# ECE570 Term Paper: Zero-shot classification with Label Propagation (ZLaP)

**Anonymous Submission**

## Abstract

In the modern age of machine learning, problems have expanded to tasks in less-than-ideal environments, such as occluded or totally absent training data. In this term project, we address zero-shot classification by re-implementing the Zero-shot classification with Label Propagation (ZLaP) method (Stojni'c, Kalantidis, and Tolias 2024), which encodes both class names and unlabeled images into a shared latent space, enabling label propagation through K-nearest neighbor (KNN) search using a metric of geodesic distance in a graph-based structure. Our approach had two phases: first, we re-implemented ZLaP's three learning modes, matching the baseline accuracies on the CalTech101 dataset (89.61% `transductive`, 86.90% `inductive`, and 86.33% `sparse-inductive`) using an `RN50-OpenAI` backbone in CLIP. In the second phase, we extended ZLaP by analyzing K as a hyperparameter and adapting ZLaP's code for practical and rapid classification of retail products by supplying arbitrary images and labels into the code without need for re-training. Hyperparameter tuning verified the K, alpha, and gamma selections made by the authors, and demonstrated that `transductive` and `inductive` learning required a higher gamma (5.0) with a moderate K (10) and alpha (0.3–0.5), while `sparse-inductive` mode was highly resilient across settings. Overall the re-implementation was successful, and manages to verify and extend the work of the original ZLaP authors. Possible expansions of this project could explore alternative similarity metrics, and verify results using more datasets.

**Code** — https://anonymous.4open.science/r/ZLaP-ECE-570-Final-Project-66A6/

## Introduction

The topic of this project is "classification of sources with limited training data represented in the latent space". Within the landscape of artificial intelligence, traditional problems such as image classification and object detection have quite robust solutions by now, such as CNNs like YOLO which can be trained by novice users. However the current landscape presents more complex challenges involving inherently noisy or incomplete data, meaning training data is intrinsically obscured from the model in tasks such as speech recognition or computer vision. A key example of this issue is ZERO-SHOT CLASSIFICATION, where a model must classify data without any prior examples or labels. Many methods emerged to collectively address issues in self-supervised or unsupervised learning using noisy data, and present novel ways to generalize models without additional (expensive) training data.

Vision models typically require vast amounts of hand-labeled data, which is costly and time-consuming to gather, and often requires data augmentation to achieve solid performance. In the absence of sufficient training data, classical (non-ML) approaches to computer vision are possible, but often necessitate extensive experimentation and manual tuning, hindering generalization. Therefore, building models which operate effectively with minimal training data is critical for having generalizable artificial intelligence

This term paper focuses on using Zero-shot Classification with Label Propagation (ZLaP) to address these challenges. Our key objectives include matching the baseline accuracy of a publicly-available ZLaP implementation, verifying their choice of hyperparameters, and finally extending their code to work for any arbitrary dataset of images and class labels. To achieve these goals, my implementation approach involved identifying functions utilized in the ZLaP code, re-implementing them, and adding additional functions to facilitate hyperparameter validation and acceptance of arbitrary images/labels. Notably, the primary goal of this project was re-implementation, but the extension is an addition on this Minimum Viable Product.

## Related Work

### CLIP (Contrastive Language-Image Pre-training)

Developed by OpenAI, CLIP (Kim 2021) is a model designed to translate images and text into the latent space. It is trained to associate images with corresponding text descriptions of them, which has applications towards zero-shot learning due to it not requiring any specific training data during deployment.

We expand on CLIP in our ZLaP implementation to obtain the embeddings of our images and text for our label propagation step. Because CLIP will embed images and text based on a measure of semantics, this sets up the ability to propagate class prediction information outward based on similarity.

### K-Nearest Neighbors in Embedding Space

K-Nearest Neighbors (KNN) is an algorithm to retrieve some number of ”neighbors” nearest to some item in any type of graph or space. Since embeddings generated from tools like CLIP map similar data points closer together in the latent space, a KNN can utilize this structure to retrieve neighbors based on how semantically-similar they are.

We build upon a KNN during label propagation in this paper. Specifically, it's used to find labels for images whose location in the embedded space is geodesically-close to the embedding for the image. This forms the basis for propagating labels towards similar/close unlabeled features in the latent space.

