



Natural Language Processing Techniques and Applications

Hello HAL! Was the first appearance of a speaking and thinking robot in the 1968 film 2001: A Space Odyssey. HAL (Heuristically programmed ALgorithmic computer) was a sentient artificial general intelligence computer that controlled the systems of the Discovery One spacecraft and interacted with the ship's astronaut crew. In fact, NLP received widespread recognition in the 1950s, when researchers and linguistics experts began developing machines to automate language translation.

Today, Natural language processing (NLP) is a branch of artificial intelligence aimed at giving computers the ability to use and understand human language and speech. Technology features we take for granted every day are a product of NLP and deep learning. NLP field encompass the entire cycle of recognizing and understanding human speech, processing natural language, and generating text that can be read and interpreted by humans. Whenever you dictate a text message to Siri or ask Alexa the weather, that's natural language processing. When our email services filter out spam, check our spelling and grammar, and even autocomplete entire messages, that's NLP too.

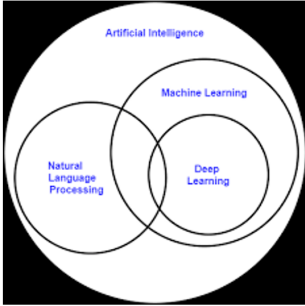
A common data type (especially on the internet) is text data. Text data representation is important, but it is also challenging. It requires preprocessing and specific domain knowledge.

What is NLP?

NLP is giving computers the ability to use and understand human language and speech. Common, everyday NLP examples are:

- dictate a text message to Siri or ask Alexa the weather
- when our email services filter out spam,
- check our spelling and grammar, and even autocomplete entire messages

What is NLP?



The diagram consists of four overlapping circles. A large circle at the top is labeled 'Artificial Intelligence'. Inside it are two smaller circles: 'Natural Language Processing' on the left and 'Machine Learning' on the right. Within the 'Machine Learning' circle is a smaller circle labeled 'Deep Learning'.

- **Natural language processing (NLP)** – is the intersection of artificial intelligence, computer science, and linguistics.
- **NLP** is giving computers the ability to use and understand human language and speech
- Common, everyday NLP examples are:
 - dictate a text message to Siri
 - ask Alexa the weather
 - when our email services filter out spam, check our spelling and grammar, and even autocomplete entire messages

Figure 1 – The NLP Defined

NLP has been growing steadily for the past couple of decades due to the explosive growth of textual data called “unstructured” data. Today, 80% of all data created are unstructured like twitter and newsfeeds and reports, etc. as shown below (courtesy of Ravenpack).

Why NLP?

A growth history of machine learning and NLP is shown below:

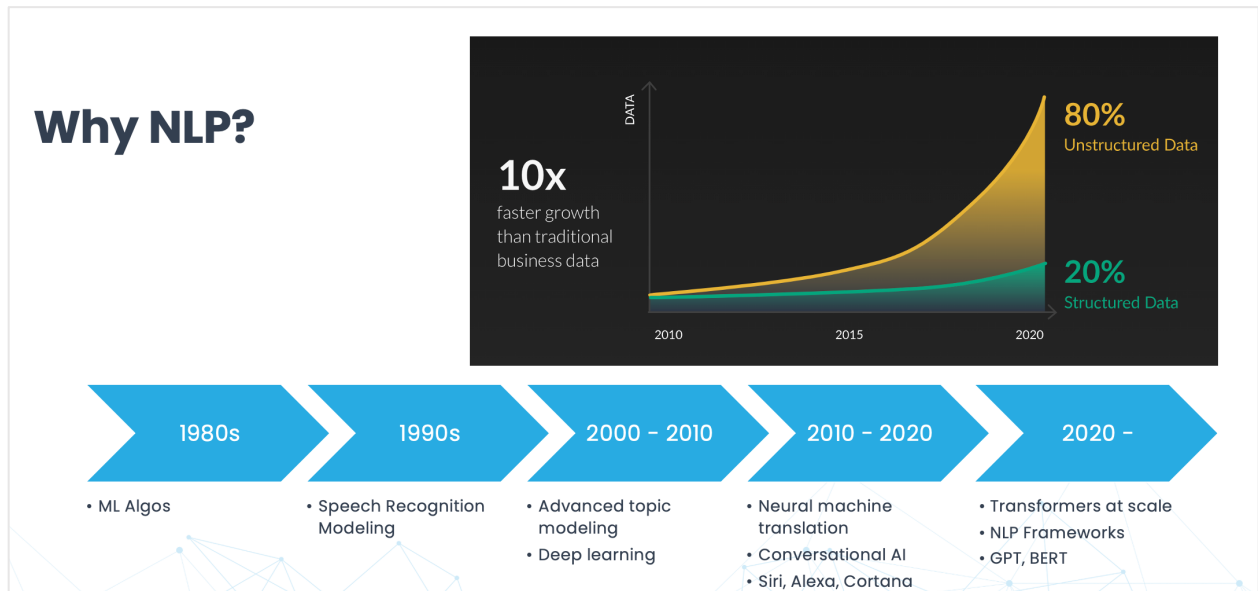


Figure 2 – Why NLP?

Natural Language Processing hit its big stride back in 2017 with the introduction of Transformer Architecture from Google called BERT. Since then, OpenAI introduced GPT series. These platforms have been built on what is called the Transformer architecture with performance results that handily beat existing state-of-the-art benchmarks.

Transformers have taken the world of NLP by storm in the last few years. Now they are being used with success in applications beyond NLP as well. These state-of-the-art approaches have helped bridge the gap between humans and machines and helped us build bots capable of using human language undetected. It's an exciting time. In addition, new frameworks have been developed for NLP to continue that innovation.

Language is one of the great untapped resources of information. With the rich and growing ecosystem, we can now access this raw data, with many options to process this kind of data.

