algorithm predicts the likelihood of a document belonging to a particular class based on the probabilities of specific words or phrases occurring within the document. The "naive" aspect refers to the assumption that the occurrence of each word is independent of the occurrence of any other word. While this assumption is strong and may not hold in all cases, it enables the algorithm to be trained quickly and efficiently, making it highly effective in many scenarios.

Classifier

In ML, a classifier is a model that is used to distinguish data points into different categories or classes. The goal of a classifier is to learn from labeled training data, and then make predictions about the class label for new data.

The following code uses the implementation of the Naive Bayes classifier (**MultinomialNB**) from the **sklearn** library to train a supervised learning model on the vectorized movie review training data:

```
from sklearn.naive_bayes import MultinomialNB

model_v1=MultinomialNB() # a Naive Bayes classifier

model_v1.fit(X_train_v1, Y_train) # fit the classifier on the vectorized training data

from sklearn.pipeline import make_pipeline

# create a prediction pipeline: first vectorize using vectorizer_v1, then use model_v1 to predict
prediction_pipeline_v1 = make_pipeline(vectorizer_v1, model_v1)
```

For example, this code will produce a result array with the first element being "1" for a positive review and "0" for a negative review:

```
prediction_pipeline_v1.predict(['One of the best movies of the year. Excellent
                                cast and very interesting plot.',
                                'I was very disappointed with his film. I lost
                                all interest after 30 minutes' ])
```

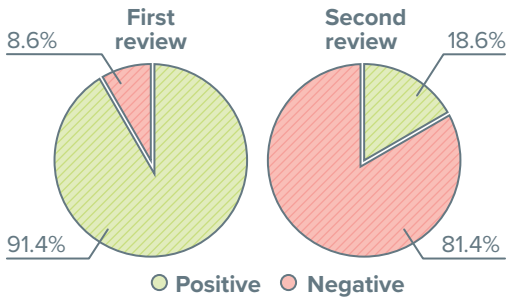
```
array([1, 0], dtype=int64)
```

The pipeline correctly predicts a positive and negative label for first and second reviews, respectively. The built-in function **predict_proba()** can be used to obtain the probabilities that the pipeline assigns to each of the two possible labels. The first element is the probability that "0" will be assigned and the second element is the probability that "1" will be assigned:

```
prediction_pipeline_v1.predict_proba(['One of the best movies of the year.
                                      Excellent cast and very interesting
                                      plot.',
                                      'I was very disappointed with his film. I
                                      lost all interest after 30 minutes' ])
```

```
array([[0.08567828, 0.91432172],
       [0.81365503, 0.18634497]])
```

The model is 8.6% certain the first review is negative and 91.4% certain it is positive. Likewise, it is 81.4% certain the second review is negative and 18.6% certain it is positive.



The next step is to test the accuracy of this new pipeline on the reviews in the movie testing set. The output is an array of all the result labels for the review given in the test data:

```
# use the pipeline to predict the labels of the testing data
predictions_v1 = prediction_pipeline_v1.predict(X_test_text) # vectorize the text data,
then predict

predictions_v1
```

```
array([0, 0, 0, ..., 0, 1, 0], dtype=int64)
```

Python provides multiple tools to analyze and visualize the results of classification pipelines. Examples include the **accuracy_score()** function from sklearn and the confusion matrix visualization from the scikit-plot library. There are also other evaluation metrics such as precision, recall, specificity, sensitivity, and F1 score, depending on the use case, which can be computed from the confusion matrix. The following output is an approximation of how accurate the prediction was:

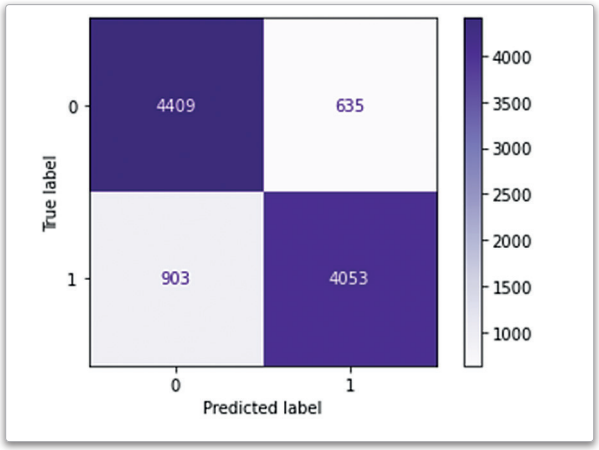
```
from sklearn.metrics import accuracy_score
accuracy_score(Y_test, predictions_v1) # get the achieved accuracy
```

```
0.8462
```

```
from sklearn.metrics import ConfusionMatrixDisplay
import matplotlib.pyplot as plt

# Assuming y_true (true labels) and y_pred (predicted labels) are defined
ConfusionMatrixDisplay.from_predictions(
                          Y_test, # actual labels
                          predictions_v1, # predicted labels
                          cmap=plt.cm.Purples #color palette to use
)
plt.show()
```

The confusion matrix contains the counts of actual vs. predicted classifications. In a binary classification task (for example, a problem with two labels, such as the movie task), the confusion matrix will have four cells:

	True Negatives (upper left): the number of times the classifier correctly predicted the negative class.
	False Positives (upper right): the number of times the classifier incorrectly predicted the negative class.
	False Negatives (lower left): the number of times the classifier incorrectly predicted the positive class.
	True Positives (lower right): the number of times the classifier correctly predicted the positive class.



The results reveal that even though this first pipeline achieves a competitive **accuracy** of 84.62%, it still misclassifies hundreds of reviews. You have 635 incorrect predictions in the upper-right quarter and 903 incorrect predictions in the lower-left corner. This totals 1,538 incorrect predictions. The first step toward improving performance is to study the behavior of the prediction pipeline in order to reveal how it processes and understands text.

Accuracy

Accuracy is the ratio of correct predictions to the total number of prediction.

$$\text{Accuracy} = \frac{(\text{True Positives} + \text{True Negatives})}{(\text{True Positives} + \text{True Negatives} + \text{False Positives} + \text{False Negatives})}$$

Explaining Black-Box Predictors

The Naive Bayes classifier uses simple mathematical formulas to combine the probabilities of thousands of words and deliver its predictions. Despite its simplicity, it is still unable to deliver an intuitive, user-friendly explanation of exactly how it predicts a positive or negative label for a specific piece of text.

Compare that to decision tree classifiers which are more intuitive, as they represent the learned decision rules in a tree-like structure, making it easier for people to understand how the classifier arrived at its predictions. The tree structure also allows for a visual representation of the decisions being made at each branch, which can be useful in understanding the relationships between input features and the target variable.

The lack of explainability is an even bigger challenge for more complex algorithms, such as those based on ensembles (combinations of multiple algorithms) or neural networks. Without explainability, supervised learning algorithms are reduced to black-box predictors: even though they understand the text well enough to predict its label, they are unable to communicate how they make their decisions.

A significant amount of research has been devoted to addressing this challenge by designing explainability methods that can interpret black-box models. One of the most popular methods is LIME (Local Interpretable Model-Agnostic Explanations).

Local Interpretable Model-Agnostic Explanations (LIME)

LIME is a method for explaining the predictions made by black-box models. It does this by examining at one data point at a time and making small changes to it to understand how it affects the model's prediction. LIME then uses this information to train a simple and understandable model, such as a linear regression, to explain the prediction. For text data, LIME identifies the words or phrases that have the biggest impact on the prediction. A Python implementation is illustrated below:

```
# install the lime library
!pip install lime
from lime.lime_text import LimeTextExplainer

# create a local explainer for explaining individual predictions
explainer_v1 = LimeTextExplainer()

# an example of an obviously negative review
easy_example='This movie was horrible. The actors were terrible and the plot was
very boring.'

# use the prediction pipeline to get the prediction probabilities for this example
print(prediction_pipeline_v1.predict_proba([easy_example]))
```

```
[[0.9987335 0.0012665]]
```

As expected, the predictor delivers a very confident negative prediction for this easy example.

```
# explain the prediction for this example
exp = explainer_v1.explain_instance(easy_example.lower(),
                                    prediction_pipeline_v1.predict_proba,
                                    num_features=10)
# print the words with the strongest influence on the prediction
exp.as_list()
```

