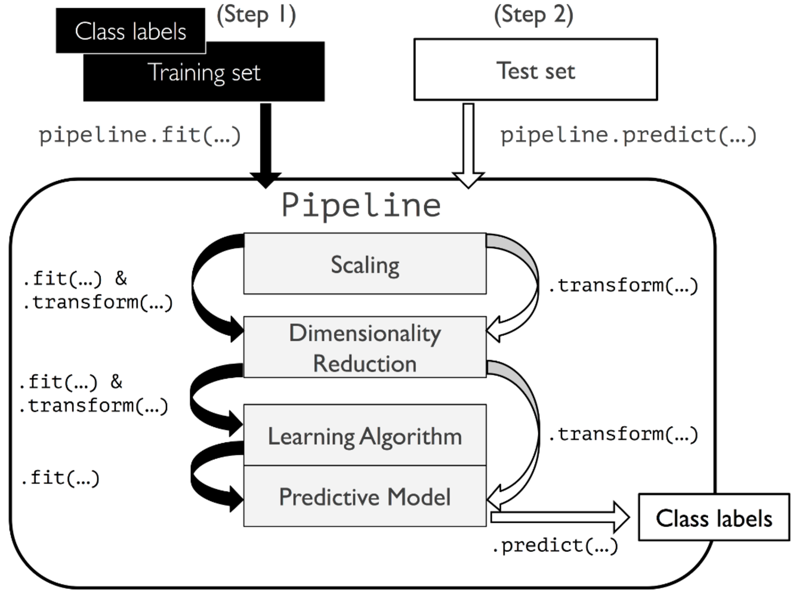
**Model Pipeline**

Pipeline is a tool that simplifies and automates the process of building a machine learning workflow.

* It allows for the **sequential execution of multiple steps**, combining **data preprocessing** and the **final model training or prediction** into a single and streamlined process.
* Sequential chain of **transformers** (e.g., preprocessors like scaling, encoding, or feature extraction) followed by an **estimator** (e.g., a classifier or regressor) 🡺 ensures that the entire workflow is applied consistently and efficiently.
* Each step in the pipeline performs its transformation and passes the output to the next step.
  + **Transformers**:
    - Apply ***.fit\_transform()*** during **training** and ***.transform()*** during **testing**.
  + **Estimator**:
    - Applies ***.fit()*** during **training** and ***.predict()*** during **testing**.
* **Workflow**



Binary classification with numeric features

A screen shot of a computer code

Description automatically generated

A black text on a white background

Description automatically generated

* The goal is to predict the binary classification with building the pipeline.

**Components of a pipeline**

1. **Steps**:

* Each step is a tuple of a name and an object (transformer or estimator) 🡺 ('scaler', StandardScaler()).

A black text on a white background

Description automatically generated

1. **Preprocessing**:

* Steps like scaling, encoding, or imputing missing values.
* Must implement ***.fit()*** and ***.transform()*** methods.

1. **Estimator**:

* The final step of the pipeline is the machine learning model.
* Must implement ***.fit()*** and ***.predict()*** methods.

A close-up of words

Description automatically generated

1. Hyperparameter tuning with pipeline

* Use “\_ \_” for each specific model for the hyperparameter tuning.
* Apply pipeline with dictionary of the hyperparameters using ***.fit(X\_train, y\_train)***

A screenshot of a computer code

Description automatically generated

**Workflow of pipeline in binary task using *Pipeline([ ])***

A screen shot of a computer code

Description automatically generated

* **'scale'**: Applies StandardScaler to standardize the features by removing the mean and scaling to unit variance.
* **'pca'**: Uses PCAto reduce the dimensionality of the dataset to 15 components.
* **'lr'**: Fits a LogisticRegression model as the final estimator.
* Using ***.fit(train, test)*** to get the sense of the model prediction by log reg.