Key Capabilities of Natural Language Processing

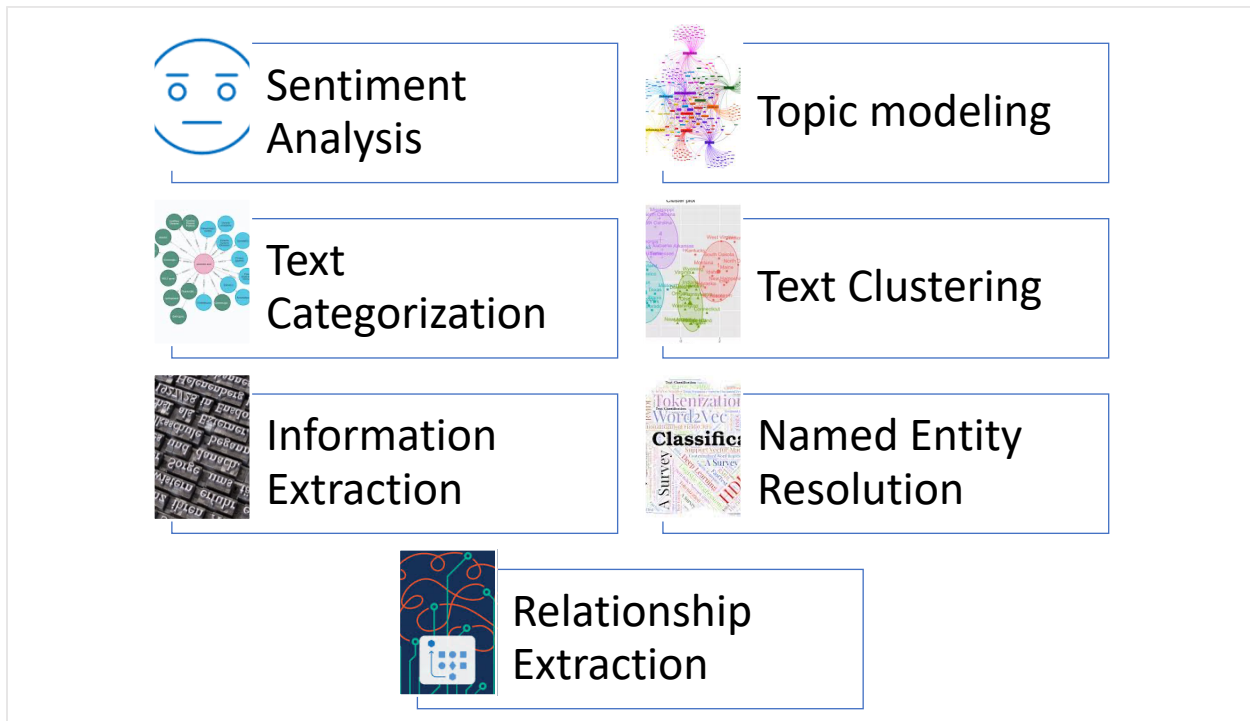


Figure 3 – Key Technical Capabilities of NLP

- **Sentiment analysis** is the interpretation and classification of emotions (positive, negative and neutral) within text data using text analysis techniques.
- **Topic modeling** is a method based on statistical algorithms to help uncover hidden topics from large collections of documents.
- **Text categorization** sorts texts into specific taxonomies following it being trained by humans.
- **Text clustering** is a technique used to group text or documents on similarities in content.
- **Information extraction** is used to automatically find meaningful information in unstructured text.
- **Named entity resolution** is a method that extracts the names of people, places, organizations, and more and classifies them into predefined labels and links the named entities to a specific logic.
- **Relationship extraction** is a capability that helps establish semantic relations between entities.

Key applications of NLP in Finance

- Legal and compliance – ex: Contract Intelligence (COIN) by JP Morgan
- Sentiment analysis – earnings call, news analytics, insider transactions. Ex: S&P sentiment analysis
- Deutsche Bank ESG analysis
- Speech recognition – speech to text/text to speech, voice bots/chat bots

- Reconciliation and dispute resolution – detecting failed trades by BNY
- Content enrichment – retrieval, trends, relationship between entities, due diligence screening

4-Step NLP Development Framework

An extensive ecosystem of technologies and best practices have been developed for NLP.

To leverage the ecosystem efficiently, we provide a 4-step framework for NLP development as displayed in the following figure:

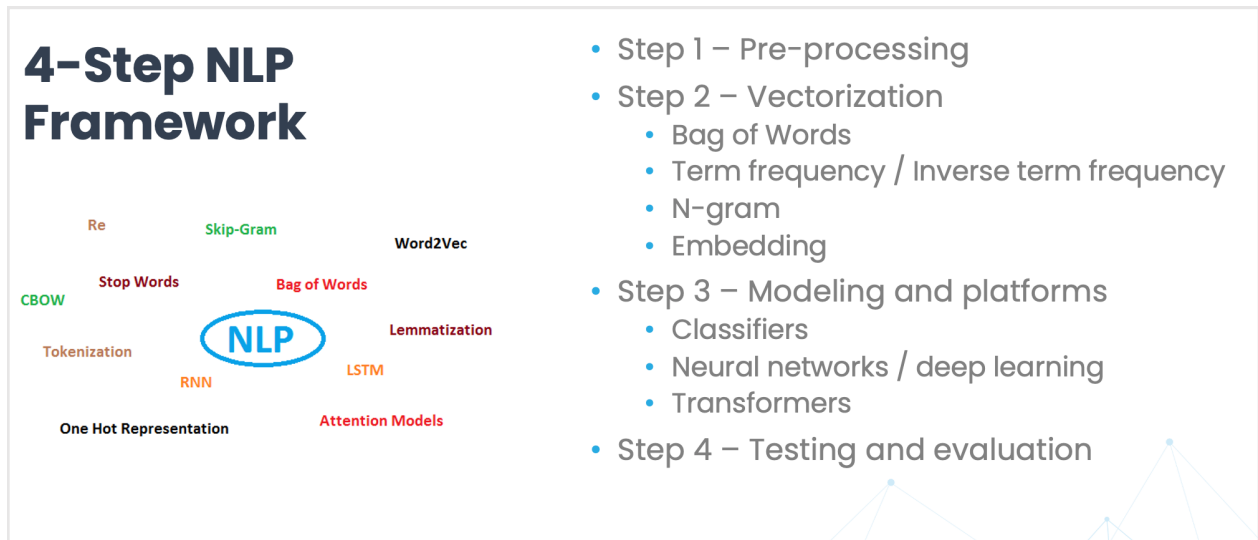


Figure 4 – The 4-Steps NLP framework

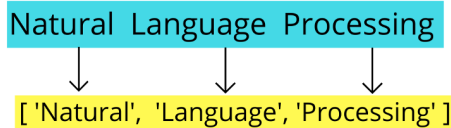
We will explain each one of these steps in more detail next.

Step 1: Pre-processing of Text Data

Text data is considered unstructured and meant for human consumption and not for computers. Words vary in length and order, and documents vary in word count. Sentences may contain grammar issues, random punctuation, and synonyms. Terminology and abbreviations in one industry may not apply to others. As a result, the context of the text data must be considered (text has linguistic structure). As a data type, text is considered “dirty”. Therefore, vast amount of text data needs to be converted into a form useable by computers. This process of Making these changes to our text before turning them into word sequences is called pre-processing. Cleaning and preparing text are called pre-processing. There are best practices steps in pre-processing of text data shown in figure below.