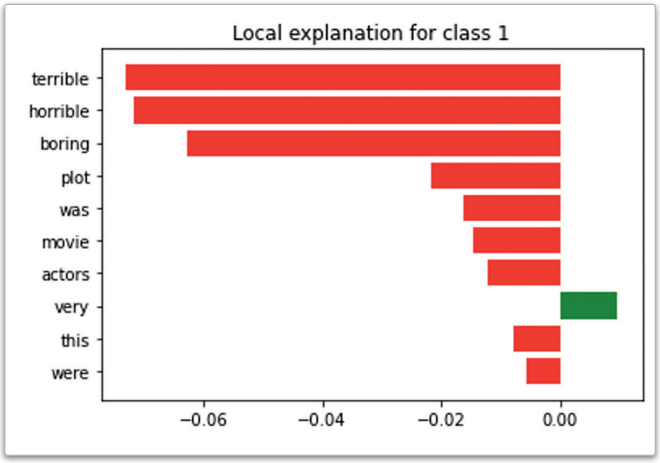
```
[('terrible', -0.07304471248484763),
 ('horrible', -0.07178969976550668),
 ('boring', -0.06274054713817784),
 ('plot', -0.021764756522240785),
 ('was', -0.016302963559748168),
 ('movie', -0.014635923349472214),
 ('actors', -0.012024588826189937),
 ('very', 0.009759621047832387),
 ('this', -0.007738594177898454),
 ('were', -0.005531412690294354)]
```

The score of each word represents a coefficient in the simple linear regression model that was used to deliver the explanation.

Focus the explainer on the 10 most influential features.

A more visual representation can be obtained as follows:

```
# visualize the impact of the most influential words
fig = exp.as_pyplot_figure()
```

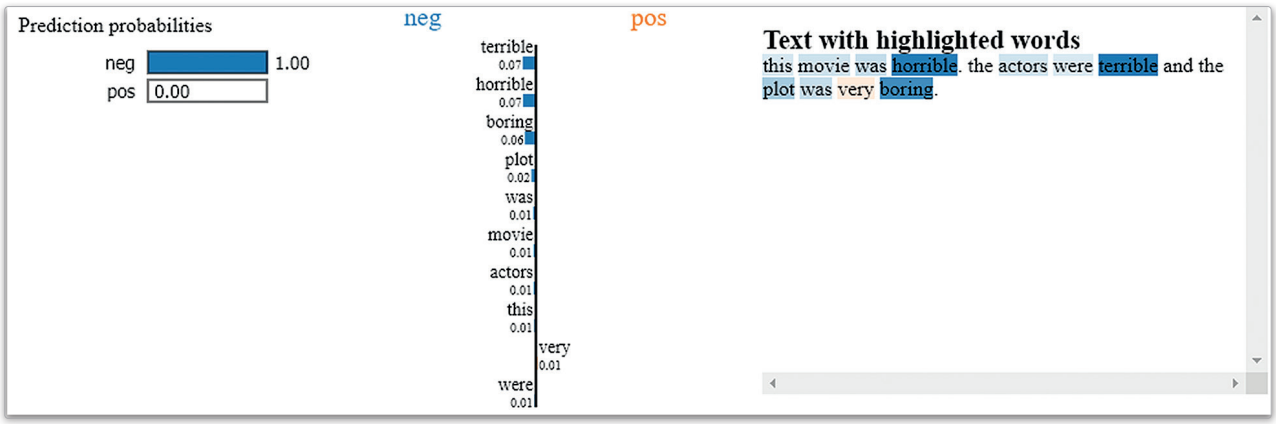


A negative coefficient increases the probability of the negative class, while a positive coefficient decreases it. For example, the words 'horrible', 'terrible', and 'boring' have the strongest impact on the model's decision to predict a negative label. The word 'very' slightly pushed the model in a different (positive) direction, but it was not nearly enough to change the decision. To a human observer, it might seem strange that sentiment-free words such as 'plot' or 'was' seem to have relatively high coefficients. However, it is important to remember that machine learning does not always follow human common sense. These high coefficients may indeed reveal flaws in the algorithm's logic and could be responsible for some of the predictor's mistakes. Alternatively, the coefficients may be indicative of latent but informative predictive patterns. For example, it may indeed be the case that

human reviewers are more likely to use the word 'plot' or use past tense 'was' when speaking in a negative context. The LIME Python library can also visualize the explanations in other ways.

