# Label Propagation.

**Intro**

Despite the abundance of data all around us, the vast majority of it is unstructured and unlabeled. At the same time, many of our Machine Learning applications, such as classification models, require us to have target labels.

Unfortunately, we may not always have the resources to go through tens of thousands of observations and assign labels manually. But what if we did not have to do that? What if we could automatically label large amounts of data using just a tiny fraction of examples?

# What is Semi-Supervised Learning?

Typically, we would use data with a specific target variable (labeled data) to build supervised models (e.g., classification, regression). Alternatively, we would build unsupervised models (e.g., clustering, dimensionality reduction) when we do not have labeled data.

However, sometimes we may find ourselves in situations with a small amount of labeled data and a significant amount of unlabeled data. That’s where Semi-Supervised Learning can help since it incorporates elements from both supervised and unsupervised techniques.

## **Example**

Let’s consider an example. Assume you have 10,000 sentences with user comments, and you want to classify them into positive and negative. Unfortunately, you only have 50 sentences to which you have previously manually assigned a label (positive, negative).

Unless you want to spend many more hours labeling the rest of the data, your options are:

* **Build a supervised model**using 50 labeled examples — this may result in a poor-performing model due to the small number of samples available.
* **Build an unsupervised model** with unlabeled data to group examples into two clusters. However, the data may naturally want to form multiple smaller clusters instead, and forcing them into just two groups may not necessarily split them amongst the intended target (positive/negative).
* **Build a semi-supervised model**using all labeled and unlabeled data — this will use 50 examples to label the rest of the data and give you a much larger dataset to work with when building a supervised sentiment prediction model.

I’m sure you are now curious to find out how this works. So, let’s take a closer look at one specific algorithm called Label Propagation.

# An intuitive explanation of how Label Propagation works

Label Propagation is a relatively simple algorithm based on the assumption that closer data points have similar class labels. As a result, we can propagate these class labels through dense unlabeled data regions.

The algorithm follows an iterative approach, which we can describe as a collection of the following steps:

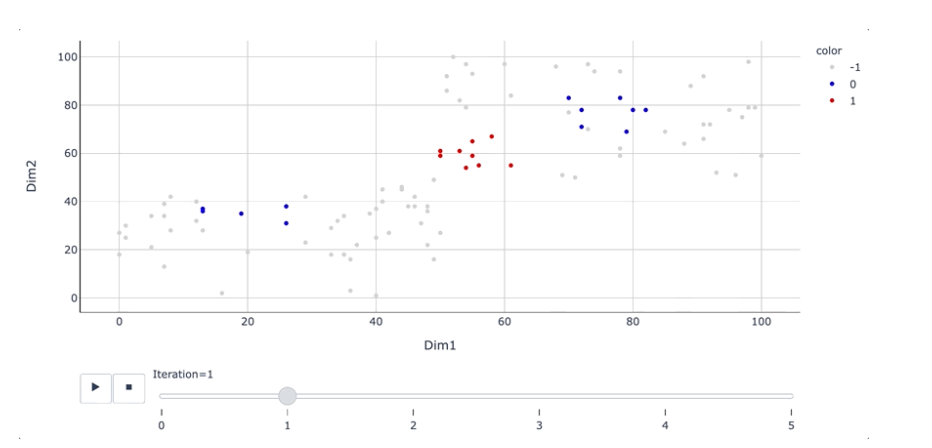
1 - **Create a connected graph**by drawing edges (links) between different nodes (data points). Note that creating afully connected graph on a large dataset may demand a high amount of resources from your machine. Hence, it is often beneficial to limit the number of neighbors that you want to join together

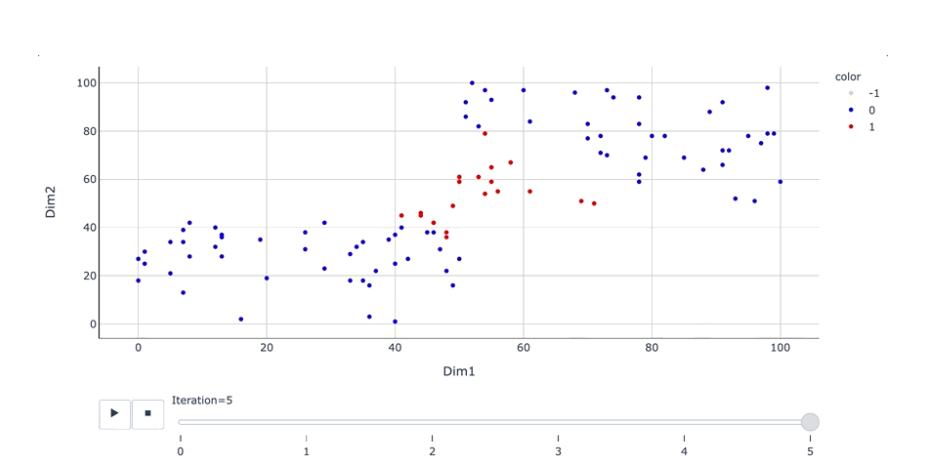
2 - **Determine the weights** for each edge, where edges for closer data points have larger weights (stronger connection), and edges for faraway points have smaller weights (weaker connection). Larger edge weights allow labels to travel through easier, increasing the probability of propagating the particular label.

3- **Perform a random walk**from each unlabeled point to find a probability distribution of reaching a labeled one. This random walk consists of many iterations and continues until convergence is reached, i.e., all paths have been explored, and probabilities no longer change.

