
NLP

Last lecture reminder

We learned about:

- NLP - Introduction
- SpaCy Library - Introduction
- SpaCy - Installation & Import
- SpaCy - Basic Functionality and Usage
- SpaCy - Tokenization
- SpaCy - Noun Chunks
- SpaCy - Display
- SpaCy - Stemming & Lemmatization
- SpaCy - Stop Words
- SpaCy - Phrase Matching

SpaCy - Part of Speech (POS)

Part of Speech (POS) → Part of Speech (POS) tagging is a crucial linguistic feature that involves assigning a part of speech to each token in a text. This helps in understanding the grammatical structure and syntactic relationships within sentences. POS tags provide valuable information for various NLP tasks such as parsing, named entity recognition, and sentence parsing.

SpaCy assigns POS tags using a statistical model trained on large annotated corpora. Each token is tagged with a coarse-grained POS tag and a fine-grained morphological analysis.
There are 2 types of POS properties:

- **Coarse-Grained Tag (POS)** → This represents the major part of speech category (NOUN, VERB, ADJ, etc…). In SpaCy this referred to the **‘pos_’** attribute.
- **Fine-Grained Tag (Tag)** → This provides more detailed information, including subcategories and morphological features (number, gender, etc…). In SpaCy this referred to the **‘tag_’** attribute.

SpaCy - Part of Speech (POS)

Corpus → A corpus (plural: corpora) is a large and structured set of texts (written or spoken) that are used for linguistic research and NLP tasks. Corpora serve as primary sources of data for empirical studies of language, allowing researchers and developers to analyze language patterns, train machine learning models, and validate hypotheses about language use.

```
In [92]: import spacy
         nlp = spacy.load('en_core_web_sm')

         doc = nlp("The quick brown fox jumped over the lazy dog's back.")

         print(doc[4].text, doc[4].pos_, doc[4].tag_, spacy.explain(doc[4].tag_))

         jumped VERB VBD verb, past tense
```

SpaCy decided that the token ‘jumped’ is a verb with VBD tag which mean it’s a past verb

```
In [97]: doc1 = nlp(u'I read books on NLP.')
         doc2 = nlp(u'I read a book on NLP.')

         print(f'{doc1[1].text:{10}} {doc1[1].pos_:{8}} {doc1[1].tag_:{6}} {spacy.explain(doc1[1].tag_)}')
         print()
         print(f'{doc2[1].text:{10}} {doc2[1].pos_:{8}} {doc2[1].tag_:{6}} {spacy.explain(doc2[1].tag_)}')

         read       VERB     VBP    verb, non-3rd person singular present

         read       VERB     VBD    verb, past tense
```

SpaCy is smart enough to understand from the text context that the first ‘read’ token is in present time and the second ‘read’ token is in past time

Named Entity Recognition (NER)

Named Entity Recognition (NER) → NER is a subtask of information extraction that seeks to locate and classify named entities mentioned in unstructured text into predefined categories. These categories include names of persons, organizations, locations, dates, times, quantities, monetary values, percentages, and more. NER is a fundamental component of various Natural Language Processing (NLP) systems and applications.

NER key concepts:

- **Named Entities** → Creating the named entities dictionary with the entity name as key and a list of values. For example, Persons (PER) - Names of people ("Albert Einstein", "Marie Curie"), Locations (LOC) - Names of geographical places ("Paris", "Mount Everest"), etc…
- **Recognition** → The process of detecting the presence of these entities in text.
- **Classification** → The process of assigning these entities to predefined categories or classes.

Named Entity Recognition (NER)

How NER works?

NER involves several steps, typically implemented using machine learning models, rule-based systems, or a combination of both:

1. **Tokenization** → Breaking the text into tokens (words, punctuation marks, etc…).
2. **Feature Extraction** → Generating features from the tokens that can be used by machine learning models. These features may include: Word itself, Part of Speech (POS) tags, Capitalization, Prefixes and suffixes.
3. **Model Training** → Training a machine learning model on annotated corpora where entities are labeled with their corresponding types.
4. **Prediction** → Using the trained model to recognize and classify named entities in new text.