For example:

```
exp.show_in_notebook()
```



The review used in the previous example was obviously negative and easy to predict. Examine the following more challenging review which can confuse the algorithm, taken from the testing set of the movie data:

```
# an example of a positive review that is mis-classified as negative by prediction_pipeline_v1
mistake_example= X_test_text[4600]
mistake_example
```

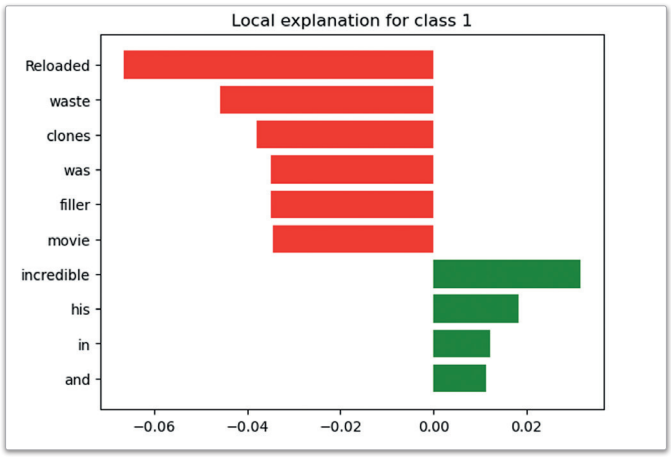
```
"The first Matrix movie was lush with incredible character development, witty
dialog, and action scenes that kept with the flow of the story. These elements
-- coupled by incredible special effects of the day -- presented a magical
ride that kept you in suspense the entire time. Enter Matrix Reloaded (and its
sequel, Revolutions). The problem here isn't the special effects or the fight
sequences as some may argue; The brothers have taken well-developed characters
from the first film and hollowed them out like rotten tree logs.\x85 The
connection that was first established between viewers and on-screen characters
in the first film is lost when you realize these are not the same characters
from the first Matrix movie......."
```

```
# get the correct labels of this example
print('Correct Label:', class_names[Y_test[4600]])

# get the prediction probabilities for this example
print('Prediction Probabilities for neg, pos:',
      prediction_pipeline_v1.predict_proba([mistake_example]))
```

```
Correct Label: [0]
Prediction Probabilities for neg, pos: [[9.99999986e-01 1.39884922e-08]]
```

Even though this is clearly a positive review, the pipeline reported a very confident negative prediction with a probability of 99.9%. The explainer can now be used to provide insight into why the predictor made this erroneous decision:

```
# explain the prediction for this example
exp = explainer_v1.explain_instance(mistake_example, prediction_pipeline_
v1.predict_proba, num_features=10)

# visualize the explanation
fig = exp.as_pyplot_figure()
```



Even though the predictor correctly captures the positive influence of certain words such as 'incredible', it ultimately makes a negative decision based on multiple words that seem to have no obvious negative sentiment (for example, 'Reloaded').

This demonstrates significant flaws in the logic that the predictor utilizes to classify the vocabulary in the text of the given reviews. The next section demonstrates how improving this logic can significantly boost the predictor's performance.

Improving Text Vectorization

The first version of the prediction pipeline used the CountVectorizer tool to simply count the number of times that each word presents in each review. This approach ignores two important facts about human language:

- The meaning and importance of a word can change based on the words that surround it.
- The frequency of a word within a document is not always an accurate representation of its importance. For example, even though two occurrences of the word 'incredible' may be a strong positive indicator in a document with 100 words, it is far less important in a larger document with 1,000 words.

Regular expression

A regular expression is a pattern of text used for matching and manipulating strings and provides a concise and flexible way to specify text patterns and is widely used in text processing and data analysis.