## Problem Definition

This term paper addresses the problem of developing robust and generalizable artificial intelligence models from less-than-ideal data, such as noisy, limited, or entirely absent training data. Put formally, we seek to answer ”how can we create robust and generalizable artificial intelligence models from limited or entirely absent training data?”

The ZLaP method is designed to tackle this problem by leveraging the relationships between known class names and unlabeled images. ZLaP encodes both into a shared latent space using an encoder such as CLIP, allowing the propagation of labels based on similarity metrics (commonly implemented using a KNN). This approach enables the model to classify new data without the need for extensive labeled training datasets. One may imagine ZLaP as taking a classification problem and reformatting it into a geometric one by modeling classifications based on distances in a latent space. ZLaP utilizes a KNN search during label propagation, which is markedly absent of a training step.

Overall, we aim to perform classification on images using purely unlabeled data and class labels by re-implementing and expanding on ZLaP. The core ZLaP functionality and hyperparameter validation was placed in `zlap.py`, and the expansion of ZLaP on custom images/classes was placed into `main.py`. A user may run either script.

## Methodology

As mentioned, the term paper was divided into two distinct phases: firstly re-implementing ZLaP, and secondly verifying the authors' choice of hyperparameters and extending our implementation to accept arbitrary images and classes. We will summarize the core principles of ZLaP, then describe details of our experimental setup, implementation, hyperparameter validation, and evaluation.

### ZLaP Overview

ZLaP's core principle is label propagation (LP), which assigns class labels to previously-unlabeled data points. A small subset of points may begin with labels, and some metric of similarities (”communities” as named in literature). The key advantage of LP is its low requirement of a-priori information of both classes and graph structure.

In LP, ZLaP chooses to use a KNN to propagate labels from known to unknown items based on a geodesic measure of distance. A quite familiar geodesic is the straight line, which is a geodesic in euclidean space since it connects two points by the shortest path. In non-euclidean spaces, a geodesic is a generalization of a curve representing the shortest path between two points on a manifold.

### Experimental Setup

Because there is no expensive training step required for implementing ZLaP, we implemented the paper on my own machine. We conducted development on a laptop which does NOT have a dedicated GPU, so we did not implement GPU acceleration in ZLaP. To ensure my machine has a comparable environment to the one used by the authors, we have configured a `conda` environment in Python and configured the following packages along with some further dependencies.

- `torch`
- `scipy`
- `faiss`
- `numpy`

To install these requirements, we provide `requirements.txt`. Simply use `python3 -m pip install -r requirements.txt` to set up your machine.

FAISS (Douze et al. 2024) is a library designed to perform clustering and calculate similarity between dense vectors based on a metric of Euclidean distance, highest dot product, or cosine similarity. We utilized this library as a part of the label propagation, specifically in the KNN search step. This enabled fast and simpler processing since FAISS is largely considered to be one of the best options for computing vector similarity.

Regarding datasets of unlabeled images and class labels, the ZLaP authors provided pre-extracted features as `.npy` files, which consist of image/textual data already sent through a Vision-Language Model (VLM) such as CLIP which helps ensure reproducibility. We utilized these features and loaded them in numpy instead of manually running images through a VLM. These features are a part of the repo, and are loaded automatically. We used the `np.load(...)` function to load the following items:

- **Features and Targets:**
  - Training Features: `train_features`
  - Training Targets: `train_targets`
  - Validation Features: `val_features`
  - Validation Targets: `val_targets`
  - Test Features: `test_features`
  - Test Targets: `test_targets`
- **Classifiers:**
  - Text Classifier: `clf_text`
  - CUPL Text Classifier: `clf_cupl_text`
  - Training Image Classifier: `clf_image_train`
  - CUPL Train Classifier: `clf_cupl_image_train`
  - Val Classifier: `clf_image_val`
  - CUPL Val Classifier: `clf_cupl_image_val`
  - Test Image Classifier: `clf_image_test`
  - CUPL Test Classifier: `clf_cupl_image_test`

## Models

We utilized the CalTech101 dataset using `RN50-OpenAI` inside CLIP as an encoder for core ZLaP functionality. CLIP was again utilized as an encoder while expanding the ZLaP code to accept arbitrary images and class labels.

## Re-implementation

We first identified the major required functionality for re-implementing ZLaP. As mentioned, ZLaP has three distinct modes: `transductive` learning which provides predictions for examples in the dataset, `inductive` learning which operates on data unseen by the model, and finally `sparse-inductive` which sparsifies various matrices in the ZLaP process to enhance both performance and accuracy during inductive learning.