NER in SpaCy - Python Example

Every entity token (token that been classified by SpaCy as an entity) will have a property called **'label_'** which provide the entity key name. In addition, we can use the **explain()** method to see the entity key name explanation. We can see all the entities in a specific doc by calling the **'ents'** property.

Now let's see some Python examples that handle entities with SpaCy:

First we will create a dedicated function that will help us print all the entities classification in a given doc object.

```
In [100]: def show_ents(doc):
              if doc.ents:
                  for ent in doc.ents:
                      print(ent.text+' - '+ent.label_+' - '+str(spacy.explain(ent.label_)))
              else:
                  print('No named entities found.')
```

```
In [101]: doc1 = nlp('Hello, where are you from?')
          doc2 = nlp('May I go to Washington, DC next May to see the Washington Monument?')

          print('doc1 entities:')
          show_ents(doc1)
          print()
          print('doc2 entities:')
          show_ents(doc2)
```

```
doc1 entities:
No named entities found.

doc2 entities:
Washington, DC - GPE - Countries, cities, states
next May - DATE - Absolute or relative dates or periods
the Washington Monument - ORG - Companies, agencies, institutions, etc.
```

Entities SpaCy been able to classify from the second doc object.

NER in SpaCy - NER Customization

SpaCy also providing us the ability to modify the entity recognition and add additional entities to it.

This can be very useful when we have unrecognized entities that we still want SpaCy to recognize for us.

For example → In the following doc 'Tesla' is not been recognized as a company entity.

```
In [9]: doc = nlp(u'Tesla to build a U.K. factory for $6 million')

        show_ents(doc)

        U.K. - GPE - Countries, cities, states
        $6 million - MONEY - Monetary values, including unit
```

We can see that 'Tesla' is not part of the recognized entities in the doc

In case we want to allow SpaCy to recognize the 'Tesla' entity as a company in the doc we will need to manually adding it to the doc entity list with the right label value.

For doing it, we need to execute the following steps:

1. Get the hash value of the entity label we want to customize (In our case 'ORG').
2. Create a span for the new entity with the wanted label hash value.
3. Add the new span entity to SpaCy entity list.

NER in SpaCy - NER Customization

Let's see this in action:

```
In [5]: from spacy.tokens import Span

        ORG = doc.vocab.strings[u'ORG']

        new_ent = Span(doc, 0, 1, label=ORG)

        doc.ents = list(doc.ents) + [new_ent]
```

Fetch the 'ORG' entity hash value

Create a span from the 'Tesla' token and add the wanted label value

Add the new span entity to entity doc list

Now let's execute again the show_ents() function:

```
In [6]: print(f'This is "ORG" entity hash value: {ORG}')
        show_ents(doc)

        This is "ORG" entity hash value: 383
        Tesla - ORG - Companies, agencies, institutions, etc.
        U.K. - GPE - Countries, cities, states
        $6 million - MONEY - Monetary values, including unit
```

We can see that now SpaCy recognize 'Tesla' as an ORG entity

ECOM SCHOOL

המכללה למקצועות הדיגיטל וההייטק

NER in SpaCy - NER Customization

Now, let's take a look at another example, but this time we will use the phrase matcher to add entity recognition based on token pattern.

For example → Let's say we want SpaCy to recognize the tokens: 'dashboard-website' and 'dashboard website' as a PRODUCT entitis.

```
In [10]: doc = nlp(u'Our company plans to introduce a new dashboard-website. '
                   u'If successful, the dashboard website will be our main cutomer payed product')

         show_ents(doc)

         No named entities found.
```

Right now we can see that SpaCy is not recognize those tokens as entities of any kind.