Step 1: Pre-Processing of Text Data

Tokenization



- Tokenization
- Remove punctuation/stop words
- Stemming
- Lemmatization
- Correct spelling mistakes
- Recognize abbreviations
- Remove rare words

Figure 5 – Pre-processing steps of text data

A simple approach is to assume that the smallest unit of information in a text is the word (as opposed to the character). To start the pre-processing step, we define the term Tokenization which is the process of breaking the text into pieces. Therefore, we will be representing our texts as word sequences. For instance, if the text is: This is a cat. Then, the word sequence is [this, is, a, cat]. In this example, we removed the punctuation and made each word lowercase because we assume that punctuation and letter case don't influence the meaning of words. In fact, we want to avoid making distinctions between similar words such as 'This' and 'this' or 'Cat' and 'cat'.

Moreover, real life text is often “dirty” since text is usually automatically scraped from the web with HTML code getting mixed with the actual text. To clean up these texts, we drop the HTML code words in our word sequences. For example, [`<div>`This is not a sentence], is converted to: [this, is, not, a, sentence].

Pre-Processing of Text Data



- The smallest unit of information in a text is the word
- Representing our texts as **word sequences**:
 - **Text:** This is a cat. --> **Word Sequence:** [this, is, a, cat]
- Real life text is often “dirty.” HTML code can mix up with the actual text. For example:
 - `<div>`This is not a sentence. --> [this, is, not, a, sentence]
- Python pre-processing functions take a block of text and outputs the cleaned version

Figure 6 – Pre-processing of Text Data

Despite being very simple, the pre-processing techniques work very well in practice. Depending on the kind of text you may encounter, it may be relevant to include more complex pre-processing steps.

Pre-processing extent depends on the follow-on vectorization methods. For example, for the term frequency analysis, the following steps are typically applied for pre-processing:

- Normalization by making every term as lowercase
- Remove punctuation/stop words (called common words)
- Stemming by removing suffixes (e.g., plural words)
- Lemmatization which is producing one term for all variations (e.g., tests, tested, testing to just test or iPhone and iphone)
- Correct spelling mistakes
- Recognize abbreviations
- Remove rare words
- Removes words like and, the, of, and on
- Numbers may also be discarded, but it depends on the context.

Step 2: Vectorization

Now that we have a way to extract information from text in the form of word sequences, we need a way to transform these word sequences into numerical features, this is **vectorization**.

The simplest text vectorization technique is **Bag OF Words (BOW)**. BOW starts with a list of words called the **Vocabulary** which is the output of pre-processing of raw text. Then, given an input text, using the vocabulary, BOW creates the output – a numerical vector which is simply the vector of word counts for each word of the vocabulary.

Step 2: Vectorization

- **Vectorization** – Methods to transform word sequences into numerical features
- **Bag Of Words (BOW)** – is the simplest text vectorization technique
- **Vocabulary** – BOW starts with a list of [raw] words. After pre-processing, we create the vocabulary which contain cleaned sequence of all the words that occur in the text
- **Numerical representation** – Given an input vocabulary, BOW, outputs a numerical vector which is the vector of word counts for each word of the vocabulary.



	1	0	0	0	...	0	0
Index:	0	1	2	3	...	99998	99999
	0	1	0	0	...	0	0
Index:	0	1	2	3	...	99998	99999

Figure 7 – Vectorization of text data

To see how the BOW vectorization works, we look at an example with the raw text as:

- ["This is a good cat", "This is a bad day"]

Applying the pre-processing method we discussed in the earlier section, we get the vocabulary as:

- [this, cat, day, is, good, a, bad]

And the numerical representation of new text comes out as:

- "This day is a good day" --> [1, 0, 2, 1, 1, 1, 0]

As we can see, the values for “cat” and “bad” are 0 because these words don’t appear in the vocabulary. This is demonstrated in the figure below.

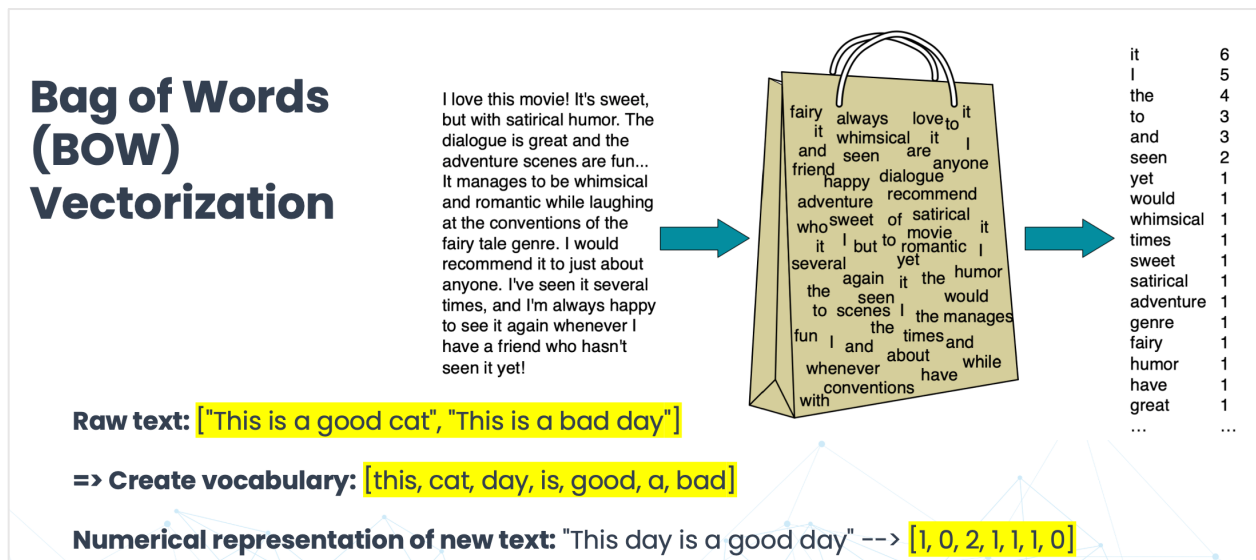


Figure 8 – Vectorization with the Bag of Words

Using BOW is making the assumption that the more a word appears in a text, the more it is representative of its meaning. Therefore, we assume that given a set of positive and negative text, a good classifier will be able to detect patterns in word distributions and learn to predict the sentiment of a text based on which words occur and how many times they do.

To use BOW vectorization in Python, we can rely on CountVectorizer from the scikit-learn library. In addition to performing vectorization, it will also allow us to remove stop words (i.e., very common words that don’t have a lot of meaning, like this, that, or the). Moreover, we can use our own custom pre-processing function from earlier to automatically clean the text before it’s vectorized.