Notably, the authors published an implementation of ZLaP on their GitHub which was utilized to identify these three modes and key functionality they utilize, however we want to be clear that all implementation was done from scratch and has many notable distinguishing features and expansions (as covered later).

We first implemented utility functions for constructing a Laplacian from the KNN graph, searching using the FAISS library, getting data, and computing the normalized adjacency matrix. These functions are helpers as used in ZLaP.

We then utilized these helpers to implement transductive learning. This first requires separately performing text-to-text and image-to-text searches in the latent space using FAISS in the `create_separate_graph()` function. This is done simply by passing the embeddings into a function call to the FAISS library, which computes distances between embeddings. After the search we then utilize our first hyperparameter gamma as an exponential scaling factor for entries in the KNN which point to a class. Gamma is chosen to be a number greater than 1.0, so entries corresponding to a class can be weighted highly during label propagation. We identify these as values in the similarity matrix whose corresponding entry in the KNN has less than `num_classes` connections:

```
# Use FAISS for KNN search
k1=min(k, features.shape[0])
k2=min(k, num_classes)
knn_im2im, sim_im2im = faiss(feat, feat, k1)
knn_im2tx, sim_im2tx = faiss(clf, feat, k2)

# scaling KNN entries pointing to class
classes = k < num_class
sim[classes] = sim[classes] ** gamma
```

Notably, we perform these searches separately due to a known limitation of VLMs giving a large modality gap between text and image features. We then combine these separate im2im and im2text networks into one network using `combine_separate_knns()`.

We then transform this combined KNN into a Laplacian matrix utilized for label propagation. This is done via creating a sparse adjacency matrix and converting it to a Laplacian. In this context the Laplacian tells us how different an element in our network is from its neighbors, which is used for disambiguation of classes. The Laplacian is given by:

$$L = (I \quad W)$$

Where  is a tunable hyperparameter and $W$ is the symmetric adjacency matrix:

```
W = csr_matrix((sim_flat, (row_idx_rep_flat,
     knn_flat)), shape=(N, N))

# Make W symmetric since W is square matrix
W_hat = W + W.T
```

Finally, label propagation is performed. This is done via an iterative process which utilizes the previously-mentioned hyperparameter  and a one-hot vector $\hat{y}_c$ indicating the current class:

$$\hat{y}_c^{(t+1)} = \quad \hat{W}\hat{y}_c^{(t)} + (1 \quad )y_c$$

This iterative process can be simplified, and ends up being equivalent to solving a linear system $L\hat{y}_c = y_c$ to find the predicted labels $\hat{y}_c$ via the conjugate-gradient (CG) method. CG is an efficient method for solving systems involving large, sparse, positive-definite matrices (true for our Laplacian). For each possible class we construct a one-hot vector (all zeros except one entry indicating the class label) and solve the system via CG.

We finally take the result of the conjugate gradient solution for the i-th class and store it in a scores matrix, which accumulates the label propagation results across all classes. Each "score" gives a notion of confidence than a sample belongs to a certain class (given by a row).

```
scores = np.zeros((features.shape[0],
    num_classes)) # shape (N, num_classes)
    for i in range(num_classes):
        # solve for the ith class
        Y = np.zeros((laplacian.shape[0],))
        Y[i] = 1
        x = conj_gradsearch(laplacian, Y)

        # store results in col for class i
        scores[:, i] = x[num_classes:]
    return scores
```

This completes the design of transductive learning. We then moved onto implementing inductive learning, whose workings are largely similar to transductive learning. As such, we begin by creating separate graphs for image-to-image and image-to-text KNNs. We once again scale entries pointing to a class by gamma, and transform the KNN into a Laplacian.

Because this is an inductive (operating on new examples) step, we now perform a search on the KNN network in the `get_neighbors_for_inductive()` function. This consists of searching between the unlabeled features and test features using FAISS, and separately between the `clf` (class data) and test features to find the closest neighbors in the embedded space. Notably we choose the K value in the search via `min(K, num_classes)` or `min(K, unlabeled_features` in the case of very low numbers of possible classes. As comparable to transductive learning. we scale the similarity scores by gamma to help disambiguate classes.

```
# unlabeled/test, then unlabeled/clf
knn_im2im, sim_im2im = search_faiss(
    unlabeled_features, test_features, k=min
    (k, unlabeled_features.shape[0]))
knn_im2text, sim_im2text = search_faiss(clf,
     test_features, k=min(k, num_classes))

# remove entries < 0
sim_im2im = np.maximum(sim_im2im, 0)
sim_im2text = np.maximum(sim_im2text, 0)

# shift for im2im, scaling for im2text
knn_im2im += num_classes
sim_im2text = sim_im2text ** gamma
```

After we find the $k$ nearest neighbors, we solve the same linear system as during transductive learning, and populate the resulting scores matrix. This completes the design of inductive learning, as the largest difference is the added KNN search step.

