Introduction to Semi-Supervised Learning

Christian Bakhit, Jonathan Bakhit, Bryan Olivo, Rongrong Liu, Yiyang Han

Johns Hopkins University

110.445 Mathematical and Computational Foundations of Data Science

Sina Hazratpour

April 11th, 2023

Abstract

As Data Science has become more commonplace in our day to day lives, there has been an increased need to understand the models and algorithms that control the function of many of our favorite applications. Semi-supervised learning (SSL) is one such technique that is often applied in Natural Language Processing, Speech Recognition, Text classification, and more. By taking the best of both supervised and unsupervised learning types, it allows for generalization of data to larger datasets. In short, Semi supervised learning allows for models to be developed for use cases where the quantity of labeled data is much less than unlabeled data.

Theory

Semi-supervised learning

The goal of supervised learning is to construct a classifier or a regressor for unseen inputs using a set of data points consisting of their input features and their output label. On the other hand, unsupervised learning does not have a labeled output, so one must find the underlying structure from just the inputs. Thus, semi-supervised learning (SSL) being a combination of these two tasks uses both labeled and unlabeled inputs to construct its learning algorithms. There are several representative methods in semi-supervised learning. In this section, we will mainly discuss four of them: Graph-Based Models, Co-training and Multiview Models, Generative Models, and Semi-supervised Supported Vector Machines.

Graph-based models

Graph based semi-supervised learning (GSSL) is a method of semi-supervised learning where a graph is constructed so that the nodes represent all samples and the weighted edges

represent how similar two nodes are to each other. The nodes include both labeled and unlabeled nodes, and nodes associated with edges of larger weight are more likely to have the same label. The two main steps to learning in GSSL can be broken down into the construction of the graph and then deducing the label of the unlabeled nodes. While some problems may naturally have an underlying graph, in all cases we first want to construct a graph $G = (V, E, W)$ which is a 3-tuple consisting of the set of vertices V, a set of edges between vertices $E \subseteq V \times V$, and a weighted adjacency matrix W so that we may approach the problem graphically. For the sake of simplification we assume, the graph is undirected and thus W is symmetric, edge weights are non-negative where a weight of 0 implies there is no edge between two nodes, and the graph is simple and so contains no multi-edges or self loops. Under these constraints we are able to employ both unsupervised and supervised methods in the construction of the graph.

Unsupervised methods do not take into consideration any of the labeled data during the construction of the graph. These methods include the k-nearest neighbor approach which uses a pre-defined metric in order to quantify the similarity between nodes in the graph, and thus assign edge weights. In the construction of the graph, the k-nearest neighbors method does not care about the labels of the nodes since the goal here is to construct edges between the k nearest nodes based on some metric and not assign labels to nodes.

Supervised methods on the other hand do take into consideration information given on node labels. This information is used in order to hone the graph construction so that later tasks, such as extrapolating node labels, can be more refined. Overall, the construction of the graph provides the foundation for making problems appropriate for GSSL approaches.

The goal of the GSSL approach then is to find a function $f: X \rightarrow Y$ such that f minimizes the error of label assignments as much as possible, and f is smooth on the graph G.

That is, f is as accurate to the given labels as possible, and f changes gradually and smoothly as one moves along the nodes of the graph through the edges without having sudden jumps or discontinuities. The prediction function can be influenced by factors such as label information, graph structure, noise, the similarity metric, etc., and it is ultimately used to give unlabeled nodes labels. The basic algorithm for giving these labels is by first propagating labels for one step, row-normalize Y, and finally keeping the labeled data and repeating the steps until convergence.

Multiview models

Multi-view learning as a SSL method focuses on learning from multiple different sets of features that together make up the entire data. The purpose of having multiple views is to improve the performance of the learning algorithm by combining the different perspectives offered by the different feature sets. Through these different perspectives, we may better understand the underlying structure of the data more accurately and more robustly.

While multi-view learning may seem like it can only be applied to data which naturally splits into different feature sets, multi-view learning may still offer learning performance improvements even when applying it to artificially split data. Alternatively, data may be collected from using different methods of measurement if single-view data can't adequately describe information for all the examples. In multimedia cases for instance, the different mediums such as audio and video may both be used to describe the multimedia piece. Another example is a web page since they can be described by their text, images, videos, or by text attached to hyperlinks pointing to a given web page.

Co-training

Co-training is an SSL method which trains two separate classifiers on two different "sufficient and redundant" sets of features such that each set is sufficient for learning and each is conditionally independent of the other given the class label. The idea of using two different learners on two different sets is so that the predictions of each classifier can provide information to the other so that the other may improve its performance. In essence, two classifiers train on different sets so that each may provide insight on the unlabeled examples, and thus augment each other's training set. Assuming the training sets are sufficient and redundant, this model aims to maximize the agreement of the classifiers over the unlabeled data in order to reduce error.

Co-training is not limited to two feature sets. If we increase the number of feature sets then we can use multiple classifiers each trained on one of the sets such that each classifier is used in the labeling of the other feature sets. By using many classifiers, each classifier is able to learn from multiple views rather than only from two.

Overall, co-training uses two classifiers on two different features sets in order to improve learning performance by having the classifiers learn from each other, but this strategy may be augmented by combining it with the multi-view model so that many classifiers are used and learning from each other.

Generative Models

In semi-supervised learning, we are given l labeled data $\{\mathbf{x}_i, y_i\}_{i=1}^{l} \sim p(\mathbf{x}, y)$ and u unlabeled data $\{\mathbf{x}_i\}_{i=l+1}^{l+u} \sim p(\mathbf{x})$. Our goal is to learn a mapping $f : \mathcal{X} \to \mathcal{Y}$, where $f \in \mathcal{F}$ is in the hypothesis space. Here, $\mathbf{x}$ is the input vector (for example, a text sentence or an image), while y is the target label, which is discrete for classification problems. The unlabeled data $\{\mathbf{x}_i\}_{i=l+1}^{l+u}$ does not provide any information on the mapping from $\mathcal{X}$ to $\mathcal{Y}$ by itself, but it can be used to induce an implicit ordering of all $f \in \mathcal{F}$. We eventually want to find a predictor f^* that both rank high in the implicit ordering and can describe the labeled data well.