A screenshot of a number

Description automatically generated

* Use log reg model performance from pipeline (pipe1) to extract the PCs coefficients

A screenshot of a number

Description automatically generated

* Use PCA model from pipeline (pipe1) to get each PCs

**Hyperparameter tuning via Pipeline**

A screenshot of a computer code

Description automatically generated

* We need to specify the hyperparameter’s name with its corresponding model names with **“\_ \_”**
* Fit the pipeline (pipe1) with the specified hyperparameters to fit the X\_train and y\_train.
* Apply ***.score(X\_train, y\_train)*** to get the result.

**Pipeline display**

* By fitting the training data from the pipeline construction, we can visualize how each parameter inside the pipeline’s content and the workflow
* Order of each of the parameter inside the pipeline matters.

A screenshot of a computer

Description automatically generated

* Display the best estimator of each model inside the pipeline using ***.best\_estimator\_***

A screenshot of a computer program

Description automatically generated

**Column Transformer**

It allows you to apply different transformations to different subsets of columns, which is particularly useful when dealing with datasets that have mixed data types, such as categorical and numerical features.

A table with numbers and letters

Description automatically generated

Column Transformer components

* Selective Transformation:
  + Apply specific preprocessing steps to **specific columns**.
* Integration with Pipelines:
  + Can be **used as part of a scikit-learn pipeline**.
* Support for Mixed Data Types:
  + Handle **numeric and categorical** data differently.

Workflow of column transformer with pipeline

1. Apply data column with **numerical features** processing with specific columns

A screen shot of a computer code

Description automatically generated

* + Fill the missing value for columns age and fare with median with ‘imputer’
  + Standardize both columns with standard scaler with ‘scaler’

1. Apply another different column with **categorical features** with different processing method

A close-up of a computer code

Description automatically generated

* + Use simple imputing and one-hot encoding to process specified columns with ‘imputer’ and ‘onehot’

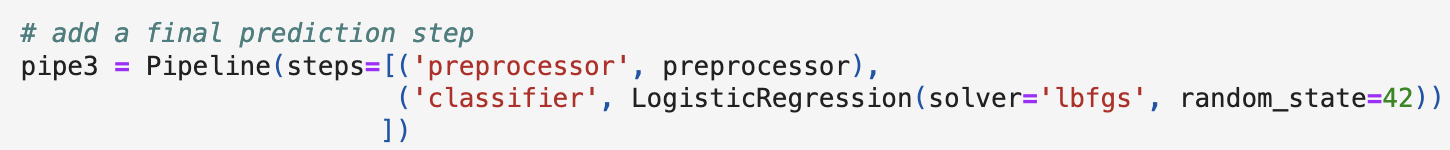
1. Combine both processing column transformer pipeline as preprocessor.

A computer code with black text

Description automatically generated

* + Combine the preprocessing with ColumTransformer

1. Set the final pipeline with the model training



1. Visualization of pipeline

A screenshot of a computer

Description automatically generated

1. Apply the data splitting for model performance

A computer screen shot of a computer code

Description automatically generated

* + We can use ‘survived’ as the binary target and all other features as the IVs
  + Apply train\_test\_split to split the data with X and y for model training

1. Connect with previous column transformation pipeline with split data using ***pipeline.fit (X, y)***

A screenshot of a computer code

Description automatically generated

* + With pipe3, we conducted all the column transformation with numerical and categorical feature transformation and apply to the splitted train and test dataset.

1. Hyperparameter tuning with pipeline

A screenshot of a computer code

Description automatically generated

* + We integrated the defined parameter for imputer and hyperparameter for classifier.
  + Use Grid Search to incorporate pipeline and set of hyperparameter with 3-fold CV.
  + After we impute the data by column transformation and hyperparameter tuning, we fit the training data with the integrated pipeline and obtain the best score for test set with 75.2%

1. Visualize the holistic pipeline structure

A screenshot of a computer

Description automatically generated

**NLP**

NLP terms with example (a corpus with 2 sentences)

**“I love NLP.”**

**“NLP is fun and interesting!”**

1. **Token (Smallest Unit**):

