SMAI [Spring 2025]

1 Problem 1

Download the Dataset and Source Code

1.1 Introduction

Principal Component Analysis (PCA) is a dimensionality reduction technique that finds an optimal set of basis vectors (eigenfaces in the case of images) that best represent the data. By selecting a limited number of principal components, we can project high-dimensional data into a lower-dimensional space, achieving compression. The original data can then be approximately reconstructed from the principal components.

In the context of face recognition, PCA is used to compute "eigenfaces"—the principal components derived from a dataset of face images. Each eigenface is an eigenvector of the covariance matrix of these images and represents a pattern that captures significant variations in facial features. By selecting a limited number of eigenfaces (those corresponding to the largest eigenvalues), you form an optimal, lower-dimensional basis for representing faces.

When a new face image is projected onto this eigenface subspace, it is described by a set of coefficients that capture its essential features. This process compresses the high-dimensional image data into a compact form, reducing storage needs and computational complexity. Moreover, by combining the selected eigenfaces weighted by these coefficients, an approximation of the original face can be reconstructed, retaining the key facial characteristics while discarding less important variations.

1.2 Dataset

Download the Dataset. It contains two subfolders, one for men's faces and one for women's faces. Each subfolder contains different subfolders for each person's faces.

If you are not able to fit the whole dataset into memory, then you can use a subset of the whole dataset as further processing.

If needed for a task or experiment, you can add your own images as well. Ensure the uploaded images are cropped faces only.

1.3 Source Code

1.3.1 PCA.ipynb

- It contains the code for finding eigen faces from the dataset. Finding eigenvalues from the covariance matrix is a time-consuming process, so if you have a large dataset or a large resolution image, then it will take more time.

- There are variables for IP, FRIEND IP, PORT, eigen face, mean face numpy array path and select the 'k' (represent top k eigenfaces) for your compression and reconstruction.
- To run the file :- python server.py
- Required library numpy, tqdm, cv2, ultralytics, threading

1.4 Setup

- **Phase - 1: Training of PCA** In this phase you will do PCA on the dataset that provided and get the all eigen faces and mean face. This will done by **PCA.ipynb**
- **Phase - 2: Visualization of Eigen faces and Reconstruction** In this phase you can visualize the eigen face and it's reconstucted images with it's cost. and find out the optimal top k eigen faces. This will be done using **Reconstruction_viz.py**
- **Phase - 3 : Video calling** This testing phase of your selected eigen faces. In this phase you will do video calling with your friend and make obersavations. This will be done by Server.py

1.5 Task

Your main task is to find out the way to get best quality in video calling under the following conditions : (Keep in mind below all conditions are independent of each other)

1. Normal setup: Do Phase-1 using the dataset given and do video call with friend (server.py). While video calling try to cover your half face by your hand or rotate your face or turn your face i.e. experiment with changing orientation or covering your face.
2. If you are male student then in Phase-1, only use the Female faces of the dataset for training the PCA, then test with you and your male friend. If you are female student then in Phase-1, only use the Male faces of the dataset for training the PCA, the test with you and your female friend.
3. During the Phase - 1 you are only allowed to use 500 images for training
4. During Phase - 1, you should only train using you and your friend's face data. Do video calling with same friend and note down the quality. Then do video call with a different friend (model is untrained on their face) and note down the video quality.

1.6 Submission Format

Screen recording of video calling under each conditions those mention above. Report of the ideas and thinking behind your approach for each condition and results and failure case of your approach. For whole report Max 2-3 pages are allowed. More than 3 pages report will not graded)

2 Problem

Dataset and Source Codes

2.1 Task

Word auto-completion is a widely used feature in modern text input systems, designed to enhance typing speed and accuracy by predicting and suggesting the next characters or words based on a given prefix. It is commonly found in search engines, messaging apps, code editors, and mobile keyboards. These systems rely on language models to generate predictions, ranging from simple dictionary lookups to advanced probabilistic models. Traditional approaches include n-gram models, which estimate the likelihood of a sequence of characters or words based on historical data, and more sophisticated deep learning-based models such as recurrent neural networks (RNNs) and transformers. In this task, you will implement a character-level n-gram language model to predict word completions based on an input prefix, integrating it into a terminal-based user interface for real-time text prediction.