For adding them, we will first create phrase matchers for each pattern:

```
In [21]: from spacy.matcher import PhraseMatcher

         matcher = PhraseMatcher(nlp.vocab)

         phrase_list = ['dashboard website', 'dashboard-website']
         phrase_patterns = [nlp(text) for text in phrase_list]

         matcher.add('newproduct', phrase_patterns)

         matches = matcher(doc)

         matches

Out[21]: [(2689272359382549672, 7, 10), (2689272359382549672, 15, 17)]
```

We created a phrase pattern for each token and added all patterns to the matcher list

SpaCy managed to successfully recognized both phrases in the doc

NER in SpaCy - NER Customization

Now, we will use the mathes tuple values to add them manually to the doc entities list:

```
In [25]: from spacy.tokens import Span

         PROD = doc.vocab.strings['PRODUCT']

         new_ents = [Span(doc, match[1],match[2],label=PROD) for match in matches]

         doc.ents = list(doc.ents) + new_ents

         show_ents(doc)

         dashboard-website - PRODUCT - Objects, vehicles, foods, etc. (not services)
         dashboard website - PRODUCT - Objects, vehicles, foods, etc. (not services)
```

We created an entities list from each match value and added to it the label ‘PRODUCT’ hash value

We can see that now SpaCy recognized both ‘dashboard-website’ and ‘dashboard website’ as ‘PRODUCT’ entities

We can also use the **‘doc.ents’** list to count how many entities from a specific value we have in the doc:

```
In [29]: doc = nlp('Originally priced at $29.50, the sweater was marked down to five dollars.')

         show_ents(doc)

         29.50 - MONEY - Monetary values, including unit
         five dollars - MONEY - Monetary values, including unit
```

```
In [30]: len([ent for ent in doc.ents if ent.label_=='MONEY'])

Out[30]: 2
```

We have in total 2 entities of the label ‘MONEY’ in the doc

Sentence Segmentation

Sentence segmentation → Sentence segmentation, also known as sentence boundary disambiguation or sentence breaking, is a process in NLP that divides a piece of text into its individual sentences. Its goal is to identify where one sentence ends and another one begins.

The common approach in sentence segmentation is using punctuation marks such as periods, question marks, and exclamation marks. However, this simple method can be ineffective due to the complexity of human language. For example, periods can be used in abbreviations, decimals, ellipsis, etc… not just to indicate the end of a sentence. Therefore, more advanced models also consider other factors like capitalization, context, or words that typically appear at the beginning of a sentence.

Proper sentence segmentation is crucial for further NLP tasks such as machine translation, text summarization, sentiment analysis, etc. because they typically work on a per-sentence level. So, without accurate sentence segmentation, the performance of these tasks could be severely affected.

Sentence Segmentation - Add Segmentation Rule

In order to understand why we will want to add additional segmentation rules to SpaCy, let's take a look at the following examples:

First we have an example of a proper basic sentence segmentation been done on '.' (dot).

SpaCy provide all different sentences in a generator called **doc.sents**.

```
In [32]: doc = nlp('This is the first sentence. This is another sentence. This is the last sentence.')

         for sent in doc.sents:
             print(sent)

         This is the first sentence.
         This is another sentence.
         This is the last sentence.
```

What is also important is to understand that doc.sents is not a list of sentences (span) but an actual object.

In case we want to have all span sentences in a list we need to generate it manually:

```
In [34]: doc_sents = [sent for sent in doc.sents]
         doc_sents

Out[34]: [This is the first sentence.,
          This is another sentence.,
          This is the last sentence.]
```

Now all the span sentences are in a list, so we can now fetch sentence by index and execute other list functionalities.

Sentence Segmentation - Add Segmentation Rule

Now let's take a look at the following example:

```
In [35]: doc3 = nlp('"Management is doing things right; leadership is doing the right things." -Peter Drucker')

         for sent in doc3.sents:
             print(sent)

         "Management is doing things right; leadership is doing the right things."
         -Peter Drucker
```

SpaCy basic segmentation rule decided to split the sentences on " " creating 2 different sentences

The problem with the basic SpaCy segmentation is that we can see that the first sentence is also combined from 2 different sentences separated by ' ; ' (semi-colon).