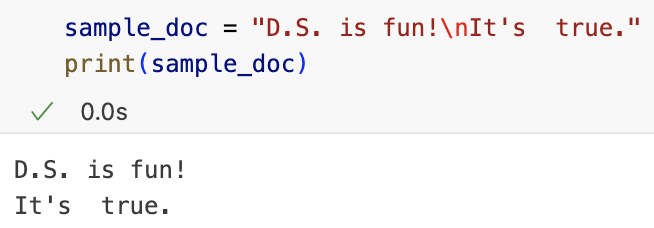
* A token is a single word or part of word extracted after splitting the text.
* ["I", "love", “NLP”, “.”] or [“I”, “l”, “o”, “v”, “e”, “,” …]

1. **String (Sequence of characters):**

* A string is a list of characters representing raw text.
* "I love NLP." Or ‘I love’

1. **Document (Unit of cohesive text)**

* A document is a cohesive unit of text that can consist of one or more strings.
* “I love NLP." Or ‘I love’



1. **Corpus (Collection of documents)**

* A corpus is a collection of multiple documents.
* ["I love NLP.", "NLP is fun and interesting!"]

1. **Vocabulary (Largest Representation):**

* The vocabulary is the set of unique tokens found in the entire corpus.
* Unique words, punctuation, and case-sensitive in this example
* {"I", "love", "NLP", ".", "is", "fun", "and", "interesting", “!”}

**Regular Expression Query**

Define search pattern in the text in Regular expression by string

* Use all lib ‘re’ for executing query

1. To find all of the whitespaces in doc by “\s+”

A screenshot of a computer code

Description automatically generated

1. To find any single character except newline (r'.' matches 'x')
2. To match 0 or more repetitions (r'x\*' matches 'x','xx','')
3. To match 1 or more repetitions (r'x+' matches 'x','xx')
4. To match 0 or 1 repetitions (r'x?' matches 'x' or '')
5. `^`: beginning of string (r'^D' matches 'D.S.')
6. `$`: end of string (r'fun!$' matches 'DS is fun!'`)

A screenshot of a computer

Description automatically generated

**Tokenization**

A process of breaking a document into a list smaller unit (tokens) to prepare text for further analysis by *converting unstructured data (raw text) into structured tokens*.

* **Goal**: we need to convert each token into feature vector for later model performance after we have tokenized raw data into a simpler form to reduce redundancy.
* **Token**: The individual elements resulting from tokenization. These can be words, punctuation marks, or subwords, depending on the tokenizer.
  + Document: "The cat sits on the mat."
  + Tokens: ["The", "cat", "sits", "on", "the", "mat", "."]
* **Vocabulary**:
  + The set of unique tokens in a corpus. It represents the complete list of words or symbols used across all documents.

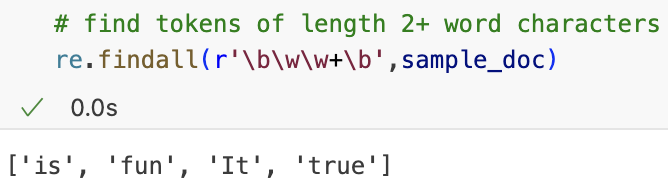
Ways of splitting document into tokens

1. Split on whitespace with ***re.split(r ‘reg expression’, document )***

A screenshot of a computer

Description automatically generated

1. Find all the regular expression query using ***re.findall(r ‘reg expression’, document )***



**NLP Preprocessing**

We can process the text sentences from raw data into structured for later analysis

1. Convert the text into lower-cased sentence using ***text.lower()***

A screenshot of a computer code

Description automatically generated

1. Remove special characters in the raw data including punctuation using regular expression query.

A screenshot of a computer

Description automatically generated

1. Lemmatization: Reduce words to their base (lemma) form to standardize variations.

* Use WordNetLemmatizer

A close-up of words

Description automatically generated

* Initialize lemmatizer

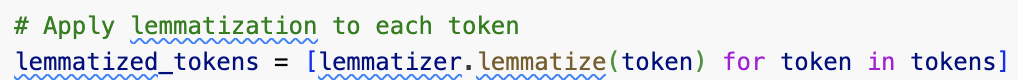
A close-up of a logo

Description automatically generated

* Tokenize the text into each token by splitting



* Apply lemmatization to each token



* + Studies 🡺 Study
  + Studying 🡺 Study
* Join lemmatized tokens into a string



**Bag of Words (BoW)**

It converts a document or corpus into a collection of tokens, ignoring grammar and word order, while keeping track of the frequency of each word.