This section will demonstrate how text vectorization can be improved to take these two facts into account. The following code imports three different Python libraries that will be used to achieve this:

- **nltk** and **gensim**: two popular libraries used for various Natural Language Processing (NLP) tasks.
- **re**: a library used to search and process text using **regular expressions**.

```
%%capture

!pip install nltk # install nltk
!pip install gensim # install gensim

import nltk # import nltk
nltk.download('punkt') # install nltk's tokenization tool, used to split a text into sentences

import re # import re

from gensim.models.phrases import Phrases, ENGLISH_CONNECTOR_WORDS # import tools
from the gensim library.
```

Detecting phrases

The following function can be used to split a given document into a list of tokenized phrases, where each tokenized sentence is represented as a list of words:

Tokenization

The process of breaking up textual data into pieces such as words, sentences, symbols, and other elements called tokens.

```
# convert a given doc to a list of tokenized sentences.
def tokenize_doc(doc:str):
    return [re.findall(r'\b\w+\b',
            sent.lower()) for sent in nltk.sent_tokenize(doc)]
```

The **sent_tokenize()** function splits the doc into a list of sentences.

The **sent_tokenize()** function from the **nltk** library splits the given string into a list of sentences, if the given string is already a sentence, it does not split it anymore but simply return a list with that single sentence as the only element. Each sentence is then lowercased and fed to the **findall()** function of the **re** library, which locates occurrences a specified pattern, in this case, "**\b\w+\b**" regular expression.

You will test it on the string provided on the **raw_text** variable. In this expression:

- \w matches all alphanumeric characters (a-z, A-Z, 0-9) and the underscore character.
- \w+ is used to capture "one or more" \w characters. So, in the string "hello123_world", the pattern \w+ would match the words "hello", "123", and "world".
- \b represents the boundary between a \w character and a non-\w character, as well as at the start or end of the given string. For example, the pattern \bcat\b would match the word "cat" in the string "The cat is cute", but it would not match the word "cat" in the string "The category is pets".

Let's check out an example of tokenization using the **tokenize_doc()** function.

```
raw_text='The movie was too long. I fell asleep after the first 2 hours.'

tokenized_sentences=tokenize_doc(raw_text)

tokenized_sentences
```

```
[['the', 'movie', 'was', 'too', 'long'],
 ['i', 'fell', 'asleep', 'after', 'the', 'first', '2', 'hours']]
```

The **tokenize_doc()** function can now be combined with the **Phrases** tool from the **gensim** library to create a phrase model, a model that can identify multi-word phrases in a given sentence. The following code utilizes the movie training data (X_train_text) to build such a model:

```
sentences=[] # list of all the tokenized sentences across all the docs in this dataset

for doc in X_train_text: # for each doc in this dataset
    sentences+=tokenize_doc(doc) # get the list of tokenized sentences in this doc

# build a phrase model on the given data
movie_phrase_model = Phrases(sentences,
                             connector_words=ENGLISH_CONNECTOR_WORDS,
                             scoring='npmi',
                             threshold=0.25).freeze()
```

As illustrated above, the **Phrases()** function accepts four parameters:

- The list of tokenized sentences from the given document collection.
- A list of common English words that are repeated frequently in phrases (for example, 'of', 'the') that do not have any positive or negative value but can add sentiment depending on the context, so they are treated differently.
- A scoring function is used to determine if a sequence of words should be included in the same phrase. The code above uses the popular Normalized Pointwise Mutual Information (NPMI) measure for this purpose. NPMI is based on the co-occurrence frequency of the words in a candidate phrase and takes a value between -1 (complete independence) and +1 (complete co-occurrence).
- A threshold for the scoring function. Phrases with a lower score are ignored. In practice, this threshold can be tuned to identify the value that yields the best results for a downstream application (for example, predictive modeling).

The **freeze()** suffix converts the phrase model into an unchangeable ("frozen") but much faster format.

