**Machine Learning**:

In general, a learning problem considers a set of n [samples](https://en.wikipedia.org/wiki/Sample_(statistics)) of data and then tries to predict properties of unknown data. If each sample is more than a single number and, for instance, a multi-dimensional entry (aka [multivariate](https://en.wikipedia.org/wiki/Multivariate_random_variable) data), it is said to have several attributes or features.

We can separate learning problems in categories:

* [**supervised learning**](https://en.wikipedia.org/wiki/Supervised_learning), in which the data comes with additional attributes that we want to predict This problem can be either:
* [**classification**](https://en.wikipedia.org/wiki/Classification_in_machine_learning) : samples belong to two or more classes and we want to learn from already labeled data how to predict the class of unlabeled data. An example of classification problem would be the handwritten digit recognition example, in which the aim is to assign each input vector to one of a finite number of discrete categories. Another way to think of classification is as a discrete (as opposed to continuous) form of supervised learning where one has a limited number of categories and for each of the n samples provided, one is to try to label them with the correct category or class.
* [**regression**](https://en.wikipedia.org/wiki/Regression_analysis) : if the desired output consists of one or more continuous variables, then the task is called regression. An example of a regression problem would be the prediction of the length of a salmon as a function of its age and weight.
* [**unsupervised learning**](https://en.wikipedia.org/wiki/Unsupervised_learning), in which the training data consists of a set of input vectors x without any corresponding target values. The goal in such problems may be to discover groups of similar examples within the data, where it is called [clustering](https://en.wikipedia.org/wiki/Cluster_analysis), or to determine the distribution of data within the input space, known as [density estimation](https://en.wikipedia.org/wiki/Density_estimation), or to project the data from a high-dimensional space down to two or three dimensions for the purpose of *visualization* .

**Training set and testing set**

Machine learning is about learning some properties of a data set and applying them to new data. This is why a common practice in machine learning to evaluate an algorithm is to split the data at hand into two sets, one that we call the training set on which we learn data properties and one that we call the testing set on which we test these properties.

**Learning and Predicting:**

In the case of the digits dataset, the task is to predict, given an image, which digit it represents. We are given samples of each of the 10 possible classes (the digits zero through nine) on which we *fit* an [estimator](https://en.wikipedia.org/wiki/Estimator) to be able to *predict* the classes to which unseen samples belong.

In scikit-learn, an estimator for classification is a Python object that implements the methods fit(X, y) and predict(T).

**Model Persistence:**

It is possible to save a model in the scikit by using Python’s built-in persistence model, namely [pickle](https://docs.python.org/2/library/pickle.html):

In the specific case of the scikit, it may be more interesting to use joblib’s replacement of pickle (joblib.dump & joblib.load), which is more efficient on big data, but can only pickle to the disk and not to a string. joblib.dump and joblib.load functions also accept file-like object instead of filenames.

Support vector machines (SVMs) are a set of supervised learning methods used for [classification](http://scikit-learn.org/stable/modules/svm.html#svm-classification), [regression](http://scikit-learn.org/stable/modules/svm.html#svm-regression) and [outliers detection](http://scikit-learn.org/stable/modules/svm.html#svm-outlier-detection).

The advantages of support vector machines are:

* Effective in high dimensional spaces.
* Still effective in cases where number of dimensions is greater than the number of samples.
* Uses a subset of training points in the decision function (called support vectors), so it is also memory efficient.
* Versatile: different [Kernel functions](http://scikit-learn.org/stable/modules/svm.html#svm-kernels) can be specified for the decision function. Common kernels are provided, but it is also possible to specify custom kernels.

The disadvantages of support vector machines include:

* If the number of features is much greater than the number of samples, avoid over-fitting in choosing [Kernel functions](http://scikit-learn.org/stable/modules/svm.html#svm-kernels) and regularization term is crucial.
* SVMs do not directly provide probability estimates, these are calculated using an expensive five-fold cross-validation (see [Scores and probabilities](http://scikit-learn.org/stable/modules/svm.html#scores-probabilities), below).

The support vector machines in scikit-learn support both dense (numpy.ndarray and convertible to that by numpy.asarray) and sparse (any scipy.sparse) sample vectors as input. However, to use an SVM to make predictions for sparse data, it must have been fit on such data. For optimal performance, use C-ordered numpy.ndarray (dense) orscipy.sparse.csr\_matrix (sparse) with dtype=float64.

C-Support Vector Classification.

The implementation is based on libsvm. The fit time complexity is more than quadratic with the number of samples which makes it hard to scale to dataset with more than a couple of 10000 samples.