Generative models in semi-supervised learning are a type of machine learning approach that uses both labeled and unlabeled data to learn the underlying distribution of the data and then make predictions on unseen data. In the generative models, we assume that both labeled and unlabeled data follow the same parametric model. Let $\theta \in \Theta$ be the parameters of the joint probability, by Bayes law, the joint probability $p(\mathbf{x}, y|\theta)$ can be expressed as:

$$p(\mathbf{x}, y|\theta) = p(\mathbf{x}|y, \theta)p(y|\theta)$$

Each parameter θ corresponds to a mapping f_θ, by Bayes rule:

$$f_\theta(\mathbf{x}) \equiv \underset{y}{\operatorname{argmax}}\big[p(y \mid \mathbf{x}, \theta)\big] = \underset{y}{\operatorname{argmax}}\frac{p(\mathbf{x}, y|\theta)}{\sum_{y'} p(\mathbf{x}, y'|\theta)}$$

The implicit ordering of those f_θ's introduced by the unlabeled data is given by the log-likelihood of θ on unlabeled data, which is defined as:

$$\log p(\{\mathbf{x}_i\}_{i=l+1}^{l+u}|\theta) = \log \prod_{i=l+1}^{l+u} p(\mathbf{x}_i|\theta) = \sum_{i=l+1}^{l+u} \log(\sum_{y\in\mathcal{Y}} p(\mathbf{x}_i, y|\theta))$$

A higher log-likelihood implies that this parameter θ fits the unlabeled data better. Eventually, we want to choose a parameter θ that fits both labeled and unlabeled data well by taking the θ

that gives maximum combined log-likelihood:

$$\theta^* = \underset{\theta}{\operatorname{argmax}} \Big[\log p(\{\mathbf{x}_i, y_i\}_{i=1}^{l} | \theta) + \lambda \log p(\{\mathbf{x}_i\}_{i=l+1}^{l+u} | \theta) \Big]$$
$$= \underset{\theta}{\operatorname{argmax}} \Big[\sum_{i=1}^{l} \log p(\mathbf{x}_i | y_i, \theta) p(y_i | \theta) + \lambda \sum_{i=l+1}^{l+u} \log \Big(\sum_{y \in \mathcal{Y}} p(\mathbf{x}_i, y | \theta) \Big) \Big]$$

Semi-Supervised Support Vector Machines

In this section, we will first introduce the idea of Support Vector Machines(SVM), and then move on to the semi-supervised SVM. The SVM is a machine learning technique widely used in classification programs. It can be used to determine the optimal boundary between different classes or categories in the dataset. For example, if we have data points that belong to two different classes (for binary classification), the SVM will try to identify the best possible boundary that separates these two categories. For an SVM model to be optimal, the hyperplane it generates should not only separate the inputs correctly but also maximize the margin. Fig 1 shows the optimal hyperplane between two sets of data points represented by blue and green dots.

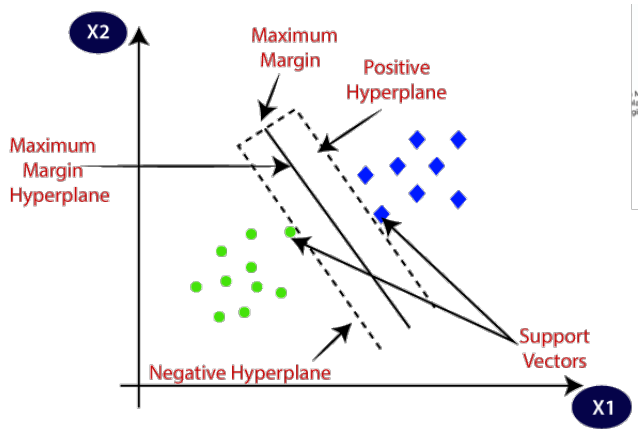


Figure 1: Maximum Margin Hyperplane from SVM

First, we consider a binary classification problem in supervised learning. The dataset is given by $\{(\mathbf{x}_i, y_i)\}_{i=1}^{l}$ where $y \in 1, -1$. Let the hyperplane be denoted as $\mathbf{w} \cdot \mathbf{x} + b = 0$. Since the distance d between two parallel hyperplanes $\mathbf{w} \cdot \mathbf{x} + b = -1$ (negative hyperplane) and $\mathbf{w} \cdot \mathbf{x} + b = 1$

(positive hyperplane) is $d = \frac{2}{||\mathbf{w}||^2}$, instead of maximizing d we can instead minimize $\frac{||\mathbf{w}||^2}{2}$:

$$\min_{\mathbf{w},b} \frac{||\mathbf{w}||^2}{2} \quad \text{subject to} \quad y_i(\mathbf{w}\mathbf{x}_i + b) \geq 1 \quad \forall i$$

We can get a solution by using the Lagrangian function:

$$L(\mathbf{w}, b, a) = \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{l} a_i(y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1) \quad \text{s.t. } a_i \geq 0$$

where a_i's are Lagrangian multipliers. The solution occurs when the partial derivative $L(\mathbf{w}, b, a)$ w.r.t $\mathbf{w}$ and b are both 0:

$$\frac{\partial}{\partial \mathbf{w}} L(\mathbf{w}, b, a) = \mathbf{w} - \sum_{i=1}^{l} a_i y_i \mathbf{x}_i = 0$$

$$\frac{\partial}{\partial b} L(\mathbf{w}, b, a) = \sum_{i=1}^{l} a_i y_i = 0$$

Plug Eq 7 and 8 back into Eq 6, and then maximizing the objective function, we get:

$$\max(a) = \sum_{i=1}^{l} a_i - \frac{1}{2} \sum_{i,j=1}^{l} a_i a_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j) \text{ s.t. } a_i \geq 0, \quad \sum_{i=1}^{l} a_i y_i = 0$$