In order to allow SpaCy to divide the doc as we want, we need to add an additional rule that will split the doc on semi-colon as well.

First we will create the rule function and annotate it as a **Language.component()**.

```
In [45]: from spacy.language import Language

         @Language.component('set_custom_boundaries')
         def set_custom_boundaries(doc):
             for token in doc[:-1]:
                 if token.text == ';':
                     doc[token.i+1].is_sent_start = True
             return doc
```

The rule function is iterating on every token in the doc (except the last one) and check if the token is semi-colon.
In case the token is a semi-colon the function mark the next token as start of a sentence with the property **'is_sent_start'** = True

Sentence Segmentation - Add Segmentation Rule

Language.component annotation → The `@Language.component` annotation is used to register a user-defined function as a component in a spaCy pipeline. This is particularly useful if you want to add a custom pipeline component and need spaCy to recognize this function.

SpaCy pipeline → In SpaCy, the "pipeline" refers to the series of operations or stages the library performs on a block of text to convert it from raw text to information that your programs can work with. Each operation is performed by a component. When a `Doc` object is created, it is automatically processed by the pipeline. The operations performed include: Tokenization, POS, Dependency Parsing, Lemmatization, NER and Sentence Segmentation.

We can customize the pipeline by disabling components we don't need for a particular task or by adding new components such as text classification, sentiment analysis, or your custom functions. The components are completed in order, and the `Doc` object is modified at each step, with the results of each component being attached to the `Doc` object. Therefore, the order in which components are applied can be significant.

Sentence Segmentation - Add Segmentation Rule

Now that we created a new segmentation rule and annotate it as SpaCy pipeline component, we can now add it as an additional component in SpyCy pipeline.

For that we need to specify the component name and the place in order that we want to add it to.

```
In [3]: nlp.add_pipe('set_custom_boundaries', before='parser')

        nlp.pipe_names

Out[3]: ['tok2vec',
         'tagger',
         'set_custom_boundaries',
         'parser',
         'attribute_ruler',
         'lemmatizer',
         'ner']
```

We added a new pipeline component called ‘set_custom_boundaries’ right before the ‘parser’ component in the pipeline.

Now, let’s run again SpaCy sentence segmentation and see if we managed to change the segmentation rule:

```
In [4]: doc4 = nlp(u'"Management is doing things right; leadership is doing the right things." -Peter Drucker')

        for sent in doc4.sents:
            print(sent)

        "Management is doing things right;
        leadership is doing the right things."
        -Peter Drucker
```

We can see that now SpaCy managed to recognized the semi-colon as another sentence segmentation and splittet the spans accordingly.

Class Exercise - NER & Segmentation Rules

Instructions:

- Identify all named entities in a given text, for each entity print the entity explanation and the entity hash value as well.

  Text: "Barack Obama was born on August 4, 1961, in Honolulu, Hawaii."
- Extend SpaCy NER to identify the following entities in the doc →

  Text: "Maccabi Tel Aviv played against Hapoel Tel Aviv in the finals of the National Basketball League", **Entities:** ‘Maccabi Tel Aviv’ & ‘Hapoel Tel Aviv’ should be Sport teams.

  Use phrase matcher in your solution.
- Add customized sentence segmentation rule to properly segment the following doc →

  Text: "User1: hello User2: Hey how are you? User1: I'm good how are you? User2: I'm good as well, thanks".

  Print the original sentence segmentation and your changed segmentation to make sure your logic worked.

Class Exercise - NER & Segmentation Rules

Instructions:

- Print how many entities of type **‘ORG’** and of type **‘PRODUCT’** there are in this doc →

  Text: "Google announced new features for its product line including the Pixel 5 and Google Home. Meanwhile, Apple released the iPhone 13 and MacBook Pro in a recent event. Microsoft introduced updates to the Surface Pro and Windows 11. Amazon continues to expand its services, and Facebook is planning to rebrand its products to Meta. Samsung unveiled the Galaxy S21 and Galaxy Buds Pro."