The multiclass support is handled according to a one-vs-one scheme.

The [sklearn.feature\_extraction](http://scikit-learn.org/stable/modules/classes.html" \l "module-sklearn.feature_extraction" \o "sklearn.feature_extraction) module can be used to extract features in a format supported by machine learning algorithms from datasets consisting of formats such as text and image. the former consists in transforming arbitrary data, such as text or images, into numerical features usable for machine learning.

Text Feature Extraction:

The Bag of Words representation

Text Analysis is a major application field for machine learning algorithms. However the raw data, a sequence of symbols cannot be fed directly to the algorithms themselves as most of them expect numerical feature vectors with a fixed size rather than the raw text documents with variable length.

In order to address this, scikit-learn provides utilities for the most common ways to extract numerical features from text content, namely:

* tokenizing strings and giving an integer id for each possible token, for instance by using white-spaces and punctuation as token separators.
* counting the occurrences of tokens in each document.
* normalizing and weighting with diminishing importance tokens that occur in the majority of samples / documents.

In this scheme, features and samples are defined as follows:

* each individual token occurrence frequency (normalized or not) is treated as a feature.
* the vector of all the token frequencies for a given document is considered a multivariate sample.

A corpus of documents can thus be represented by a matrix with one row per document and one column per token (e.g. word) occurring in the corpus.

We call vectorization the general process of turning a collection of text documents into numerical feature vectors. This specific strategy (tokenization, counting and normalization) is called the Bag of Words or “Bag of n-grams” representation. Documents are described by word occurrences while completely ignoring the relative position information of the words in the document.

Count Vectorizer

Convert a collection of text documents to a matrix of token counts

This implementation produces a sparse representation of the counts using scipy.sparse.csr\_matrix.

If you do not provide an a-priori dictionary and you do not use an analyzer that does some kind of feature selection then the number of features will be equal to the vocabulary size found by analyzing the data. Here we do not have to write a custom code for counting words and representing those counts as a vector. Scikit's CountVectorizer does the job very efficiently. It also has a very convenient interface. The parameter min\_df determines how CountVectorizer treats words that are not used frequently (minimum document frequency). If it is set to an integer, all words occurring less than that value will be dropped. If it is a fraction, all words that occur less than that fraction of the overall dataset will be dropped. The parameter max\_df works in a similar manner. Once we vectorize the posts using feature vector functionality we'll have 2 simple vector. We can then simply calculate the Euclidean distance  between these two vector and calculate the nearest one to identify *similarities*. This is nothing but step towards clustering/classification of *similar* posts.

1. We nTokenizing the text. -- Vectorization and tokenizing
2. Throw away some less important words. -- stop word
3. Throwing away words that occur way too often to be of any help in detecting relevant posts. -- stemming
4. Throwing away words that occur so seldom that there is only a small chance that they occur in future posts.
5. Counting the remaining words.
6. Calculating TF-IDF values from the counts, considering the whole text corpus. -- calculate TF-IDF

eed to cautiously move with below steps towards bringing our raw text to a more meaningful [{*bag of words*}](http://en.wikipedia.org/wiki/Bag-of-words_model).

When we use feature extraction and vectorized the text then this feature values simply count occurrences of terms in a post. We silently assumed that higher values for a term also mean that the term is of greater importance to the given post. But what about, for instance, the word "subject", which naturally occurs in each and every single post? Alright, we could tell CountVectorizer to remove it as well by means of its max\_df parameter. We could, for instance, set it to 0.9 so that all words that occur in more than 90 percent of all posts would be always ignored. But what about words that appear in 89 percent of all posts? How low would we be willing to set max\_df? The problem is that however we set it, there will always be the  
problem that some terms are just more discriminative than others. This can only be solved by counting term frequencies for every post, and in addition, discounting those that appear in many posts. In other words, we want a high value for a given term in a given value if that term occurs often in that particular post and very rarely anywhere else. This is exactly what term frequency – inverse document frequency (TF-IDF)

So, continue to the previous code where we have imported CountVectorizer library to vectorize and tokenized the text and in below example we are going to compare "Big Data Hype" term with 2 different posts published about "Hype" of "Big Data". To do this we first need to vectorized the posts in question (new post) and then get the third post vectorized using the same method of scikit. Once we have vectors then we can calculate the distance of the new post. This code snippet ONLY covers vectorizing and tokenizing the text.

http://blog.christianperone.com/2011/09/machine-learning-text-feature-extraction-tf-idf-part-i/