From Eq 9, the optimal solution $a^* = (a_1^*, ..., a_l^*)^T$ can be calculated, and in turn, we can derive the optimal $\mathbf{w}$ and b:

$$\mathbf{w}^* = \sum_{i=1}^{l} a_i^* y_i \mathbf{x}_i$$

$$b^* = y_i - \mathbf{w}^* \cdot \mathbf{x}_i$$

Therefore, the optimal mapping $f : \mathcal{X} \to \mathcal{Y}$ is defined as:

$$f(\mathbf{x}) = \text{sgn}(\mathbf{w}^*\mathbf{x} + b^*)$$

This is the solution for the linear case. In the non-linear case, we need another mapping Φ to map from input space to the high-dimensional feature space. Eq. 12 will be modified to:

$$f(\mathbf{x}) = \text{sgn}(\mathbf{w}^*\Phi(\mathbf{x}) + b^*)$$

Now we have identified the way to construct an SVM in supervised learning. To better utilize the great amount of unlabeled data in reality, Semi-Supervised Support Vector Machines (S3VM) is introduced. We assume that the boundary is in a low-density region of unlabeled data, so a penalty term ξ is added to the Eq. 5:

$$\min_{w,b,\xi} \left[\frac{1}{2}\|w\|^2 + C\sum_{i=1}^{l}\xi_i\right] \quad \text{s.t.} \quad y_i(w\cdot \mathbf{x}_i + b) \geq 1 - \xi_i \quad \text{and} \quad \xi_i \geq 0$$

where $\xi_i = \max(1 - y_i[\mathbf{w}\cdot\mathbf{x}_i + b], 0)$. Plug in, the equation becomes

$$\min_{\mathbf{w},b}\{\frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{l}\max(1 - y_i[\mathbf{w}\cdot\mathbf{x}_i + b], 0)\}$$

Where the first term is the regularization term and the second term is the loss function over labeled data. Now we suppose that the unlabeled data are also labeled, then $y_i = \text{sgn}(\mathbf{w}\mathbf{x} + b)$. The loss function over one unlabeled data point would be:

$$\max(1 - y_i[\mathbf{w}\cdot\mathbf{x}_i + b], 0) = \max(1 - |\mathbf{w}\cdot\mathbf{x}_i + b|, 0)$$

Combine all three terms together, we get the function to minimize:

$$\min_{w,b}\{\frac{1}{2}\|w\|^2 + C_1\sum_{i}^{l}\max(1 - y_i[\mathbf{w}\cdot\mathbf{x}_i + b], 0).. + C_2\sum_{i=l+1}^{l+u}\max(1 - |\mathbf{w}\cdot\mathbf{x}_i + b|, 0)\}$$

Where C_1 and C_2 are the weights of loss over labeled and unlabeled data. In the algorithms, it is difficult to find such $\mathbf{w}$ and b to optimize this function since the loss is non-convex.

Logistic Regression

In this section, we will introduce the mathematical derivation of logistic regression which we later used in the algorithm. Again we consider a binary classification problem, the output variable $y = 0$ or 1. Let the conditional probability be $P(y = 1|\mathbf{x}) = p(\mathbf{x})$. The assumption for linear regression is that the log likelihood ratio of class distributions is linear:

$$\log\left(\frac{p(\mathbf{x})}{1 - p(\mathbf{x})}\right) = \beta^T\mathbf{x} + \beta_0$$

Here, β is a set of parameters in the model. Solving for p from this equation, we get

$$p(\mathbf{x}) = \frac{e^{\beta^T \mathbf{x}+\beta_0}}{1+e^{\beta^T \mathbf{x}+\beta_0}} = \frac{1}{1+e^{-(\beta^T \mathbf{x}+\beta_0)}}$$

When $p(x) \geq 0.5$, the classifier should predict 1, while when $p(x) < 0.5$, the classifier should predict 0. This is equivalent to predicting 1 whenever $\beta^T \mathbf{x} + \beta_0 \geq 0$ and the classifier we obtained is essentially a linear classifier with desicion boundary given by the solution of equation $\beta^T \mathbf{x} + \beta_0 = 0$. Since the logistic regression predicts a probability (instead of just outputting the classes), we can compute the likelihood:

$$L(\beta, \beta_0) = \prod_{i=1}^{n} p(\mathbf{x}_i)^{y_i} (1 - p(\mathbf{x}_i))^{1-y_i}$$

And we can also obtain the log likelihood function:

$$\begin{aligned}
\ell(\beta_0, \beta) &= \sum_{i=1}^{n} \Big[y_i \log p(\mathbf{x}_i) + (1-y_i) \log(1 - p(\mathbf{x}_i)) \Big] \\
&= \sum_{i=1}^{n} \log \big(1 - p(\mathbf{x}_i)\big) + \sum_{i=1}^{n} y_i \log \frac{p(\mathbf{x}_i)}{1 - p(\mathbf{x}_i)} \\
&= \sum_{i=1}^{n} -\log(e^{\beta_0 + \beta^T \mathbf{x}_i} + 1) + \sum_{i=1}^{n} y_i (\beta_0 + \beta^T \mathbf{x}_i)
\end{aligned}$$

To find the maximum log likelihood, we take the derivative of the equation with respect to each parameter β_j: (here $\mathbf{x}_i j$ means the jth component of $\mathbf{x}_i$)

$$\begin{aligned}
\frac{\partial \ell}{\partial \beta_j} &= -\sum_{i=1}^{n} \frac{1}{1 + e^{\beta_0 + \mathbf{x}_i \cdot \beta}} e^{\beta_0 + \mathbf{x}_i \cdot \beta} \mathbf{x}_{ij} + \sum_{i=1}^{n} y_i \mathbf{x}_{ij} \\
&= \sum_{i=1}^{n} \left(y_i - p\left(\mathbf{x}_i; \beta_0, \beta\right) \right) \mathbf{x}_{ij}
\end{aligned}$$

The zero point of this derivative (which is equivalent to the maximum point of log probability can be numerically approximated to solve for the optimal parameters $\{\beta_i\}$.

Stochastic Gradient Descent

A common approach to learning it to minimize the cost function in order to make accurate predictions based on some input training data. Given the cost function $C(w) = \frac{1}{N}\sum_{i=1}^{N} J(z_i, w)$ and the discrete distribution $dP(z) = \sum_{i=1}^{N} \frac{1}{N}\delta(z - z_i)$ the gradient descent (GD) algorithm updates weights at each iteration in the following way:

$w_{t+1} = w_t - \epsilon_t \nabla_w C(w_t) = w_t - \epsilon_t \int \nabla_w J(z, w_t) dP(z) = w_t - \frac{\epsilon_t}{N}\sum_{i=1}^{N} \nabla_w J(z_i, w_t)$. The weights are updated with the intention of minimizing the cost function. Because typically a large training set is needed for learning, each update iteration of the weights becomes computationally expensive since calculating $\nabla_w J(z, w)$ over the entire training set is necessary for calculating the average. As an alternative to this we consider consider the following update rule for the training weights: $w_{t+1} = w_t - \epsilon_t \nabla_w J(z, w_t)$. With this update rule we have what is called stochastic gradient descent (SGD). The SGD algorithm chooses a training example z at random at each iteration to update the weights. Naturally, SGD iterates much quicker than GD since it calculates the gradient only at one random point z instead of over all of them. Furthermore, while GD may converge to a local minimum and not be able to escape this local minimum, SGD is likely to not get stuck. This is due to the random behavior of SGD. Overall, SGD is able to converge to a minimum of the empirical cost $C(w)$, but if a training set is too small it may overfit the data.