When applied to the two tokenized sentence examples illustrated above, this phrase model produces the following results:

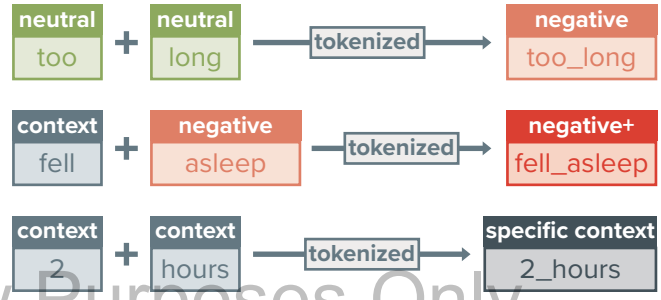
```
movie_phrase_model[tokenized_sentences[0]]
```

```
['the', 'movie', 'was', 'too_long']
```

```
movie_phrase_model[tokenized_sentences[1]]
```

```
['i', 'fell_asleep', 'after',
'the', 'first', '2_hours']
```

The phrase model identifies three phrases: 'too_long', 'fell_asleep', and '2_hours'. All three carry more information than their individual words.



For example, 'too_long' clearly carries a negative sentiment, even though the words 'too' or 'long' by themselves do not. Similarly, even though encountering the word 'asleep' in a movie review is likely negative evidence, the phrase 'fell_asleep' delivers a much clearer message. Finally, '2_hours' captures a much more specific context than the words '2' and 'hours'.

The following function uses this phrase-detection capability to annotate phrases in a given document:

```
def annotate_phrases(doc:str, phrase_model):

    sentences=tokenize_doc(doc)# split the document into tokenized sentences

    tokens=[] # list of all the words and phrases found in the doc
    for sentence in sentences: # for each sentence
        # use the phrase model to get tokens and append them to the list
        tokens+=phrase_model[sentence]
    return ' '.join(tokens) # join all the tokens together to create a new annotated document
```

The following code uses the **annotate_phrases()** function to annotate both the training and testing reviews from the movie dataset:

```
# annotate all the test and train reviews
X_train_text_annotated=[annotate_phrases(doc,movie_phrase_model) for doc in X_
train_text]
X_test_text_annotated=[annotate_phrases(text,movie_phrase_model)for text in X_
test_text]
# an example of an annotated document from the movie review training data
X_train_text_annotated[0]
```

```
'beautifully_photographed and ably acted generally but the writing is very
slipshod there_are scenes of such unbelievability that there_is no joy in the
watching the fact_that the young_lover has a twin_brother for instance is so
contrived that i groaned out_loud and the emotion light_bulb connection seems
gimmicky too br_br i_don t_know though if_you have a few glasses of wine and
feel_like relaxing with something pretty to look at with a few flaccid comedic
scenes this_is a pretty_good movie no major effort on the part of the viewer
required but italian film especially italian comedy is usually much much_better
than this'
```

Using TF-IDF for Text Vectorization

The frequency of a word within a document is not always an accurate representation of its importance. A better way to represent frequency is the popular **Term Frequency Inverse Document Frequency** (**TF-IDF**) measure. TF-IDF uses a simple mathematical formula to determine the importance of tokens (for example, words or phrases) in a document based on the two factors:

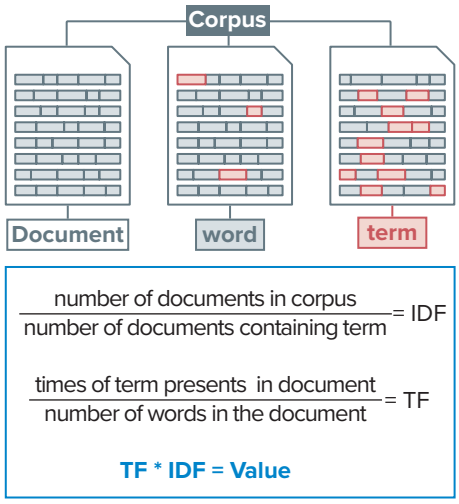
- **TF**: the frequency of the token in the document, as measured by the number of times the token presents in the document divided by the total number of tokens in the documents.

Term Frequency Inverse Document Frequency (TF-IDF)

TF-IDF is a statistical method which is used to determine the importance of tokens in a document.

- **IDF**: the token's inverse document frequency, computed by dividing the total number of documents in the dataset by the number of documents that contain the token.

The first factor avoids the overestimation of the importance of terms that are present in longer documents. The second factor penalizes terms that are present in too many documents, which helps to adjust for the fact that some words are more common than others.