The BOW method assumes that:

- Every document is a collection of words
- Each individual word is equally important
- Ignores grammar, sentence structure, word order, and punctuation

Where the benefits are:

- The process counts the frequency of tokens
- The implementation is very easy
- The classification and feature extraction applications can be based on this technique

However, there are shortcoming in this method:

- The tokens increase in the bag as the length of the data increases
- The sparsity in the matrix will also increase as the size of the input data increases. The number of zeros in the sparsity matrix is more than non-zero numbers
- There is no relationship/semantic connection with each other because the text is split into independent words

Summary for BOW

- BOW assumes:
 - every document is a collection of words
 - each individual word is equally important
 - Ignores grammar, sentence structure, word order, and punctuation
- Benefits:
 - The process counts the frequency of tokens
 - The implementation is very easy
 - The classification and feature extraction applications can be based on this technique
- Limitations:
 - The tokens increase in the bag as the length of the data increases
 - The sparsity in the matrix will also increase as the size of the input data increases. The number of zeros in the sparsity matrix is more than non-zero numbers
 - There is no relationship/semantic connection with each other because the text is split into independent words

Figure 9 – Benefits and Challenges for the BOW Method

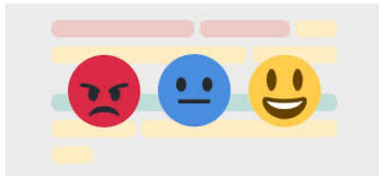
Stock Sentiment Analysis with BOW:

Sentiment analysis aims to estimate the **sentiment polarity** of a body of text based solely on its content. The sentiment polarity of text can be defined as a value that says whether the expressed opinion is **positive** ($polarity=1$), **negative** ($polarity=0$), or neutral. In this analysis, we will assume that texts are either positive or negative, but that they can't be neutral. Under this assumption, sentiment analysis can be expressed as the following classification problem:

We need to transform the main feature — i.e., a succession of words, spaces, punctuation and sometimes other things like emojis — into some numerical features that can be used in a learning algorithm. To achieve this, we will follow two basic steps:

- A **pre-processing** step to make the texts cleaner and easier to process
- And a BOW **vectorization** step to transform these texts into numerical vectors

Sentiment Analysis



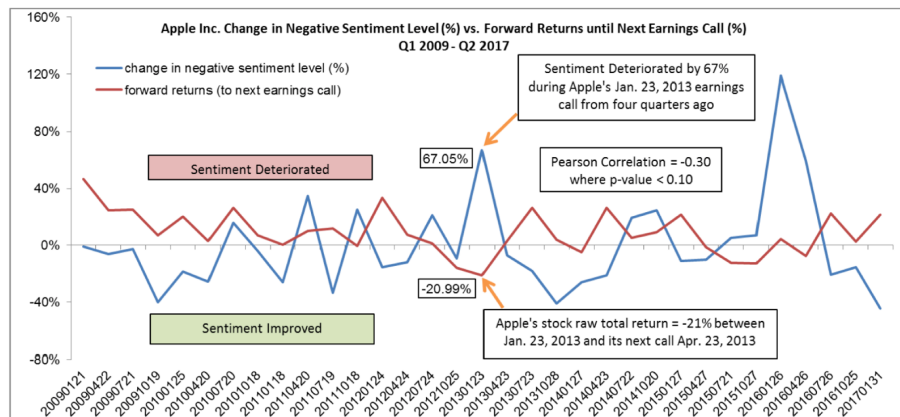
- Problem: Estimate the **sentiment** of a body of text with **Positive** opinion, **negative**, or **neutral**
- Method: Transform a succession of words, spaces, punctuation and things like emojis – into numerical features that can be used in a learning algorithm. To achieve this, we will follow two basic steps:
 - A **pre-processing** step to make the texts cleaner and easier to process
 - And a **vectorization** step to transform these texts into numerical vectors.

Figure 10 – Sentiment Analysis Methods

In this use case, by S&P 500, we capture the historical relationship of Apple Inc.’s sentiment level changes from its earnings calls and its forward returns until the next call (Figure below). The changes in sentiment are defined as quarter-over-quarter (QoQ) changes from four quarters ago (to account for seasonality). The sentiment of each of Apple’s earnings calls is defined by the proportion of negative words in its earnings call transcript where the classification of both the negative and the master word list is based on the Loughran and McDonald (2011) financial dictionary. Because sentiment in this use case is measured with negative words, positive (negative) changes reflect sentiment deterioration (improvement). Apple’s forward returns until its future calls have been shifted back a quarter such that its sentiment changes and its forward returns are aligned vertically in the Exhibit. One promising observation is that the Pearson correlation is about -0.30 since Q1 2009, which suggests that Apple’s forward returns historically go down when its sentiment deteriorates.

Stock Sentiment Analysis

BOW Method



Note: Sentiment is defined as the proportion of negative words in an earnings call using [Loughran and McDonald \(2011\)](#). Sentiment changes are measured quarter-over-quarter from four quarters ago. Source: S&P Global Market Intelligence Quantamental Research. Data as of 08/08/2017.

Figure 11 – Stock Sentiment Analysis

Vectorization with Term Frequency (TF), Inverse Document Frequency (IDF) and TF-IDF:

Another vectorization method is IF-IDF as following:

Putting aside anything fine-tuning related, there are some changes we can make to immediately improve the current BOW model. The first thing we can do is improve the vectorization step. In fact, there are some biases attached with only looking at how many times a word occurs in a text. In particular, the longer the text, the higher its features (word counts) will be.

To fix this issue, we can use Term Frequency (TF) instead of word counts and divide the number of occurrences by the sequence length. We can also downscale these frequencies so that words that occur all the time (e.g., topic-related or stop words) have lower values. This downscaling factor is called Inverse Document Frequency (IDF) and is equal to the logarithm of the inverse word document frequency. Put together, these new features are called TF-IDF features.