Algorithms

Due to the expansive nature of semi-supervised learning, there are a lot of possible classifiers that can be used to train a model. These classifiers, which describe how the data is split up and expressed, are important for the understanding of the topic of "Semi-supervised

learning". There are classifiers specifically for data to be separated into classes, for example, while there are others which are used for prediction only. As mentioned earlier, we'll analyze the classifiers and models in which the data is categorized under labels and clusters.

For this section, we will look into the classifier that uses Logistic Regression. Logical Regression is a form of Classification Expectation Maximization or CEM, in which data points are labeled in an iterable fashion with three main steps. Of course, the nature of semi-supervised learning is that there is both labeled and unlabeled data. Therefore, Logistic Regression will abuse that fact in creating our model. Furthermore, Logistic regression is a linear classifier, so we will be using a linear function (logit) with the predicted weights generated by the model. The reason why this is established as "logistic" regression is due to how the weights are created - which typically involves the maximum likelihood estimation (MLE) which involves the log likelihood.

The steps of a typical logistic regression or Logistic-CEM are as follows. First, we train a logistic model over the labeled data. This gives us a baseline of the data in which we will adjust as we iterate over the data. Then, we start the iterative process of adjusting our model to the unlabeled data. For each iteration, there's three main steps: Expectation calculation, Cluster Assignment, and Maximization. In the expectation step, we estimate the posterior probability that the current data in question exists in one of the current labels/clusters. In the Cluster Assignment, we then pick the cluster with the maximal posterior probability and assign the data point to it. Finally, in the Maximization step, we update the label parameters of all the labels to maximize the log likelihood of all the data within the cluster. These are the 3 steps done iteratively to generate our label parameters.

Here's the code for a Logistic-CEM program using sklearn's Logistic Regression Model:

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression

# Generate some example data
x, y = make_classification(n_samples=1000, n_features=10, n_informative=5,
                           n_redundant=0, n_classes=2, random_state=42)

# Create Training and Testing Data
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=42)

# Train the Logistic-CEM model
model = LogisticRegression(solver='liblinear', random_state=0)
model.fit(x_train, y_train)

# Predict labels for the test data
y_pred = model.predict(x_test[:,:-1])

# Compute accuracy of the predictions
acc = accuracy_score(y_test, y_pred)

print(f"Accuracy: {acc}")
print("Predicted labels:")
print(y_pred)
```

As mentioned above, here is some example code that uses scikit's logistic regression model to train a model with data and creates predicted labels and an accuracy score based on the model. Note, in this code we randomly generate data to use the model on. In the real world, the data would be generated from whatever dataset you have but as we don't have yet a dataset to test on, the data for testing purposes is randomly generated. Furthermore, for analysis purposes,

recognize that this code is using the Logical Regression model made by scikit-learn which uses a base max-iteration at 100. It's recommended to adjust the algorithm with several different parameters to better train the model, however with a lack of dataset to train on if testing the code yourself.

In theory, this form of classifier can be both pretty memory-intensive and computationally-intensive. As we regenerate the label parameters on each "M" step, a lot of calculations can be done every run through. Furthermore, there are plenty of matrix-calculations done on each step (Especially on the "C" and "E" steps), so it's pretty memory-intensive. Also, depending on how many iterations we want to do to train the model, then the model can be even more computationally-intensive. That being said, as the number of computations is limited by the size of the matrices, it can be a lot more memory-intensive than computationally intensive, especially with large datasets. Therefore, this type of classification model is better used in small-datasets in which you want to sort the data in different clusters and/or labels.

This next section is useful to understand some of the underlying issues you may come across when training models:

1. Margin Bias: A common issue where the algorithm prefers samples that are closer to the boundary. This is due to unlabeled data points at this boundary having a high chance of being mislabeled, and given a higher weight in the learning process. This causes overfitting to the nearby data points. To counteract this, there should be a regularization term to penalize the algorithm when there is uncertainty near the boundary.

2. Noise: Noise is irrelevant data that is found within the unlabeled dataset, such that it will negatively impact the performance of the new algorithm and often will lead to overfitting.

3. Overfitting: This is when the model is overtrained on noise within the unlabeled data. This leads to the model being unable to generalize to new data. To counteract this, the size of the labeled dataset should be comparable to the unlabeled set or maintaining regularization.

4. Angle Variances: This is an issue that comes up in manifold regularization. (Minimization of a loss function). This can cause problems if the model cannot account for the orientation of the manifold in different spaces. Often, the manifold will cause dimensions of the space to be miss accounted for, either overutilized, or underutilized. To solve this, often alternative distance calculations are used (one such model used a linear function to convert it into a Gaussian distribution).

5. Finding Feature Vectors: Simply, if there are extra features that are not very impactful to the model's accuracy, this may lead to the model underperforming, as it has over emphasized the importance of these features. Likewise, if important features are ignored, then the model will underperform.

6. Bias in the labeled set: If you train the model on a subset of a complete dataset, (such as only training the model on one language, and trying to use it on another), where not all of the set is represented, the resulting predicted data will be biased, and will not be able to generalize to the larger set.

Another algorithm we will take a look at is text classification. As with other algorithms, this algorithm works by implementing labeled data to classify unlabeled data. Often used in Natural Learning Processing, (Such as Chat-GBT), this field of research is defined as SSTC (Semi-Supervised Text Classification).

For this example, we will only be displaying a basic model of SSTC, deployed using Python. The code is as follows below.