Unlabeled points get their new labels assigned based on the probabilities found by the process above. **Note that original labeled points never change since their labels are clamped (fixed).**

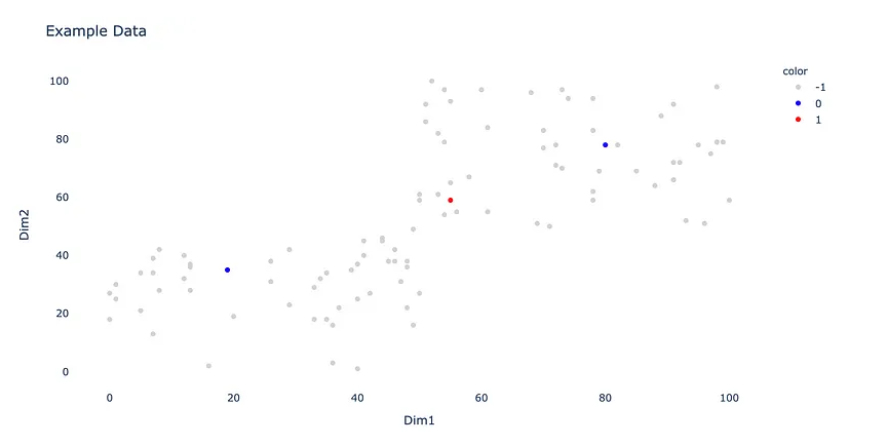
Check these two images to give you an intuitive view of how labels get propagated through the network.





I have designed the above example to show a scenario where a Semi-Supervised approach would have an advantage over using either a Supervised or Unsupervised one.

Note that we have three labeled samples available at the start (see below image). Based on this information, we can infer that red labels are likely to be centered in the middle, with blue ones around the outside (although it would always be beneficial to have more known labels to ensure our inference is correct):

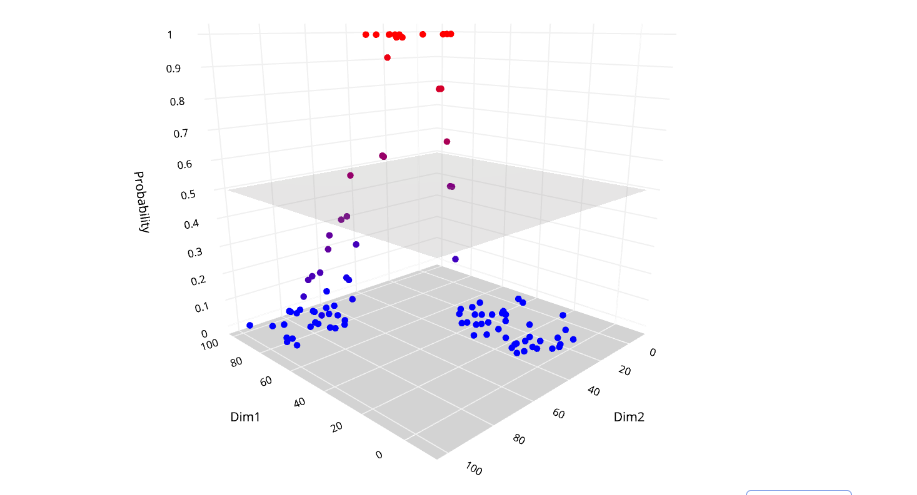


An unsupervised model would not do a great job either since there are no two clearly defined clusters that would separate red and blue points.

The Label Propagation algorithm is able to propagate labels outwards through the network, making the best use of the **entire data** (labeled and unlabeled).

**Another critical aspect of the Label Propagation algorithm is that we can view the corresponding probabilities in addition to hard labels after the algorithm has finished running.** Hence, we could manually adjust the threshold and re-label some points if we were not happy with the boundary determined by the algorithm.

See the below interactive 3D graph, which shows a probability of belonging to the red label (label 1) alongside the two dimensions (Dim 1, Dim 2) that we had in the picture and image above.



As you can see, hard labels are assigned based on the probability of belonging to a particular class, with 0.5 being the threshold. However, the model is less sure about the points located closer to the boundary. Hence, if we wished to do so, we could move the threshold up or down and reclassify marginal cases.

**Parameters:**

**Kernel *{‘knn’, ‘rbf’} or callable, default=’rbf’***

String identifier for kernel function to use or the kernel function itself. Only ‘rbf’ and ‘knn’ strings are valid inputs. The function passed should take two inputs, each of shape (n\_samples, n\_features), and return a (n\_samples, n\_samples) shaped weight matrix.

**Gamma *float, default=20***

Parameter for rbf kernel.

**n\_neighbors *int, default=7***

Parameter for knn kernel which need to be strictly positive.

**max\_iter *int, default=1000***

Change maximum number of iterations allowed.

**Tol *float, 1e-3***

Convergence tolerance: threshold to consider the system at steady state.

**n\_jobs *int, default=None***

The number of parallel jobs to run. None means 1 unless in a **[joblib.parallel\_backend](https://joblib.readthedocs.io/en/latest/generated/joblib.parallel_backend.html" \l "joblib.parallel_backend" \o "(in joblib v1.4.dev0))** context. -1 means using all processors. See [Glossary](https://scikit-learn.org/stable/glossary.html#term-n_jobs) for more details.