- For each entity print it’s start and end position.
- Provide entity visualization for all the ‘ORG’ and ‘PRODUCT’ entities.

Class Exercise Solution - NER & Segmentation Rules

Text Classification

Text classification → Text classification is a fundamental task in the field of NLP. It involves assigning predefined categories or labels to textual data based on its content. The process is been done using common supervised learning classification models.

Key Concepts:

- **Categories / Labels** → These are the predefined classes that textual data can be classified into. Examples include `positive`/`negative` for sentiment analysis, `sports`/`politics` for news categorization, and `spam`/`not spam` for email filtering.
- **Features** → Features are the characteristics or attributes used to represent the text data numerically. Common features include word frequencies, n-grams, TF-IDF values, and embeddings (like those from Word2Vec, GloVe, or BERT).
- **Model** → The model is an algorithm trained on a labeled dataset to learn patterns that can be applied to classify new, unseen text data. Common algorithms include Naive Bayes, Support Vector Machines (SVM), Random Forests, and deep learning models like LSTM.

Text Classification - Feature Extraction

Feature extraction → Feature extraction refers to the process of transforming raw text into numerical representations or features that can be used as inputs for machine learning models. This is a crucial step because machine learning algorithms require numerical data to perform analyses, make predictions, and identify patterns.

Common Techniques for Feature Extraction in NLP:

- **Tokenization** → Splitting the text into individual units called tokens (words, phrases, or characters).
- **Bag of Words (BoW)** → Represents text as an unordered collection of words, without considering grammar or word order.
- **Term Frequency-Inverse Document Frequency (TF-IDF)** → Measures the importance of a word in a document relative to a collection of documents (corpus).
- **Part-of-Speech Tagging (POS Tagging)** → Assigns parts of speech (like noun, verb) to each word in a text.
- **Named Entity Recognition (NER)** → Identifies and classifies entities like names of people, organizations, locations within text.

Text Classification - Feature Extraction

In the previous lectures we already talked about tokenization, POS tagging and NER so we are going to focus now on **Bag of Words (BoW)** and **Term Frequency-Inverse Document Frequency (TF-IDF)** techniques.

In **Bag of Words (BoW)** feature extraction technique we are going to count the occurrences of each individual word in the text. By doing it we will transform the raw text data into a matrix of token (word) counts. This will enable us to provide numerical data that is suitable for machine learning.
This process can also be called **Count Vectorization**.
How Count Vectorization Works:

1. **Tokenization** → The text is split into individual tokens, typically words, but can also include phrases or characters.
2. **Vocabulary Creation** → A vocabulary, or a corpus-wide list of all unique tokens from the training dataset, is created.
3. **Counting** → For each document, the frequency of each token (from the vocabulary) is counted.

Text Classification - Feature Extraction

For example → Let's consider we have the following text files:

- **Document 1:** "I love NLP"
- **Document 2:** "NLP is fun"

Now, let's consider we have the following vocabulary that we managed to collect from multiple previous training datasets:

- **Vocabulary:** ["I", "love", "NLP", "is", "fun"]

Finally, let's run count vectorization on the two documents and create the **Count Vector Matrix**:

- **Count Vector Matrix**:

"I"	"love"	"NLP"	"is"	"fun"
1	1	1	0	0	-> Document 1
0	0	1	1	1	-> Document 2

Text Classification - Feature Extraction

Count Vector Matrix → Count vector matrix is a matrix representation of the text corpus where each row corresponds to a document, and each column corresponds to a unique token (word) from the corpus vocabulary. The elements of the matrix are the counts of how many times each token appears in each document. The matrix is the output result of performing count vectorization.

Now that we understand what is Bag of Words (BoW), let's discuss what is **Term Frequency-Inverse Document Frequency (TF-IDF)**.