```
import numpy as np
```

```
from sklearn.linear_model import LogisticRegression
from sklearn.feature_extraction.text import CountVectorizer
from datasets import load_dataset

# load the IMDB dataset from the datasets library, this is a binary
classification dataset with 25k training and 26k test examples
imdb = load_dataset("imdb")

def sstc(train_data, test_data, train_labels, test_labels):
    '''
    This function trains a logistic regression model on the IMDB dataset
and prints the accuracy on the test set using scikit-learn
(LogisticRegression)
        Parameters:
            train_data (list): list of strings, each string is a review
from the training set
            test_data (list): list of strings, each string is a review
from the test set
            train_labels (list): list of integers, each integer is the
label (0 or 1) for the corresponding review in train_data
            test_labels (list): list of integers, each integer is the
label (0 or 1) for the corresponding review in test_data
    '''
    # create a vectorizer object to generate feature vectors, we will use
word counts as features
    vectorization = CountVectorizer()
    X_train = vectorization.fit_transform(train_data)
    X_test = vectorization.transform(test_data)

    # train a logistic regression model on the training set
    model = LogisticRegression(random_state=0).fit(
        X_train, train_labels)

    # make predictions on the test data
    y_pred = model.predict(X_test)
    return np.mean(y_pred == test_labels), y_pred

# create a list of training set sizes to iterate over, we will use 1k,
10k, and 50k examples.
```

```
# You will notice that as the training set size increases, the accuracy
increases.
size = [1000, 10000, 25000]
results = []
for i in size:
    train_set = imdb["train"].shuffle(seed=42).select(
        [i for i in list(range(i))])
    test_set = imdb["test"].shuffle(seed=42).select(
        [i for i in list(range(i))])

    # split the data into text and labels
    train_data = train_set["text"]
    test_data = test_set["text"]
    train_labels = train_set["label"]
    test_labels = test_set["label"]

    # convert the labels to numpy arrays for use with scikit-learn
    y_train = np.array(train_labels)
    y_test = np.array(test_labels)
    results.append(sstc(train_data, test_data, y_train, y_test))

for i in range(len(size)):
    print("Accuracy on test set with {} training examples: {}".format(
        size[i], results[i][0]))
    print("Predictions on test set with {} training examples: {}".format(
        size[i], results[i][1][0:10]))