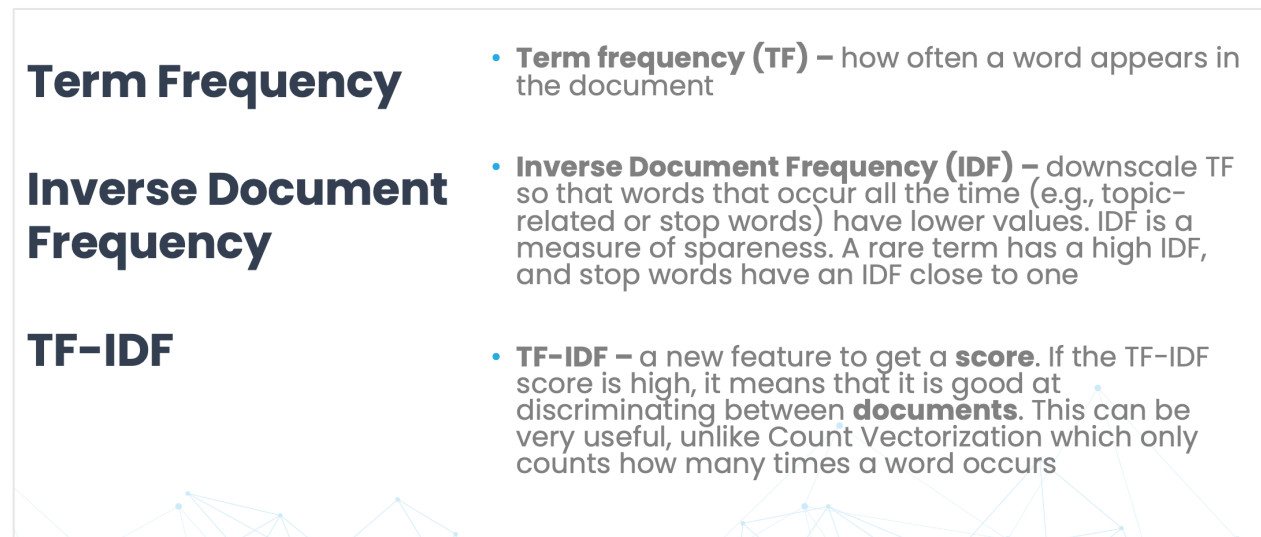


Figure 12 – TF-IDF Analysis

There are two methods in TF-IDF in the sklearn library: TF-IDF Vectorizer and TF-IDF Transformer.

- The TF-IDF Vectorizer method, takes the raw data as input and does further process.
- The TF-IDF Transformer, takes the output of the bag of words and does the further process.

The TF-IDF implementation tries to get information from the uncommon words.

So, in summary, for computing TF-IDF features, we can train a new Linear SVM on TF-IDF features simply by replacing the CountVectorizer with a TfidfVectorizer. This results in an improved accuracy over using BOW features.

Use Case with TF-IDF:

How can a search engine find the best document given certain search words?

To accomplish this task, we measure TF and IDF for the search word and the documents that might be chosen. We follow the steps below:

- Compute the number of times the word appears in the document divided by number of words in the document
- Compute Inverse document frequency (IDF) by taking the logarithm of number of documents divided by number of documents containing the word
- TF-IDF is the product of the two measures above

Results – Relevance of the document = Sum of the TF-IDFs across the search words

Information Retrieval

TF-IDF Vectorization

- **How can a search engine find the best document given certain search words?**
- Compute **TF** as number of times the word appears in the document divided by number of words in the document
- Compute **IDF** by taking the logarithm of number of documents divided by number of documents containing the word
- Compute **TF-IDF** as the product of the two measures above
- **Relevance of the document = Sum of the TF-IDFs across the search words**

Figure 13 – TF-IDF for Information retrieval

Vectorization with N-gram Method:

The second thing we can do to further improve the TF-IDF model is to provide it with more context. In fact, considering every word independently can lead to some errors. For instance, if the word ‘good’ occurs in a text, we will naturally tend to say that this text is positive, even if the actual expression that occurs is actually ‘not good’. These mistakes can be easily avoided with the introduction of N-grams. An N-gram is a set of N successive words (e.g., very good [2-gram] and not good at all [4-gram]). Using N-grams, we produce richer word sequences. For example, with N=2: This is a cat. becomes [this, is, a, cat, (this, is), (is, a), (a, cat)]. In practice, including N-grams in our TF-IDF vectorizer is as simple as providing an additional parameter n-gram range (=1, N). Generally speaking, the use of bi-grams improves performance, as we provide more context to the model, while higher-order N-grams have less obvious effects.

N-gram Vectorization

- N-grams consider sequences of adjacent words (i.e., order is important)
- Beneficial when a phrase is more significant than individual words, for example: **exceed analyst expectations** may be more meaningful than the separate words: **exceed, analyst, and expectations**
- Disadvantage to this approach is that it increases feature vector size

For example, with N=2: This is a cat. --> [this, is, a, cat, (this, is), (is, a), (a, cat)]

Figure 14 – N-gram Vectorization

Challenges with TF-IDF and N-gram methods:

- Pros of TF-IDF: It slightly overcomes the semantic information between tokens.
- Issue: TF-IDF: This method gives chance for the model to overfit. Not so much a semantic relationship between the tokens.
- Issue: N-gram method increases vector size

What Next?

- Features resulting from **count-based vectorization** methods like BoW and TF-IDF have some disadvantages. For instance:
- They don't account for word position and context (despite using N-grams, which is only a quick fix).
- TF-IDF word vectors are usually very high dimensional (>1M features if using bi-grams).
- **Need more comprehensive notion of semantics and context while reducing vector dimension**

Figure 15 – Challenges for Vectorization with TF-IDF and N-grams

Next Generation Vectorization – Word Embedding:

Word embedding is a process to convert words/tokens into numbers (i.e., vectors) for the purpose of analysis by natural language models. To perform word embedding, first, we create words or tokens from a corpus of text as we mentioned in the earlier sections. In the typical NLP project, we can easily produce a very large vocabulary of words, let's say around 100,000. We will then assign a number to each word in the vocabulary. The first word in our vocabulary will

be number 0. The second word will be number 1, and so on up to number 99,998. Then we represent every word as a vector of length 100,000, where every single item is a zero except one corresponding to the index of the number that the word is associated with. This is called the “**one-hot**” encoding for words. Consider an example with only three words in our vocabulary: ‘apple’, ‘banana’ and ‘king’. The one hot encoding vector representations of these words would be the following. If we then plotted these word vectors in a 3-dimensional space, we would get a representation like the one shown in the following figure, where each axis represents one of the dimensions that we have, and the icons represent where the end of each word vector would be.