1. **Word Order Ignored**:

* BoW treats the document as a “bag” of words, disregarding their sequence.
  + "NLP is fun" and "fun is NLP" have the same BoW representation.

1. **Frequency-Based Representation**:

* The model counts the occurrence of each word in the document.

1. **Vocabulary**:

* The unique words across the entire corpus form the vocabulary.
* The size of the vocabulary determines the dimensions of the BoW representation.

**n-Grams**

An n-gram is a contiguous sequence of n tokens (words or characters) from a given text.

**Types of n-Grams**

1. Unigram:

* Consists of single tokens.
  + "I love NLP" → ["I", "love", "NLP"]

1. Bigram:

* Consists of two consecutive tokens.
  + "I love NLP" → ["I love", "love NLP"]

1. Trigram:

* Consists of three consecutive tokens.
  + "I love NLP" → ["I love NLP"]

1. n-Gram:

* General case where n can be any positive integer
  + For n=4: "I love NLP tools" → ["I love NLP tools"]

**Initialize the n-grams**

* Tokenize the document using ***.split()***

A screen shot of a computer screen

Description automatically generated

* Choose unigram

A screenshot of a computer

Description automatically generated

* Choose bigram

A screenshot of a computer

Description automatically generated

Every time it moves to the next gram, it maintain the **same size of the n-gram but move by 1 step** until the token reaches to the end.

**TF (Term Frequency) and DF (Document Frequency)**

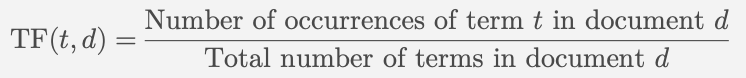
TF and DF are statistical measures that describe the importance of words in a document or corpus.

Example corpus: 2 documents in one corpus

A screenshot of a computer code

Description automatically generated

* TF: the # of times a specific term is seen per document 🡺 The # of times a **term t** appears in a **document d** in the corpus.



* + Count the # of vocabs in each document



* DF: The # of different **documents** containing the same **term t** in the corpus.



* + Count # of document with particular token in the corpus

A screen shot of a computer

Description automatically generated

**Motivation of TF**

* Common words like “the,” “is,” and “and” will have high frequencies, even though they provide little meaning.

**Motivation of DF:**

* It doesn’t consider how frequently the term appears in individual documents.
* Once we know how those terms without significant meaning, we can use DF to see how often they appear in each document.

**Stopwords**

* Stopwords are common words that carry little meaningful information in text analysis. These words often appear frequently across many documents, *such as “the,” “a,” “in”*.
* We tend to remove the stopwords since it will **lead high DF while containing no unique meaning and little understanding of context**.

**Remove stopwords using *.CountVectorizer(stop\_words = English)***

1. Use count vectorizer to **convert text data into a numerical representation** using the **BoW** model.

**A screen shot of a computer code

Description automatically generated**

* + - lowercase = True 🡺 Convert text to lowercase (default)

A group of black text

Description automatically generated

* + - ngram\_range = (1,1) 🡺 Extract unigrams (individual words)

A close-up of red text

Description automatically generated

* + - min\_df = 1 🡺 Include words appearing in at least 1 document
    - max\_df=1.0 🡺 Include words appearing in at most 100% of documents

1. get the vocabulary from the corpus

A screenshot of a phone

Description automatically generated

1. get the corresponding feature names regard to feature index

A screenshot of a computer

Description automatically generated

1. get term frequencies

A screenshot of a computer

Description automatically generated

1. transfer the term frequency into the vocab mapping

A screenshot of a phone

Description automatically generated

**Combining TF and DF:**

* TF-IDF helps normalize the importance of terms by considering both term frequency and document rarity.