# Accuracy on test set with 1000 training examples: 0.789
# Predictions on test set with 1000 training examples: [1 0 0 1 0 1 0 0 0
0]
# Accuracy on test set with 10000 training examples: 0.8593
# Predictions on test set with 10000 training examples: [1 0 1 1 0 1 0 0 1
1]
# Accuracy on test set with 25000 training examples: 0.86396
# Predictions on test set with 25000 training examples: [1 0 1 1 0 0 0 0 0
1]
```

This algorithm does a version of sentiment analysis, where user reviews are manually marked as either positive (1), or negative, (0). We then convert the text into a vectorizer object so that we can create our training features for the model, then use the LogisticRegression (same classifier method as used above, but with a different solver) to train our model. Afterwards, we use this model on the test data and see what labels it comes up with for an unlabeled dataset. We check with what it should actually be to get our accuracy. To that point, I wanted to show that as the model gets more data to work on, we would see that the results become more accurate to what we expect.

The computational and memory use of this algorithm can be quite high due to the logistic regression. Firstly, depending on how large of a dataset you want to train with, just vectorizing it and storing it in memory can be expensive for the memory. Further, Logistic regression implements an iterative optimization, which, has an average runtime of O(*number of iterations* * *number of features* * *number of data points*), which can be quite complex depending on the sizes and how accurate you want your model to be. In this case, it was time efficient and memory efficient because of how small of a set it was trained on, and a minimum of features and iterations were used. For more complex models, you can improve the algorithm by splitting up the dataset more, or using sparse matrices for developing the features. There are also other algorithms that are more efficient than Logistic regression for more complex models.

Data Exploration

Datasets:

We selected 2 datasets that both involve binary sentiment classification. The input is a short text and the output is {0, 1}, representing negative and positive sentiment, respectively. The two datasets are

The IMDB dataset (available at https://huggingface.co/datasets/imdb), which contains 100k highly-polarized movie reviews. The original dataset has the following split: 25k training (labeled data), 25k test (labeled data), and 50k unsupervised (unlabeled data). For example, a data entry in the training set would look like this:

Input (text)	Output (class label)
"Ned aKelly is such an important story to Australians but this movie is awful. It's an Australian story yet it seems like it was set in America. Also Ned was an Australian yet he has an Irish accent...it is the worst film I have seen in a long time"	0 (negative)

In the actual training, to simulate a classical semi-supervised learning task, which features limited labeled dataset and large unlabeled dataset, we select 4% of the original training dataset as our labeled data. The other parts of the dataset are used in their entirety. This results in the following dataset split: 1k labeled training data, 50k unlabeled training data, and 25k labeled test data.

Amazon review dataset (available at https://huggingface.co/datasets/amazon_polarity), which contains about 35 million reviews on Amazon. An entry in the original dataset contains the comment title, the comment text, and a classification label. In the actual training, we only use the text and label. A data entry in the training set would look like

Input (text)	Output (class label)
"This was a easy to read book that made me want to keep reading on and on, not easy to put down.It left me wanting to read the follow on, which I hope is coming soon. I used to read a lot but have gotten away from it. This book made me want to read again. Very enjoyable."	1 (positive)

In training, again we try to simulate a typical semi-supervised learning task featuring a large unlabeled dataset. Also, since the dataset is very large, to shorten the training time, we selected the first and last 1% of the training set as our labeled data, the middle 20% (40%-60%) of the

training set as our unlabeled data, by artificially removing the label, and the middle 4% of the test dataset (48%-52%) as our test data. This results in the following dataset split: 72k labeled training data, 720k unlabeled training data, and 16k labeled test data.

Parameters:

In the semi-supervised learning task where we use the logistic regression as the classifier, we have 2 major parameters in use:

1. The probability threshold p

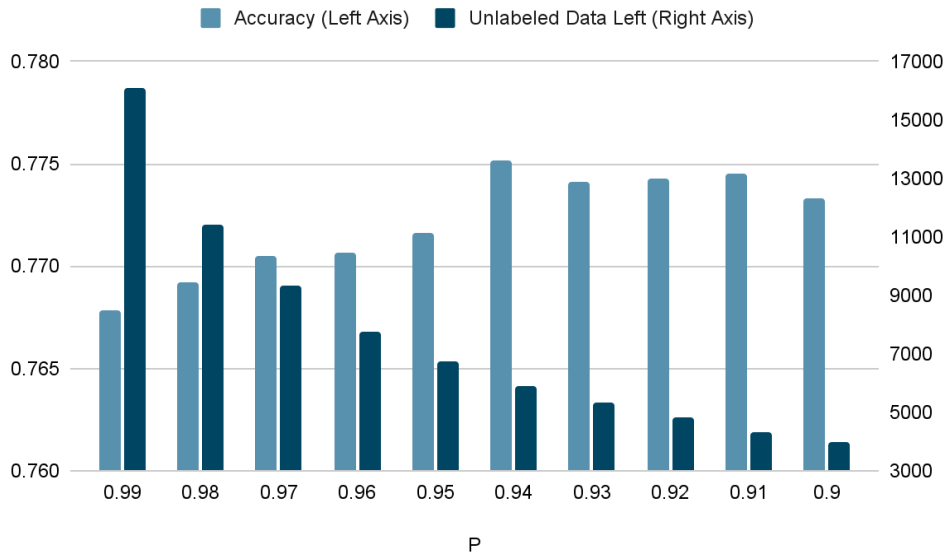
In training, we continuously make predictions on unlabeled data. Our classifier will give two probabilities with regard to which category the data belongs to: P(0) (resp. P(1)) = the probability that the text expresses negative (resp. positive) sentiment. P(0) + P(1) = 1. If the probability of either P(0) or P(1) is greater than the threshold p, the unlabeled data will be added to labeled data where the label is argmax(i $\in${0, 1}) P(i). As we increase the value of p, we are essentially adding less unlabeled data each iteration in the training, because we require the added data to be classified with higher confidence. It is difficult to predict how the performance will vary when we change p. On the one hand, adding data with high prediction confidence means the labels of these added data are of higher quality (i.e. mostly right). This means we have a larger, and high-quality training dataset. On the other hand, data with high prediction confidence could be very similar to those in the training dataset. In our context, for example, data with a high P(1) or P(0) probably contain several key sentiment-identifying keywords that are already present in the training dataset, and therefore already learned by our regression model. Therefore, a high p could be detrimental if we want our regression model to learn new features.