The last mode of operation for ZLaP is the `sparse-inductive` mode, which conceptually follows the same steps as inductive learning but sparsifies the inverse of the Laplacian. This works because solving the prior-mentioned system $L\hat{y}_c = y_c$ is actually equivalent to solving $N \quad C$ systems $\hat{z}_j = L^{-1}e_j$ where $\hat{z}_j$ contains the estimated labels, and $e_j$ is a basis vector with the $j$th entry being $1$.

This sparsification is done via the following code, which takes the max element along each row and sets the rest to zero:

```
Linv_sparse = np.zeros_like(L_inv)
top = np.argmax(L_inv, axis=1, keepdims=True
    )
np.put_along_axis(Linv_sparse, top, np.
    take_along_axis(L_inv, top, axis=1),
    axis=1)
Linv_sparse = csr_matrix(Linv_sparse)
```

This ends up having significant benefits. We will cover later how this allows ZLaP to be massively more resilient to hyperparameter choice than the standard inductive mode, and it also gives ZLaP a massive performance boost due to the matrix sparsity. Finally, we receive accuracy benefits. The ZLaP authors detail that this accuracy increase likely comes from lower-probability predictions given for each class (after solving using CG) now being be set to zero.

## Expansion to Accept Arbitrary Images/Classes

As mentioned, the authors of ZLaP originally provided images and classes already put through CLIP, so there was no native support for datasets besides standard ones like Caltech101. As mentioned in our abstract, we propose a scenario in which a classification model is to be deployed in a retail environment to classify products in a store. If the store changes their catalogue often, they would typically need to re-train their classifier even if there's a minor catalogue change.

This paper's ZLaP expansion fixes this problem. We have provided `main.py` which allows a user to provide fully arbitrary images and potential class labels, and ZLaP will classify each image. The file contains an entirely-custom class vector, which the user may update at any time. We also accept a program argument `--img_path` which points to a folder of images the user wishes to classify. They may provide new images or entirely different class labels as they wish, and as ZLaP does not require a training step they may deploy the model within seconds.

To allow the use of this arbitrary class vector and custom images, we need to move both of these into the embedded space. We chose to utilize `RN50` as a backbone in CLIP.

We take both the custom classes and images and embed them using CLIP. The embedding step is done image-by-image and class-by-class, and the embeddings are added to separate lists for use in ZLaP. The embedding is done by first using the CLIP preprocessor, and feeding into the CLIP embedder:

```
def encode_img(img_path, model, preprocess,
    device='cpu', normalize=True):
    img = Image.open(img_path)
    img_input = preprocess(img).unsqueeze(0)
        .to(device)
    with torch.no_grad():
        image_features = model.encode_image(
            img_input)

    if normalize: image_features /=
        image_features.norm(dim=-1, keepdim=
        True)
    return image_features.cpu().numpy()

def encode_text(txt, model, preprocess,
    device='cpu', normalize=True):
    text = clip.tokenize([txt]).to(device)
    with torch.no_grad():
        text_features = model.encode_text(
            text)

    if normalize: text_features /=
        text_features.norm(dim=-1, keepdim=
        True)
    return text_features.cpu().numpy()
```

At this point, we have accepted arbitrary images and the user's custom class label vector, and encoded them. The next step is to feed the embeddings into our implementation of ZLaP, where we have chosen to utilize the `sparse-inductive` mode for higher performance and resiliance to hyperparameter choice as discussed earlier. The "unlabeled features" are best taken from an existing dataset (Caltech101 in this case), and we supply the user's custom data as the "test features". This promotes a diverse set of image feature data which can be used for label propagation. Notably, the unlabeled features DO NOT need to have the same classes as the custom dataset.

```
scores = do_sparse_inductive_lp(
    train_features, class_features,
    image_features, args.k, args.gamma, args
    .alpha)
```