The **TF-IDF** feature extraction technique help us to evaluate the importance of a word in a document relative to a collection of documents (corpus). Unlike basic Count Vectorization, which merely counts the occurrences of words, TF-IDF adjusts these counts by considering how often the words appear across all documents in the corpus. This helps to give more weight to words that are more informative and less weight to common words.

Text Classification - Feature Extraction

Why we want to also use TF-IDF feature extraction?

The problem with the count vectorization feature extraction is that it can give specific words unjustified importance. For example, words like ‘the’ or ‘a’ can repeat in multiple documents and should not be marked as important because they are common stop words in english. Count vectorization doesn’t have the ability to differentiate between common words and important words. This is why TF-IDF feature extraction can help us.

How TF-IDF been calculated?

1. **Calculate the TF (Term Frequency)** → For each document, calculate the frequency of each token. This is similar to the counting step in BoW.
2. **Calculate the IDF (Inverse Document Frequency)** → Calculate the inverse document frequency of each token.

$$idf(t, D) = log\frac{N}{|\{d \in D : t \in d\}|}$$

The number of documents (N), divided by the number of times the token appear in all the documents

3. **Calculate the TF-IDF** → Compute the TF-IDF score for each token in each document by multiplying the TF and IDF values. (TF-IDF score = TF * IDF)

Text Classification - Feature Extraction

For example → Let's consider we have the following text files:

- **Document 1:** "I love NLP"
- **Document 2:** "NLP is fun"

Now, let's consider we have the following vocabulary that we managed to collect from multiple previous training datasets:

- **Vocabulary:** ["I", "love", "NLP", "is", "fun"]

The **TF** of each token in each document is the following:

- **Document 1** → TF("I", D1) = 1/3 , TF("love", D1) = 1/3 , TF("NLP", D1) = 1/3 , TF("is", D1) = 0, TF("fun", D1) = 0
- **Document 2** → TF("I", D2) = 0, TF("love", D2) = 0, TF("NLP", D2) = 1/3, TF("is", D2) = 1/3, TF("fun", D2) = 1/3

Text Classification - Feature Extraction

Now let's calculate the **IDF** for each token in both documents:

IDF("I") = log(2/1) = 0.3010

IDF("love") = log(2/1) = 0.3010

IDF("NLP") = log(2/2) = 0

IDF("is") = log(2/1) = 0.3010

IDF("fun") = log(2/1) = 0.3010

Finally, let's calculate the **TF-IDF score** for each token:

- **Document 1 →**

  TF-IDF("I", D1) = (1/3) * 0.3010 ≈ 0.1003

  TF-IDF("love", D1) = (1/3) * 0.3010 ≈ 0.1003

  TF-IDF("NLP", D1) = (1/3) * 0 ≈ 0

  TF-IDF("is", D1) = 0 * 0.3010 = 0

  TF-IDF("fun", D1) = 0 * 0.3010 = 0

Document 2 →

TF-IDF("I", D2) = 0 * 0.3010 = 0

TF-IDF("love", D2) = 0 * 0.3010 = 0

TF-IDF("NLP", D2) = (1/3) * 0 ≈ 0

TF-IDF("is", D2) = (1/3) * 0.3010 ≈ 0.1003

TF-IDF("fun", D2) = (1/3) * 0.3010 ≈ 0.1003

Text Classification - Feature Extraction

Now that we have the TF-IDF scores of each token in the documents we can build the **TF-IDF matrix**:

"I"	"love"	"NLP"	"is"	"fun"
0.1003	0.1003	0	0	0	-> Document 1
0	0	0	0.1003	0.1003	-> Document 2

Beside applying feature extraction, building the vocabulary for the extraction is also an important task

The steps for building the vocabulary are:

1. Gather a comprehensive and representative set of documents (text data) that will serve as the training set for building the vocabulary.
2. Convert all text to lowercase to ensure consistency
3. Add each unique word from all documents to a dedicated dictionary and give to each word a unique id.