We ran experiments on both datasets and the results are in the following table:

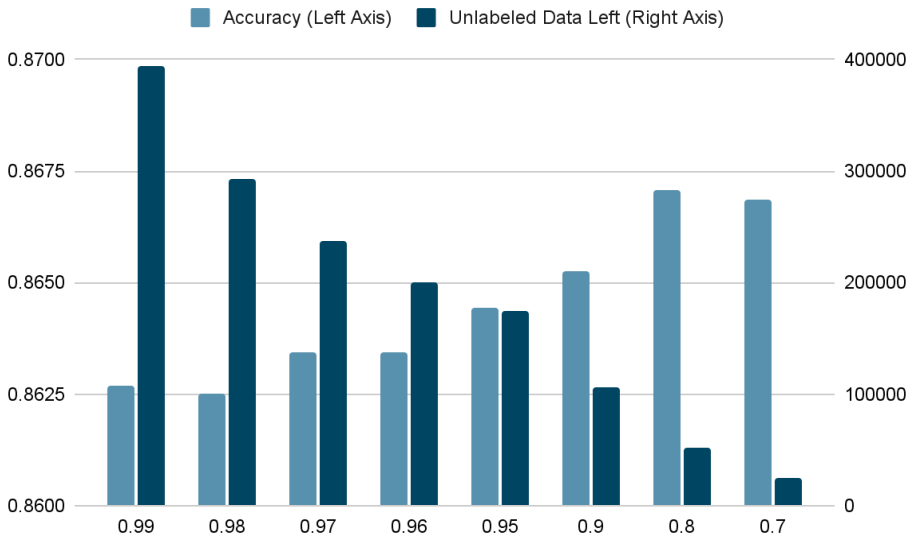
IMDB	Amazon

P	Accuracy	Unlabeled data left	P	Accuracy	Unlabeled data left
0.99	0.76784	16105	0.99	0.8626875	394000
0.98	0.7692	11426	0.98	0.8625	292560
0.97	0.77052	9307	0.97	0.8634375	237018
0.96	0.77064	7737	0.96	0.8634375	200331
0.95	0.7716	6757	0.95	0.8644375	175231
0.94	0.7752	5918	0.9	0.86525	106654
0.93	0.77408	5331	0.8	0.8670625	52040
0.92	0.77428	4808	0.7	0.866875	24972
0.91	0.77452	4340			
0.9	0.77328	3970			

Graph for IMDB:



Graph for Amazon:



As we can see, when lowering p, more unlabeled data are added during the training. The number of low-confidence data left decreases as p decreases. The accuracy of our algorithm first increases, because more features are learned by the regression model if we add more unlabeled data. However, the accuracy began to decrease at some p, although this p varies under different datasets. This could be due to some undesirable features being learned by our algorithm, if we add samples that the classifier not feeling extremely confident in the prediction. The optimal choice of p, therefore, depends on the dataset. For example, in the IMDB dataset, we should choose p = 0.94 where the accuracy is the highest. In the Amazon dataset, we should select p = 0.8, although a more accurate value can be determined by running more experiments.

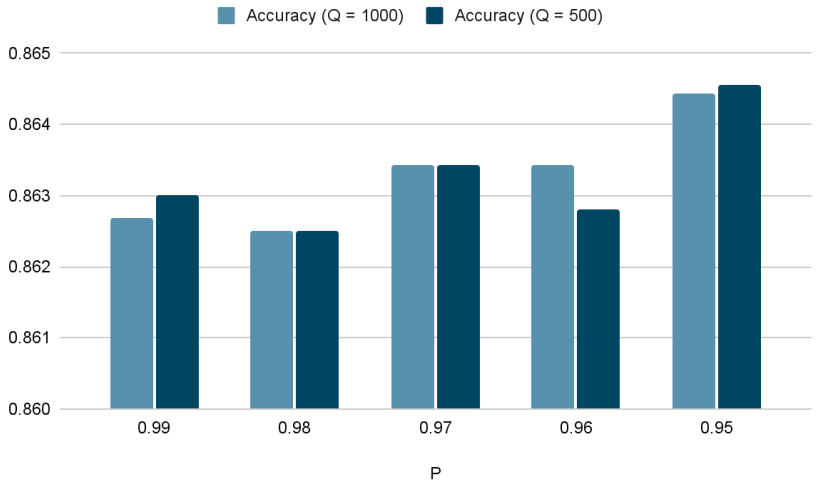
2. The stop condition q

We keep track of the number of unlabeled data that enters our labeled dataset in every iteration, denoted q here. As more high-confidence unlabeled data are added, we will observe a declining q because the most leftover data are the data that the regression model feels uncertain about. In our IMDB dataset, we set q = 0 as the stop condition. In theory, the lower the q, the better the model should perform because we are adding more data under a lower q. However, the trade-off is

longer training time: in later iterations, very few unlabeled data can be predicted with high confidence. If we wait until the point where the algorithm can no longer find any data where P(0) or P(1) > p we specified, there will be a lot of terminal iterations with limited improvements. Therefore, in the Amazon dataset, we increased the value of q. In theory, this should not have a major impact on our outcome because a higher q would not decrease the number of added unlabeled data by too much.

For our Amazon dataset, we experimented with 2 different stop conditions (2 different q, which are 1000 and 500, respectively). The training time under 1000 is shorter.

	Q = 1000		Q = 500	
P	Accuracy	Unlabeled data left	Accuracy	Unlabeled data left
0.99	0.8626875	394000	0.863	393180
0.98	0.8625	292560	0.8625	292152
0.97	0.8634375	237018	0.8634375	236146
0.96	0.8634375	200331	0.8628125	199912
0.95	0.8644375	175231	0.8645625	174252



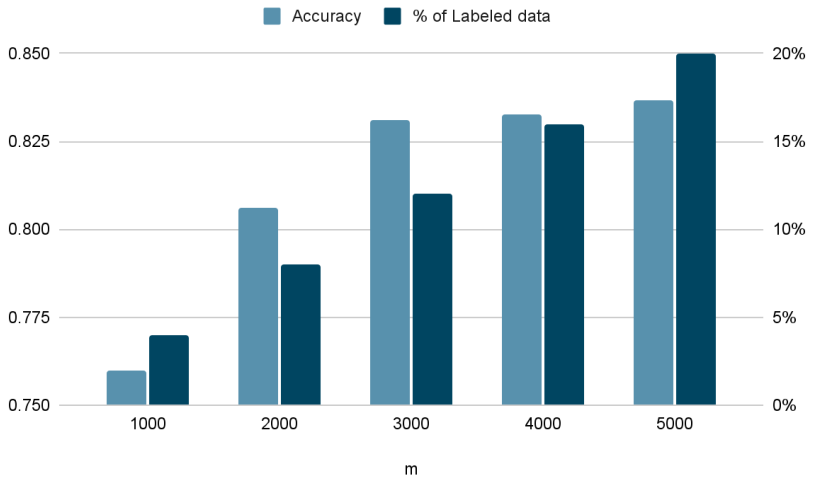
Accuracy under 2 different values of q do not vary by much. This is because the numbers of unlabeled data added during the training are almost the same, especially considering the large

size of the dataset. The optimal choice of q, in theory, should be 0, because in this way we ensure the number of the added unlabeled data is maximized. However, if the training time is too long, we can consider increasing q a bit to shorten the training time, as long as q is small relative to the size of the dataset.

Performance under different n and m:

This experiment is run on the IMDB dataset. We first keep n = 25000 constant and change m. In this way, we are essentially increasing the proportion of labeled data in our training dataset. To make sure the results are unaffected by the choice of p and q as we specified above, we select p = 0.99 and q = 100.

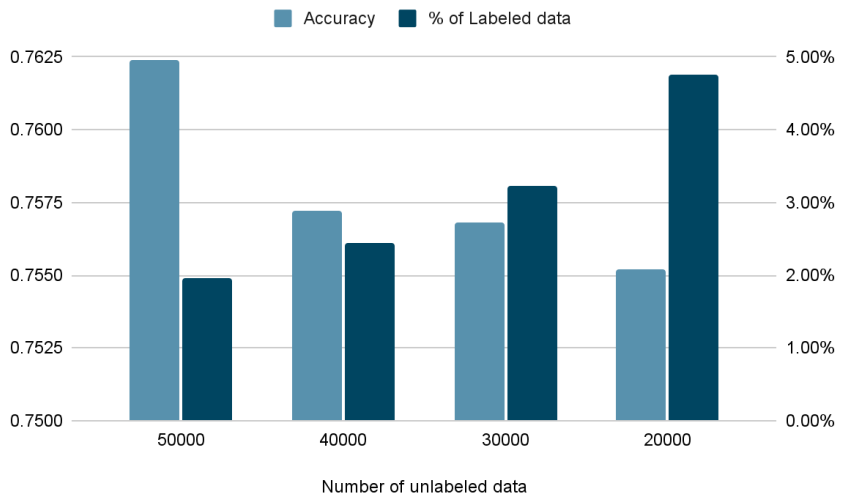
m (Labeled)	n (All)	Accuracy	Remaining unlabeled data	% of Labeled data
1000	25000	0.76	10373	4%
2000	25000	0.806	11432	8%
3000	25000	0.8312	11253	12%
4000	25000	0.8328	10796	16%
5000	25000	0.8368	10449	20%



As we increase the proportion of labeled data in the training, the accuracy increases. However, the accuracy stuck as m is over 3000. This could be due to the phenomenon that almost all properties that reveal the sentiment are already being learned by the classifier.

We also tried to keep the number of labeled data constant at 1000 (m = 1000) and increase the size of unlabeled data. We also used the IMDB dataset for this task and select p = 0.99 and q = 100.

m (Labeled)	n-m (Unlabeled)	Accuracy	Remaining unlabeled data	% of Labeled data
1000	50000	0.7624	16394	1.96%
1000	40000	0.7572	13924	2.44%
1000	30000	0.7568	11725	3.23%
1000	20000	0.7552	9114	4.76%



As we decrease the number of unlabeled data while keeping the number of labeled data constant, the overall accuracy slightly decreases. However, there is no substantial impact on the accuracy. The accuracy is largely dependent on the number of labeled data. We may therefore conclude that there are few additional features to be learned in unlabeled data. Therefore, the unsupervised part of this semi-supervised learning task only gives marginal improvement to the model.

Connection to the Class

Semi-supervised learning works with many algorithms that are adjacent to classwork discussion. For example, while the Logistic Regression algorithm uses only stochastic gradient descent (SGD) compared to pure gradient descent (GD) as described in class, the algorithms are mostly the same. Both attempt to find the minima using a set of parameters, and likewise both work to minimize the cost function.

That being said, Stochastic is used in these kinds of algorithms because it minimizes the batch of training samples that are required to train a model, greatly increasing the runtime, especially if the sample to train with is very large. This sample is created from a random collection of data from the main training set, and used to train the model every epoch. In exchange, you may get a worse error function than with pure GD, but a close approximation to what GD would have produced.