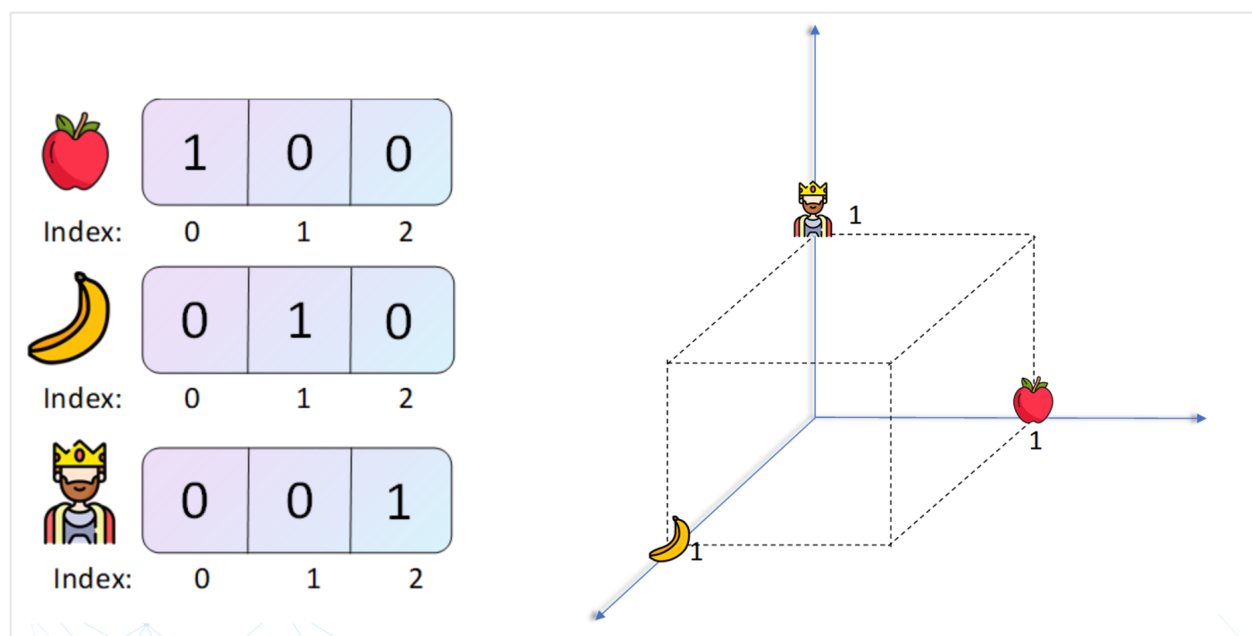


Figure 16 – One-hot Encoding

As we can see, the distance from any vector (position of the icons) to all the other ones is the same: two size 1 steps in different directions. This would be the same if we expanded the problem to 100,000 dimensions, taking more steps but maintaining the same distance between all the word vectors. However, the one-hot encoding present several challenges:

- One-hot encodings are very inefficient. They are huge empty vectors with only one item having a value different than zero. They are very sparse and can greatly slow down our calculations.
- Ideally, we would want vectors for words that have similar meanings or represent similar items to be close together [in a meaning dimensional space], and far away from those that have completely different meanings.
- The one-hot encoding doesn't account for the context or meaning of the words, all the words vectors have the same distance between them and are highly inefficient.

Word embeddings solve these problems by representing each word in the vocabulary by a fairly small (150, 300, 500 dimensional) fixed size vector, called an embedding, which is learned during the training. These vectors are created in a manner so that words that appear in similar contexts or have similar meaning are close together, and they are not sparse vectors like the ones derived from one-hot embeddings.

Word Embedding

- **Embedding** – type of **word representation** that allows words with similar meaning to have a similar representation (i.e., similar vectors)
- Word embedding include implicit relationships between words that can translate to **contextual** information
- Word embedding method requires **large training data**
- **Embedding method + deep learning** are the new paradigm in modern NLP

Figure 17 – Word Embedding

If we had a 2-dimensional word embedding representation of our 4 words vocabulary [king, queen, apple, banana] and plotted it on a 2D grid, it would look something like the following figure.

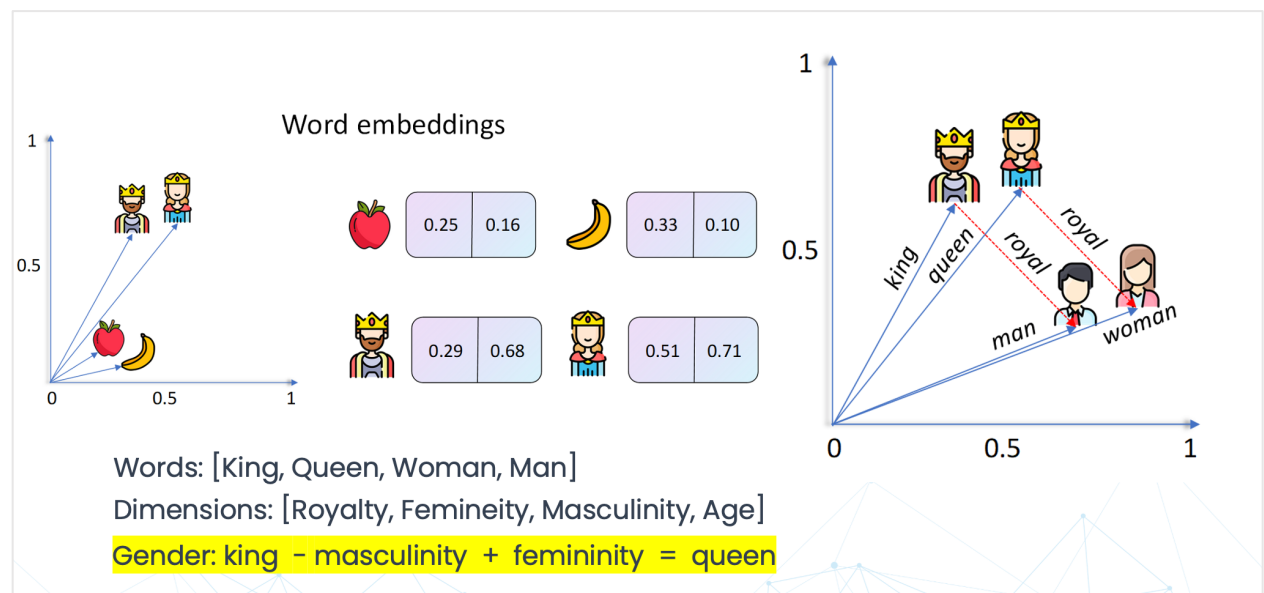


Figure 18 – Word Embedding with Relationships

As we can see, the word embedding representations of the words ‘*apple*’ and ‘*banana*’ are closer together than to the words like ‘*king*’ and ‘*queen*’. So, words with similar meanings are close together when we use word embeddings. The embedding method also allows us to operate with word embeddings, using representations of words to go from a known word to another one. If we subtract the word embedding of the word ‘*royal*’ from the embedding of the word ‘*king*’ we arrive somewhere near the embedding of the word ‘*man*’. In a similar manner, if we subtract the embedding of ‘*royal*’ from the embedding of queen, we arrive somewhere near the embedding of the word ‘*woman*’, which means the algorithm has discovered the notion of ‘Gender’!

Lastly, as we can see in the word embedding vectors, they usually have a smaller size (2 in our example, but most times they have 150, 200, 300, or 500 dimensions) and are not sparse, making calculations with them much more efficient than with one-hot vectors.

How are Word Embeddings Built?

Word embeddings are built by learning from [training] data. Artificial neural networks (deep learning) can learn word embeddings. The main objective of this learning is to build a matrix E , that can translate a one-hot vector representing a word, to a fixed sized vector that is the embedding of such word. Let's see a very high-level example of one way this could be done.