**TF-IDF (Term Frequency-Inverse Document Frequency)**

TF-IDF is a statistical measure to evaluate the importance of a term in a document relative to a corpus. It helps identify terms that are informative by balancing term frequency (TF) and document frequency (DF). The goal is to **downweigh common terms that appear across many documents**, reducing their influence on the final text representation.

**Purpose of TF-IDF**

* Problem with Term Frequency (TF): Common words like “the,” “is,” and “and” appear frequently but don’t provide much information.
* Solution: Use **Inverse Document Frequency (IDF) to reduce the weight of frequent terms and highlight important ones**.

**TF-IDF function components**

1. **Term Frequency (TF):** How often a term t appears in document d.

A close-up of a number

Description automatically generated

1. **Inverse Document Frequency (IDF):** A measure that decreases the importance of terms appearing in many documents.

A math equation with numbers and symbols

Description automatically generated

* + N: total # of document
  + DF(t): # of document containing term t.

1. TF-IDF weight



Coding technique

* Initialize TF-IDF vectorizer

A screenshot of a computer code

Description automatically generated

* + Fit and transform the corpus with TF-IDF
  + Convert the input corpus with vocabs by default index.
* Calculate TF-IDF for each term frequency in 2 documents

A screenshot of a computer code

Description automatically generated

* + We can consider each document with feature importance by magnitude of TF-IDF relative to the whole corpus.
* Compare with the original term frequency