The least square method is often used in linear regressions, as a technique to produce coefficients on independent variables that minimize the mean squared error. The linear regression aims to estimate a continuous dependent variable while the logistic regression estimates a discrete (mostly dichotomous, i.e. 0 and 1) dependent variable. The logistic regression has some similarity to linear regressions, although it does not use least square method to produce an estimation model. In logistic regression, we fit the model to our data by minimizing the logistic loss, or cross-entropy loss over all samples (note that different packages may have slightly different approaches to optimize a logistic regression model. We used the scikit learn package in actual implementation, which uses cross-entropy loss). The logistic regression loss for a specific sample is calculated as:

$$L_{\log}(y, p) = -\log \Pr(y|p) = -(y \log(p) + (1 - y) \log(1 - p))$$

where y is the sample's true class (label, y = 0 or 1), p = P (y = 1), which is the probability that the sample's label is 1, produced by our model. The loss is -log(p) = -log(P (y = 1)) if the sample's label is 1, or -log(1-p) = -log(P (y = 0)) if the sample's label is 0. By minimizing the loss, we are essentially trying to make P (y = true label) as large as possible. This can be considered as minimizing the difference between P (y = true label) and 1. In this sense the logistic regression loss has some similarity to the least square, which minimizes the distance between the predicted y and actual y.

Further Discussion

While researching for this paper, I discovered the notion of a manifold, which takes a set of points defined in a higher dimension down to a lower dimension for easier use. PCA is often used to create this manifold, as it takes a matrix of NxN, down to a size of nxn, where nxn defines the most important components of the original matrix. This is often useful when working with datasets, however, some of the information would be lost in this transformation, and I

wanted to explore how researchers counteract that loss. For example, when there is sentiment analysis, how do you use a manifold that takes eight values down to two principal components, and then use that to create assumptions about the eight? Do they just create 7 boundaries in this 2 dimensional space?

Bibliography:

3.3. metrics and scoring: Quantifying the quality of predictions. scikit. (n.d.). Retrieved April 16, 2023, from https://scikit-learn.org/stable/modules/model_evaluation.html#classification-metrics

Banerjee, W. (2020, August 27). *Train/Test Complexity and space complexity of logistic regression*. Medium. Retrieved April 16, 2023, from https://levelup.gitconnected.com/train-test-complexity-and-space-complexity-of-logistic-regression-2cb3de762054

Banerjee, W. (2020, August 27). *Train/Test Complexity and space complexity of logistic regression*. Medium. Retrieved April 16, 2023, from https://levelup.gitconnected.com/train-test-complexity-and-space-complexity-of-logistic-regression-2cb3de762054#:~:text=So%2C%20the%20runtime%20complexity%20of,of%20the%20data%20is%20small.

Ding, S., Zhu, Z., & Zhang, X. (2015, November 18). *An overview on semi-supervised support Vector Machine - Neural Computing and Applications*. SpringerLink. Retrieved April 16, 2023, from https://link.springer.com/article/10.1007/s00521-015-2113-7

Duarte, J. M., & Berton, L. (2023, January 31). *A review of semi-supervised learning for Text Classification - Artificial Intelligence Review*. SpringerLink. Retrieved April 16, 2023, from https://link.springer.com/article/10.1007/s10462-023-10393-8

Goldberg, A., Zhu, X., Singh, A., Xu, Z., & Nowak, R. (2009, April 15). *Multi-manifold semi-supervised learning*. PMLR. Retrieved April 16, 2023, from http://proceedings.mlr.press/v5/goldberg09a.html

ISSN (online) 2394-2320 (IJERCSE) vol 4, issue 9 ... - technoarete. (n.d.). Retrieved April 16, 2023, from https://www.technoarete.org/common_abstract/pdf/IJERCSE/v4/i9/Ext_08954.pdf

Logistic Regression - Carnegie Mellon University. (n.d.). Retrieved April 16, 2023, from https://www.stat.cmu.edu/~cshalizi/uADA/12/lectures/ch12.pdf

Real Python. (2022, September 1). *Logistic regression in python*. Real Python. Retrieved April 16, 2023, from https://realpython.com/logistic-regression-python/

Semi-supervised Learning - University of Wisconsin–Madison. (n.d.). Retrieved April 16, 2023, from https://pages.cs.wisc.edu/~jerryzhu/pub/SSL_EoML.pdf

Semi-supervised Logistic Regression - Jenkins Society Reunion. (n.d.). Retrieved April 16, 2023, from http://www.yaroslavvb.com/papers/amini-semi.pdf

Semi-supervised text classification with Balanced Deep Representation ... (n.d.). Retrieved April 16, 2023, from https://aclanthology.org/2021.acl-long.391.pdf

van Engelen, J. E., & Hoos, H. H. (2019, November 15). *A survey on semi-supervised learning - machine learning*. SpringerLink. Retrieved April 16, 2023, from https://link.springer.com/article/10.1007/s10994-019-05855-6

Wang, R., Jia, X., Wang, Q., Wu, Y., & Meng, D. (2022, September 29). *Imbalanced semi-supervised learning with Bias Adaptive Classifier*. OpenReview. Retrieved April 16, 2023, from https://openreview.net/forum?id=rVM8wD2G7Dy

Zhang, T., Zhu, T., Han, M., Chen, F., Li, J., Zhou, W., & Yu, P. S. (2022, October 1). *Fairness in graph-based semi-supervised learning - knowledge and information systems*. SpringerLink. Retrieved April 16, 2023, from https://link.springer.com/article/10.1007/s10115-022-01738-w

Zhou, Z., Lu, C., Wang, W., Dang, W., & Gong, K. (2022, July 2). *Semi-supervised Medical Image Classification based on attention and intrinsic features of samples*. MDPI. Retrieved April 16, 2023, from https://www.mdpi.com/2076-3417/12/13/6726