Consider the sentence “I love drinking apple smoothies”. If we remove the word ‘apple’ we are left with the following incomplete sentence: ‘I love drinking __ smoothies’. Naturally, you would guess words like ‘banana’, ‘strawberry’, or ‘apple’, which all have a similar meaning, and usually appear in similar contexts. One of the main ways to learn word embeddings, is by a very similar process performed by algorithms – by guessing missing words in a huge corpus of text sentences.

Key takeaway: An embedding matrix E (the matrix that translates a one hot embedding into a word embedding vector) is calculated by training an Artificial Neural Network to predict missing words, in a similar manner to how the weights and biases of the network are calculated.

In practice, you can avoid training your own word embeddings, as there are publicly available word embeddings built from various corpuses (like Wikipedia or Twitter GloVe Word embeddings).

What are the most popular word embeddings?

The two most used Word embedding algorithms are Word2Vec and GloVe.


- Word2Vec is a group of related models that produce word embeddings by using two-layer, shallow artificial neural networks that try to predict words using their context (Continuous bag of words — CBOW), or to predict the context using just one word (Skip-gram model).
- GloVe, short for Global Vectors, the GloVe algorithm calculates word embeddings by using a co-occurrence matrix in between words. This matrix is built by reading through a huge corpus of sentences and creating a column and a row for every unique word it finds. For every word, it registers how many times it appears in the same sentence with other

words using a specific window size, so it also has a measure of how close together two words are in a sentence.

Step 3: Language Modeling

In the last few years, the NLP field has greatly benefited from the scale roll out of Deep Neural Networks (DNNs), due to their high performance with less need of engineered features. Although, several ML models have been used in the last decade, DNNs have been the driver of a paradigm shift in language modeling.

Step 3: Language Modeling



- AI/ML models used with NLP: using
 - Naïve Bayes
 - SVM
 - Logistic regression
 - Decision trees
 - **Deep Neural Learning**
- Labeled data divided into training set, validation set, and test set
- The number of features (i.e., number of words) is large

Figure 19 – Language Modeling for NLP

Up until a few years ago, when dealing with NLP, what most people and organizations were doing was using recurrent neural networks, or RNN, a branch of deep neural learning. Recurrent Neural Networks (RNN) are a type of neural network where the output from previous step is fed as input to the current step. In traditional neural networks, all the inputs and outputs are independent of each other, but in cases like when it is required to predict the next word of a sentence, the previous words are required and hence there is a need to remember the previous words. Therefore, the approach with RNN seemed like a more “natural” approach due to the inherent sequential structure of text, a.k.a. the fact that each word comes after another. While still widely used in business applications, the technique has revealed challenges because RNNs are inherently sequential, it is very hard to parallelize their training or their inference. This, along with their high memory bandwidth usage (as such, they are memory-bandwidth-bound, rather than computation-bound), makes them hard to scale.

Use Case: Deutsche Bank ESG Evaluation

Deutsche Bank decided to develop alternative ways to analyze sustainability reports, in order to gauge if companies are truly aligning their business with sustainable practices. They decided to investigate independently whether the commitments firms make to reducing carbon emissions were associated with actual achieved sustainability performance.

The bank researchers analyzed carbon-related discussions within the reports using topic modeling and identified five different topics, along with the top keywords associated with each topic. Companies were then ranked based on their focus on the mitigation and adaptation topics. The NLP system also scanned for numbers and quantitative words (like ‘first’ and ‘half’), and for active (vs. passive) language.

As a result, the bank found that companies using highly active and numeric language have, on average, a 74% chance of reducing their future emissions. Also, companies that frequently discuss mitigating or adapting to climate change have a 65% higher chance of achieving reductions.

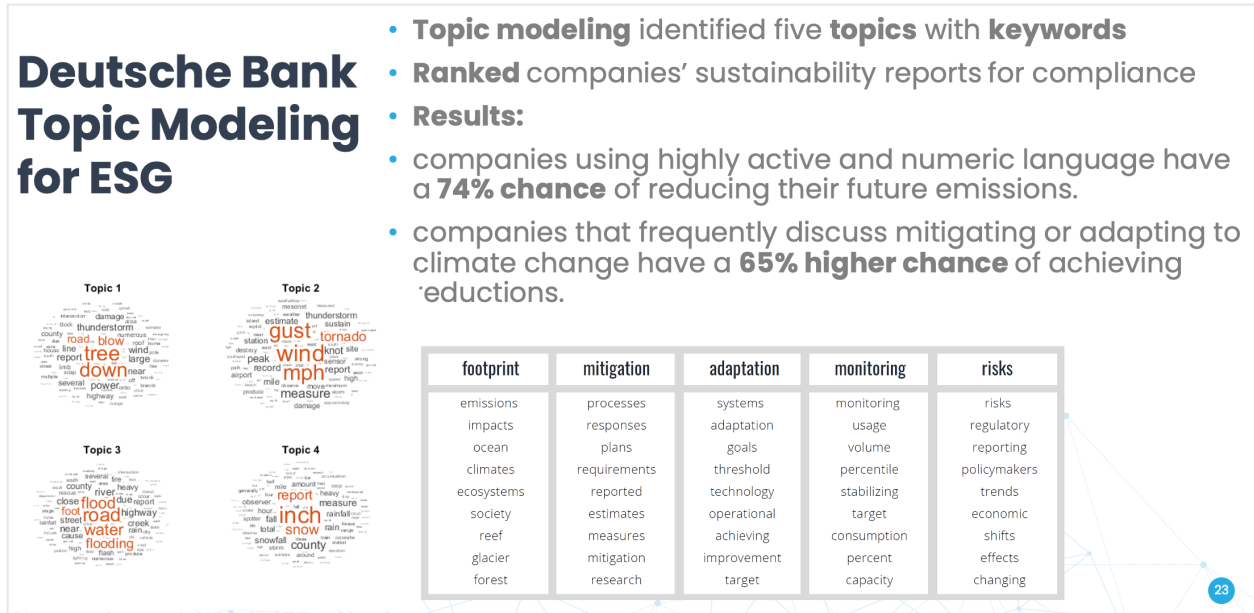
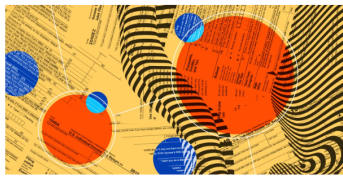


Figure 20 – Topic Modeling with NLP

Use Case: JP Morgan Contract Intelligence COIN

Feature selection speeds up contract reviews – JP Morgan Chase implemented a program called COIN, which stands for Contract Intelligence. The NLP system identifies and categorizes repeated clauses. It does so by classifying clauses according to about 150 different “attributes” (Also known as features). COIN analyses contract documents to find words or phrases relevant to these attributes. Based on these attributes, the system extracts from the contract the relevant sections warranting human review. If the system fails to analyze a contract, it directs it to human reviewers, for them to manually search the document.