Feature Extraction - Building A Vocabulary

Let's build a simple vocabulary using Python:

```
In [9]: %%writefile 1.txt
        This is a story about cats
        our feline pets
        Cats are furry animals

        Overwriting 1.txt
```

```
In [10]: %%writefile 2.txt
         This story is about surfing
         Catching waves is fun
         Surfing is a popular water sport

         Overwriting 2.txt
```

Using the '%%writefile' Python command we generated 2 'txt' files representing 2 different documents

```
In [11]: vocab = {}
         i = 1

         with open('1.txt') as f:
             x = f.read().lower().split()

         for word in x:
             if word in vocab:
                 continue
             else:
                 vocab[word]=i
                 i+=1

         print(vocab)

         {'this': 1, 'is': 2, 'a': 3, 'story': 4, 'about': 5, 'cats': 6, 'our': 7, 'feline': 8, 'pets': 9, 'are': 10, 'furr
         y': 11, 'animals': 12}
```

In this code we iterate on each word in the first 'txt' file, perform lowercase and add it to the vocab dictionary with a unique number as id

The dictionary result will contain all the unique words with unique id for each word

Feature Extraction - Python Example

```
In [12]: with open('2.txt') as f:
             x = f.read().lower().split()

         for word in x:
             if word in vocab:
                 continue
             else:
                 vocab[word]=i
                 i+=1

         print(vocab)
```

```
{'this': 1, 'is': 2, 'a': 3, 'story': 4, 'about': 5, 'cats': 6, 'our': 7, 'feline': 8, 'pets': 9, 'are': 10, 'furr
y': 11, 'animals': 12, 'surfing': 13, 'catching': 14, 'waves': 15, 'fun': 16, 'popular': 17, 'water': 18, 'sport':
19}
```

We repeat the process on the same dictionary instance for all documents

The final dictionary result will contain all unique words from all the documents

Now that we understand how perform feature extraction, let's start to train a supervised learning model on a dataset using feature extraction.

For this example we will use the **'smsspamcollection.tsv'** file.

This file contain data about emails like the email message itself, the length and the punctuation of the email.

In addition, the dataset contain data about each email label as 'ham' (legitimate email) or 'spam' (spam email).

Our mission is to use supervised learning model to predict if future emails should be classified as 'spam' or not.

Feature Extraction - Python Example

Once we have our train / test split datasets we can apply feature extraction. We will use the **CountVectorizer** to apply **Bag of Words (BoW)** feature extraction.

Keep in mind that the result Count Vector Matrix can be very large so its not recommended to print it.

```
In [21]: from sklearn.feature_extraction.text import CountVectorizer
         count_vect = CountVectorizer()

         X_train_counts = count_vect.fit_transform(X_train)
         X_train_counts.shape

Out[21]: (3733, 7082)
```

The Count Vector Matrix shape is 3733 X 7082.
3733 → The number of datapoints in the training set.
7082 → The number of unique words in the vocabulary built from the entire dataset.

The fit_transform action combine the following actions:
fit() → Build the vocabulary from the provided dataset, count the number of words in each row.
transform() → Transforming the original message text column to the corresponding count vector of that datapoint.

Feature Extraction - Python Example

In case we want to apply TF-IDF feature attraction we can use the **TfidfTransformer** on the CountVectorizer results or we can perform both BoW and TF-IDF in one command by using the **TfidfVectorizer**.

```
In [22]: from sklearn.feature_extraction.text import TfidfTransformer
         tfidf_transformer = TfidfTransformer()

         X_train_tfidf = tfidf_transformer.fit_transform(X_train_counts)
         X_train_tfidf.shape

Out[22]: (3733, 7082)
```

We transformed the CountVectorizer results to TF-IDF vectorizer results using the TfidfTransformer

```
In [25]: from sklearn.feature_extraction.text import TfidfVectorizer
         vectorizer = TfidfVectorizer()

         X_train_tfidf = vectorizer.fit_transform(X_train)
         X_train_tfidf.shape

Out[25]: (3733, 7082)
```

We can also combine the two together using the TfidfVectorizer. The result will be the same as with the TfidfTransformer.