2.2 N Gram Character Language Model

A classical language model is a statistical model that assigns probabilities to sequences of words or characters in a language. At its core, it tries to predict the likelihood of a word or character appearing given the context of previous words or characters.

The simplest form is the unigram model, which assumes all words/characters are independent of each other. It calculates the probability of each word based solely on its frequency in the training corpus:
$P(\text{word}) = count(\text{word})/total\_words$

This approach ignores context entirely, treating each word as if it were drawn randomly from a fixed distribution. N-gram models extend this by considering a short history of preceding words or characters. $P(t_k|t_{k-n+1}, ..., t_{k-1}) = count(t_{k-n+1}, ..., t_{k-1}, t_k)/count(t_{k-n+1}, ..., t_{k-1})$

Character-level language models operate on individual characters rather than words. Instead of modeling word sequences, they model character sequences. They can generate any word, even those not seen in training.

You have been given the class layout for `NGramCharacterModel` in `ngram.py`. Each function contains descriptions of its respective task. You need to implement a character level n-gram model which will train on a given corpus and for a given n. This model will then be called to predict word suggestions given a prefix.

2.3 User Interface

For evaluation, a terminal-based interface, `user_interface.py`, has been provided. In this interface, the user types a given text, and after each typed letter, the implemented model is called to provide completion suggestions. These suggestions are displayed above the text input.

- Use the `tab` key to cycle through the suggestions.
- Press `enter` to auto-complete the word using the selected suggestion.
- Use `esc` key to close the interface.

The text corpus for evaluation is provided in `text_content.txt`. To determine which paragraph to use, select the ith paragraph, where:

$$i = (\text{your roll number}) \mod 5 \tag{1}$$

To launch the interface, execute the following command:

```
python user_interface.py <path_to_corpus_folder>
```

2.4 Testing Metrics

The following metrics must be logged during testing:

1. Total number of letter keys typed.
2. Total number of `Tab` key presses.
3. Average number of letter keys typed per word i.e. for each word: $\frac{\text{Letter keys typed}}{\text{Letters in the word}}$
4. Average number of `Tab` key presses per word: $\frac{\text{Total tab key presses}}{\text{Total words}}$

These metrics need to be integrated into `user_interface.py` for analysis and performance assessment.

Note: All the required tasks have been marked with `TODO` tags in the provided codebase.

2.5 Analysis

The analysis report should contain two parts and should not be of more than 2 pages

1. **Results Table**: Log the metric 'average number of tab key presses per word' scores for each model vs each corpus
2. **Analysis of Results**: Answer the following questions in brief
   (a) **Corpus Size**: How does the corpus size and the content of the corpus affect the model's predictions? Run experiment using the three following corpus for each model:

i. General English Corpus
ii. Topic Specific Corpus
iii. part_i.txt from the previous folder, where

$$i = ((\text{your roll number}) \mod 5) * 2 + 1 \tag{2}$$

(b) **Model Type**: Does the amount of context provided (number of character tokens) while predicting affect the results? Run experiment using the following 4 values of n:
   i. 2
   ii. 3
   iii. number of your choice between 3 and 10
   iv. 10

(c) **Metrics**: Comment on the metric scores defined above and what they tell us about the performance of the model

(d) **Generalization**: Given the General English Corpus, take a 100 word paragraph of a topic of your choice and test on it. Comment on the results obtained.

2.6 Submission Format

Your final submission should have the following:

1. Modified code files
2. Report
3. 2 short screen recordings of you typing on the interface
   (a) Given paragraph, with the best performing model and corpus combination
   (b) Paragraph of your choice, on the General English Corpus, with the best performing model

2.7 Bonus

Additionally, you can try using implementing a character level RNN model to fulfill the above task. Resources for the same have been linked. Note that this is a bonus question, not mandatory. It is advised to complete the main question before attempting this part.

2.8 Resources and Additional Reading Material

1. Character Level Bigram Language Model
2. Character Level RNN