TfidfVectorizer tool

The **sklearn** library provides a tool that supports this type of TF-IDF vectorization. The **TfidfVectorizer** tool can be used to vectorize a phrase.

```
from sklearn.feature_extraction.text import TfidfVectorizer

# Train a TF-IDF model with the movie review training dataset
vectorizer_tf = TfidfVectorizer(min_df=10)
vectorizer_tf.fit(X_train_text_annotated)
X_train_tf = vectorizer_tf.transform(X_train_text_annotated)
```

This new vectorizer can now be input to the same Naive Bayes classifier to build a new predictive pipeline and apply it to the movie testing data:

```
# train a new Naive Bayes classifier on the newly vectorized data
model_tf =MultinomialNB()
model_tf.fit(X_train_tf, Y_train)

# create a new prediction pipeline
prediction_pipeline_tf = make_pipeline(vectorizer_tf, model_tf)

# get predictions using the new pipeline
predictions_tf = prediction_pipeline_tf.predict(X_test_text_annotated)

# print the achieved accuracy
accuracy_score(Y_test, predictions_tf)
```

```
0.8812
```

This new pipeline achieves an accuracy of 88.12%, a significant improvement over the 84.62% reported by the previous one. This improved pipeline can now be used to revisit the test example that was misclassified by the first pipeline:

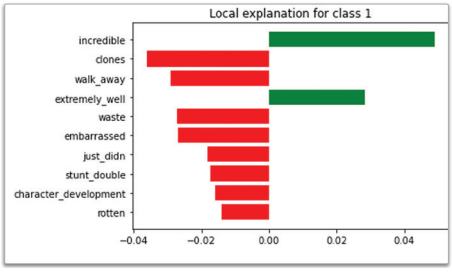
```
# get the review example that confused the previous algorithm
mistake_example_annotated=X_test_text_annotated[4600]

print('\nReview:',mistake_example_annotated)
```

```
# get the correct labels of this example
print('\nCorrect Label:', class_names[Y_test[4600]])

# get the prediction probabilities for this example
print('\nPrediction Probabilities for neg, pos:',prediction_pipeline_tf.predict_
proba([mistake_example_annotated]))
```

```
Review: the first matrix movie was lush with incredible character_development
witty_dialog and action_scenes that kept with the flow of the story these_
elements coupled by incredible special_effects of the day presented a magical
ride that kept you in suspense the entire time enter matrix_reloaded and its
sequel revolutions the problem here isn_t the special_effects or the fight_
sequences as some may_argue the brothers have_taken well_developed characters
from the first film and hollowed them out like rotten tree logs the connection
that was first established between viewers and on screen characters in the
first film is lost when_you realize these are not the same characters from
the first matrix movie br_br to wit morpheus was developed as a charismatic
philosophical character with insight far exceeding anyone_else in the movie but
here in reloaded we_re presented by a different morpheus who stands hard and
hollow reduced to corny one_liners that contradict the character we saw develop
in the first film this character ..............
Correct Label: [0]

Prediction Probabilities for neg, pos: [[0.65729403 0.34270597]]
```

The new pipeline confidently predicts the correct positive label for this review. The following code uses the LIME explainer to explain the logic behind this prediction:

```
# create an explainer
explainer_tf = LimeTextExplainer()

# explain the prediction of the second pipeline for this example
exp = explainer_tf.explain_instance(mistake_example_annotated, prediction_
pipeline_tf.predict_proba, num_features=10)

# visualize the results
fig = exp.as_pyplot_figure()
```



The results verify that the new pipeline follows a significantly more intelligent logic. It correctly identifies the positive sentiment of phrases like 'incredible' and 'extremely_well'. It is also not misguided by the words that erroneously drove the first pipeline toward a negative prediction.

The performance of the predictive pipeline can be further improved in multiple ways, such as replacing the Naive Bayes classifier with more sophisticated methods and tuning the parameters of these methods to maximize their potential. Another option would be to experiment with alternative vectorization techniques that are not based on token frequency, such as the word and document embeddings that will be explored in the following lesson.