The bank reported that the solution saves lawyers and loan officers work 360,000 hours annually.



JP Morgan Contract Intelligence (COIN)

- **Problem: Contract reviews**
- JP Morgan Chase set out to improve the process of reviewing commercial-loan agreements
- The bank implemented a program called COIN, which stands for Contract Intelligence.
- **Topic modeling and text classification** using 150 different “attributes”/“dimensions” to find words or phrases relevant to these attributes.
- Based on these, the system extracts from the contract the relevant sections warranting human review.
- **Result: The bank reported that the solution saves lawyers and loan officers work 360,000 hours annually.**

Figure 21 – Topic and Text Modeling with NLP

The Next Generation NLP Systems:

More recent breakthroughs in NLP architecture, called **Transformer** models process words in relation to all the other words in a sentence, rather than one-by-one in order. Transformers have provided a paradigm shift in language modeling. In contrast to RNNs, the main advantage of the Transformer models is that they are not sequential, which means they can be parallelized and scaled much more easily. But, in order to understand Transformers, we will need to dive into its core technique: the novel paradigm called **Attention**.

The Attention technique allows us to get rid of the inherent sequential structure of RNNs, which hinders the parallelization of such models. When translating a sentence or transcribing an audio recording, a human agent would pay special attention to the word they are presently translating or transcribing. Neural networks can achieve this same behavior using Attention by focusing on part or a subset of the information. The Attention technique (a revolutionary idea in sequence-to-sequence systems such as translation models), has given rise to the new language models based on the transformer architecture.



The transformer models provide several new capabilities that have caused an inflection in the language models such as:

- The Transformer learns embeddings etc., in such a way that words that are relevant to one another are more aligned.
- Transformer models are not sequential, which means they can be parallelized, and that bigger and bigger models can be trained by parallelizing the training
- Transformers are massive pre-trained models, which can be then fine-tuned to specific language-related tasks. Transfer learning allows to re-use knowledge from previously built models, which can give a boost in performance, while demanding much less labelled training data
- Another major new development in NLP is that you don’t need to have labeled data anymore. Newer language models are typically trained on very large amounts of publicly available data, i.e., unlabeled text from the web, for instance to predict the next word in a sentence based on previous words or to predict masked parts of the sentence. This is

called **self-supervised learning**, and it's in its own a very interesting and promising technique that will further grow the use of NLP and spawn new applications

Since the introduction of Transformer architecture, numerous projects including Google's BERT and OpenAI's GPT series have built on this foundation and published performance results that handily beat existing state-of-the-art benchmarks. BERT released in 2018 by the Google research team, has been applied it to improving the query understanding capabilities of Google Search. By applying BERT models to both ranking and featured snippets in Search, BERT can help Search better understand one in 10 searches in the U.S. in English.

The Next Gen NLP




- **Transformers** models process words in relation to all the other words in a sentence
 - Open AI GPT-3, pretrained with 175B parameters!
 - Google BERT, Language model behind search and translation
- Applications:
 - Voice assistants and chatbots in customer services
 - Information retrieval and sentiment analysis of corporate documents and news feeds
 - Compliance, regulations and contracts

Figure 22 – Transformers and Applications with NLP

Key Takeaways – Advances in NLP are driving the growth in leveraging unstructured data. Transformer architectures are expanding fast due to the new capability for self-supervised learning that enable pre-training of NLP models on unlabeled data. As a result, many high impact applications such as – sentiment analysis, tracking relationships, legal and compliance, speech recognition and advanced chatbots, information retrieval, and ESG – will see adoption at scale.

Summary



- Textual data is growing exponentially with 80% of all created data as text.
- As a result, NLP is a fast-growing field in AI and the basis for many large-scale applications
- Adoption in financial services is steady with key applications such as:
 - Sentiment analysis
 - Tracking relationships between entities
 - Legal and compliance
 - Speech recognition and advanced chatbots
 - Information retrieval
 - ESG

Figure 23 – Summary

APPENDIX: NLP Frameworks

Below, we highlight some of the more popular NLP platforms:

- [PyTorch](#) is [an open-source machine and deep learning library](#). It's often used for NLP and integrates with Facebook AI's newest RoBERTa project. It's fast and flexible, supports GPU computation, and operates Recurring Neural Networks for things like classification, tagging, and text generation.
- [SpaCy](#) is fast and agile for cutting edge NLP by making it practical and accessible. It works with other well-known libraries like Gensim and Scikit Learn. SpaCy is optimized for performance and allows developers a more natural path to more advanced NLP tasks like named entity recognition.
- **Facebook AI XLM/mBERT**, is [Facebook's brand new multilingual language model](#) that brings new training data sets to the table.
- XLM-R achieved the best results to date on four cross-lingual benchmarks becoming the first multilingual model to outperform traditional monolingual baselines. It performs particularly well for low-resource languages like Urdu or Swahili.
- <https://odsc.com/boston/>**Baidu ERNIE**, Otherwise known as "[Enhanced Representation through kNowledge IntEgration](#)," ERNIE is a state of the art NLU framework offering pre-trained models that outperformed BERT in both English and Chinese. It includes continual pretraining and is word aware, structure-aware, and semantic aware.
- **TensorFlow**, [TensorFlow](#) remains [one of the most popular frameworks](#) for machine and deep learning, but you can translate that power to NLP tasks. Its most famous application is the Google Translate.
- **Stanford CoreNLP**, [Stanford's generalized tool](#) can perform sentiment analysis, bootstrapped pattern learning, and named entity recognition across 53 languages with these neural models in addition to a whole suite of other common NLP tasks.
- [Keras](#) runs on CPU or GPU, making it suitable for high level, deep learning. [Keras](#) focuses on rapid iterations, enabling users to execute experiments efficiently. Plus, you've got all the usual NLP functions, including parsing, machine translation, and classification.
- [Chainer](#) belongs to the Python ecosystem and is [a standalone framework](#) for deep learning. It comes in handy with RNNLM (recurrent neural network language models and modeling sequential data — think sentences in natural language).
- [Gensim](#) was explicitly designed for sentiment analysis and unsupervised topic modeling. It's a workhorse with NLP, working with raw, unstructured data like a champ. The Gensim [Word2Vec model](#) helps with things like word embedding or processing documents. If you're working with its specific use cases, it's a game-changer.
- [Scikit-Learn](#) is an excellent framework for implementing things like regression and classification data. People often use it for classifying news publications, for example, or even working with tweets. It's beginner-friendly and well documented, allowing those just starting in the field to get started quickly