ECOM SCHOOL

המכללה למקצועות הדיגיטל וההייטק

Feature Extraction - Python Example

Now that our TF-IDF features are ready we can move to a regular supervised learning model.

For this example we will use the **Linear SVC (Support Vector Classifier)** model.

```
In [26]: from sklearn.svm import LinearSVC
         clf = LinearSVC()
         clf.fit(X_train_tfidf,y_train)

Out[26]: ▾ LinearSVC
         LinearSVC()
```

In order to test our model performance we will need to transform original text feature to the TF-IDF feature extraction as well.

To make our job more easy we can create a **‘pipeline’** that will have both the TF-IDF task

and the SVC training model task in one pipeline instance. This will save us unnecessary code and will make our training process more organized.

Feature Extraction - Python Example

The pipeline instance is a very powerful tool provided by Scikit-Learn. It can understand from the instances we provided what it should call in each task without the need to manually specify it.

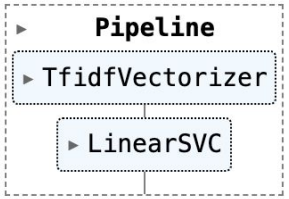
For example → In the following pipeline we configure 2 tasks the TF-IDF transformer and the LinearCSV estimator (model).

```
In [28]: from sklearn.pipeline import Pipeline

         svc_pipeline = Pipeline([('tfidf', TfidfVectorizer()),
                                  ('svc', LinearSVC()),
         ])

         svc_pipeline.fit(X_train, y_train)
```

Out[28]:



The pipeline knows that when we call the fit() method it should apply fit_transform() on the data provided because the 'tfidf' instance is a transformer.
The pipeline will use the output of the 'tfidf' as the input to the 'svc' task and perform the fit() method because its an estimator.

```
In [39]: predictions = svc_pipeline.predict(X_test)
         predictions

Out[39]: array(['ham', 'ham', 'ham', ..., 'ham', 'ham', 'ham'], dtype=object)
```

When calling the predict() method the pipeline is smart enough to use the fit() results from previously because it knows that it has already a trained model.
It will just execute a transform() on the test dataset and predict() on the trained 'svc' model.

Feature Extraction - Python Example

Finally, we can use classification metrics to evaluate our model performance:

```
In [40]: from sklearn import metrics

         print(metrics.confusion_matrix(y_test,predictions))
         print()
         print(metrics.classification_report(y_test,predictions))
         print()
         print(metrics.accuracy_score(y_test,predictions))
```

```
[[1586    7]
 [  12  234]]

              precision    recall  f1-score   support

         ham       0.99      1.00      0.99      1593
        spam       0.97      0.95      0.96       246

    accuracy                           0.99      1839
   macro avg       0.98      0.97      0.98      1839
weighted avg       0.99      0.99      0.99      1839


0.989668297988037
```

We can see that our feature extraction model perform very good with above 98% accuracy as well as F1-score and recall

We can also test it on unknown new data and see its predictions:

```
In [63]: svc_pipeline.predict([
             "TEXT WON to 12345 to get your prise, you have been selected as WINNER!!"
         ])

Out[63]: array(['spam'], dtype=object)
```

```
In [62]: svc_pipeline.predict(["Hey Tom, how are you today?"])

Out[62]: array(['ham'], dtype=object)
```

Class Exercise - Feature Extraction

Instructions:

- Proform manual calculate to create the **Count Vector Matrix** and the **TF-IDF Matrix**

  Of the following documents:

  1. "The cat sat on the mat"
  2. "The dog barked at the cat"
  3. "The cat and dog sat together"

- Use the **‘moviereviews.tsv’** file that provide customers reviews about movies and label that indicate if the review was positive (pos) or negative (neg).
- Apply TF-IDF feature extraction on the review column and simple logistic regression classification model to predict of a review is positive or negative.
- Configure a Scikit-Learn pipeline to train your model
- Evaluate your model performance with different classification metrics.

Class Exercise Solution - Feature Extraction