EXERCISES

1 Explain the reason the dense matrix format requires more space in the memory than the sparse format.

2 Analyze how the two mathematical factors in TD-IDF are utilized to inspect the importance of a word in a document.

3 You are given a numPy array X_train_text that includes one document in each row. You are also given a second array Y_train that includes the labels for the documents in X_train_text. Complete the following code so that it uses TF-IDF to vectorize the data, trains a MultinomialNB classification model on the vectorized version, and then combines the vectorizer and classification model into a single prediction pipeline. Write the answers in your notebook.

```
from   1   .naive_bayes import MultinomialNB
from sklearn.pipeline import make_pipeline
from sklearn.feature_extraction.text import   2
vectorizer =   3  (min_df=10)

vectorizer.fit(   4   ) # fits the vectorizer on the training data

X_train = vectorizer.   5  (X_train_text) # uses the fitted vectorizer to vectorize the data
model_MNB=MultinomialNB() # a Naive Bayes classifier
model_MNB.fit(X_train,   6   ) # fits the classifier on the vectorized training data
prediction_pipeline = make_pipeline(   7   ,   8   )
```

4 Complete the following code so that it builds LimeTextExplainer for the prediction pipeline that you built in the previous exercise and uses the explainer to explain the prediction for a specific text example. Write the answers in your notebook.

```
from  1  import LimeTextExplainer
text_example="I really enjoyed this movie, the actors were excellent"
class_names=['neg','pos'] # creates a local explainer for explaining individual predictions
explainer =  2 (class_names=class_names) # explains the prediction for this example
exp = explainer. 3 (text_example.lower(),prediction_pipeline. 4 ,
                 5 =10) # focuses the explainer on the 10 most influential features
print(exp. 6 ) # prints the words with the highest influence on the prediction
```

PROJECT

Text classification

Text classification is a two-step process that includes:

Step 1: Using a set of training documents with known labels (classes) to train a classification model.

Step 2: Using the trained model to predict the label for each document in a testing set. The labels in the testing set are either unknown or hidden and used later for verification.

The documents in both the training and testing sets have to be vectorized before they can be used. The CountVectorizer or TfidfVectorizer tools from the sklearn library can be used for vectorization.

The Python sklearn library offers a long list of classification models. Some of them are:

> GradientBoostingClassifier()

> DecisionTreeClassifier()

> RandomForestClassifier()

Your task is to use the Movie Review training dataset that was used in the first lesson to train a model that achieves the highest possible accuracy on the Movie Review testing dataset (imdb_data/imdb_test.csv). You can achieve this by:

1. Replace the MultinomialNB classifier with other classification models from sklearn, such as the ones listed above.

2. Re-run your notebook after each replacement to compute the accuracy of each new model that you try.

3. Create a report that compares the accuracy of all the models that you tried and identifies the one that achieved the best accuracy.

2 WRAP UP

THIS UNIT COVERED HOW TO:

> classify text with unsupervised learning models.
> analyze text with supervised learning models.
> use machine learning models for NLG.
> program a simple chatbot.

KEY TERMS

- Black-Box Predictors
- Chatbot
- Classification
- Cluster
- Dendrogram
- Dimensionality Reduction
- Document Clustering
- Machine Learning
- Natural Language Generation (NLG)
- Natural Language Processing (NLP)
- Part of Speech (POS) Tags
- Regression
- Sentiment Analysis
- Supervised Learning
- Syntax Analysis
- Tokenization
- Transfer Learning
- Unsupervised Learning
- Vectorization

Foundations of AI

Artificial Intelligence 1

Optimize with AI

Picture a world where machines can think, learn, and solve problems on their own. What if you could design algorithms that enable intelligent systems to make decisions, tackle challenges, and drive innovation—whether it's automating everyday tasks or advancing the field of robotics?

Foundations of AI: This course teaches you the basics of artificial intelligence and optimization. Explore optimization problems, resource management, and scheduling, while learning techniques to improve decision-making in real-world situations.

By the end of this course, you'll have the skills to design and implement powerful AI algorithms, optimize systems for efficiency, and apply your knowledge to robotics and automation. You'll be empowered to build intelligent systems that can reshape industries and solve problems across the globe.

Mc Graw Hill

binarylogic

ISBN: XXX-XXX-XXX-XXX-X