A screen shot of a computer

Description automatically generated

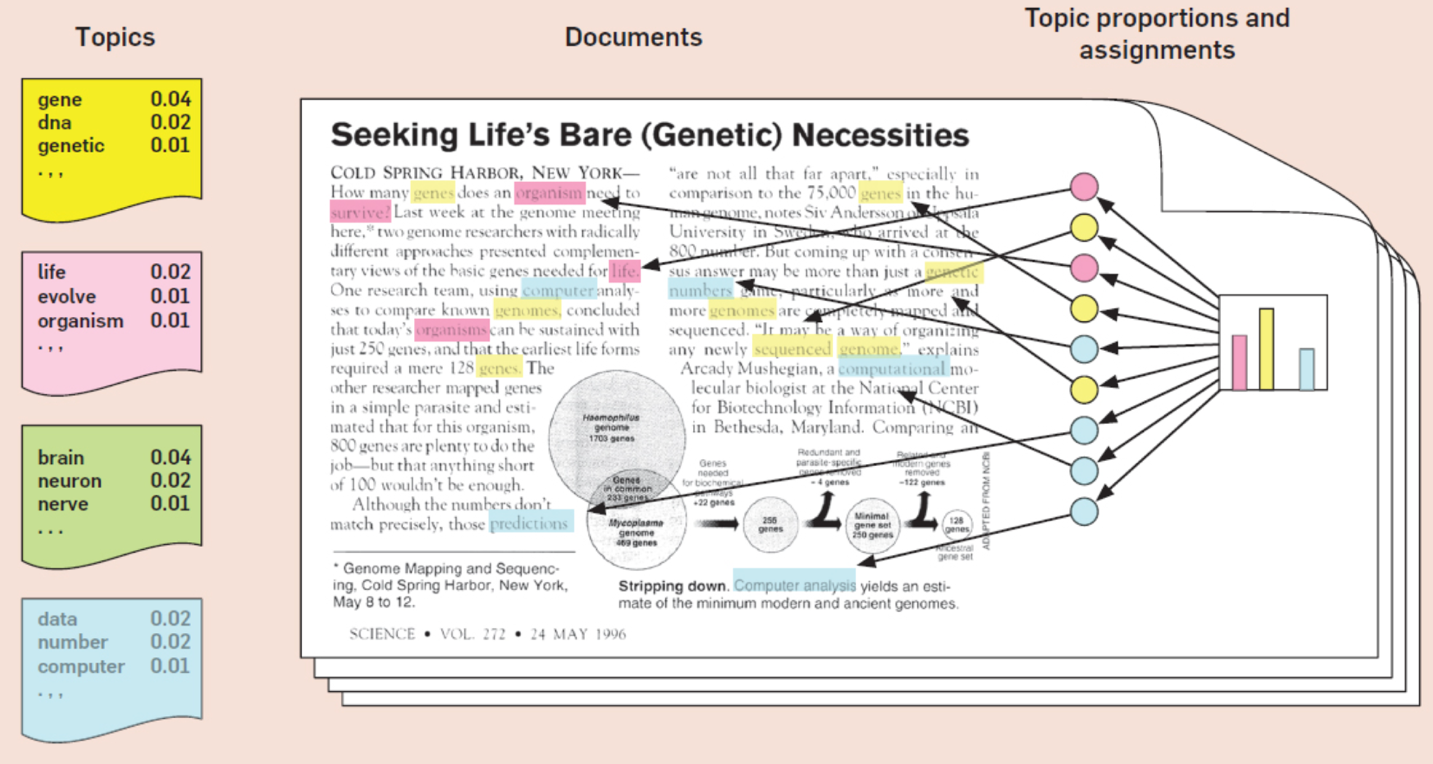
**Topic Modeling**

Topic Modeling is an **unsupervised learning** technique used to discover hidden topics in a collection of documents.

* It assumes that
  + ***Documents are composed of topics.***
  + ***Topics are collections of related terms.***
* Given the collection of documents
  + **Topic**:
    - ***A collection of terms that frequently appear together that define a topic***
    - ["data", "model", "analysis", "algorithm"]
  + **Term Distribution (Per Topic):**
    - ***What terms make up each topic?***
    - Topic 1 might include terms like {"data": 0.3, "model": 0.25, "analysis": 0.15}.
  + **Topic Distribution (Per Document):**
    - ***What topics make up each document?***
    - Document A might be {"Topic 1": 70%, "Topic 2": 30%}.
* This helps summarize, organize, and explore large datasets of text by automatically grouping terms into topics.

**Motivation**

1. **Understand Large Text Collections**: Extract topics from thousands of documents quickly.
2. **Dimensionality Reduction**: Represent documents using topic weights instead of individual terms.
3. **Text Summarization**: Extract key ideas from lengthy texts.
4. **Document Classification & Clustering**: Group similar documents into clusters based on shared topics.



**Latent Dirichlet Allocation (LDA)**

Assumes documents are mixtures of topics, and each topic is a mixture of terms, and the **distributions in topic modeling** are determined **based on term frequency patterns**.

* 2 important matrices learned by LDA

1. **Per topic term distribution (phi)**

A screenshot of a computer code

Description automatically generated

* + Phi describes how each term distributed in each topic

1. **Per document topic distribution (theta)**

A screen shot of a computer code

Description automatically generated

* + Theta indicates how each topic distributed in each document.

**Example of topic modeling in code**

1. Construct a topic, document, and vocab in the predefined corpus.

A computer screen shot of a computer code

Description automatically generated

* + We need to define the metrics for later LDA modeling

1. Per topic term distribution given documents and vocab

A screenshot of a computer

Description automatically generated

1. Per document topic distribution

A screenshot of a computer

Description automatically generated

**Purpose of LDA**

* **To predict the topics of documents based on their content** by identifying latent topics in a collection of documents.
* LDA assumes that documents are mixtures of topics, and each topic is a collection of related words.

**How LDA Works**

1. Input: A collection of documents (text data).
   * A set of documents
   * A number of topics K
2. Process:
   * LDA models each document as a distribution of topics (Theta): M \* K
   * Each topic is a distribution of words (Phi): K \* V
3. Output:
   * **Per-topic term distribution**: What words belong to each topic.
   * **Per-document topic distribution**: What topics describe each document.

LDA Example with Python

1. Define a corpus with libs

A screen shot of a computer code

Description automatically generated

1. Create count vectorizer by removing stop words

A close-up of words

Description automatically generated

1. Initialize LDA with 2 topics

A computer screen shot of a computer code

Description automatically generated

* + n\_components = 2 indicate we will generate 2 topics from the vectorized corpus.

1. We need to display the topics consisted of top terms based on the LDA

A screen shot of a computer code

Description automatically generated

* + Based on each term for topic, we can come up with the general topic name specifically.