COMP0088
Introduction to Machine Learning
Lab Assignment 7

Maximilian Mozes Matthew Caldwell

November 20, 2021

Task Summary

Week 7 Notes

In this week's exercise, you will implement clustering methods in Python. The exercise consist of two parts. In the first one, you are asked to implement the *k*-means clustering method as introduced in the lecture. Your implementation will be used for the task of image colour quantisation. In the second part, you will implement principal component analysis (PCA) using two methods as discussed in the lecture: eigendecomposition and singular value decomposition.

The assignment script is `week_7.py`, generating output file `week_7.pdf`. There are some potentially-useful command line options, which you can list using:

```
$ python week_7.py -h
```

Example output from a completed assignment is shown in Fig. 1. Plotting functionality is provided, so your output should look similar, though the contents may vary quite a bit with different configurations.

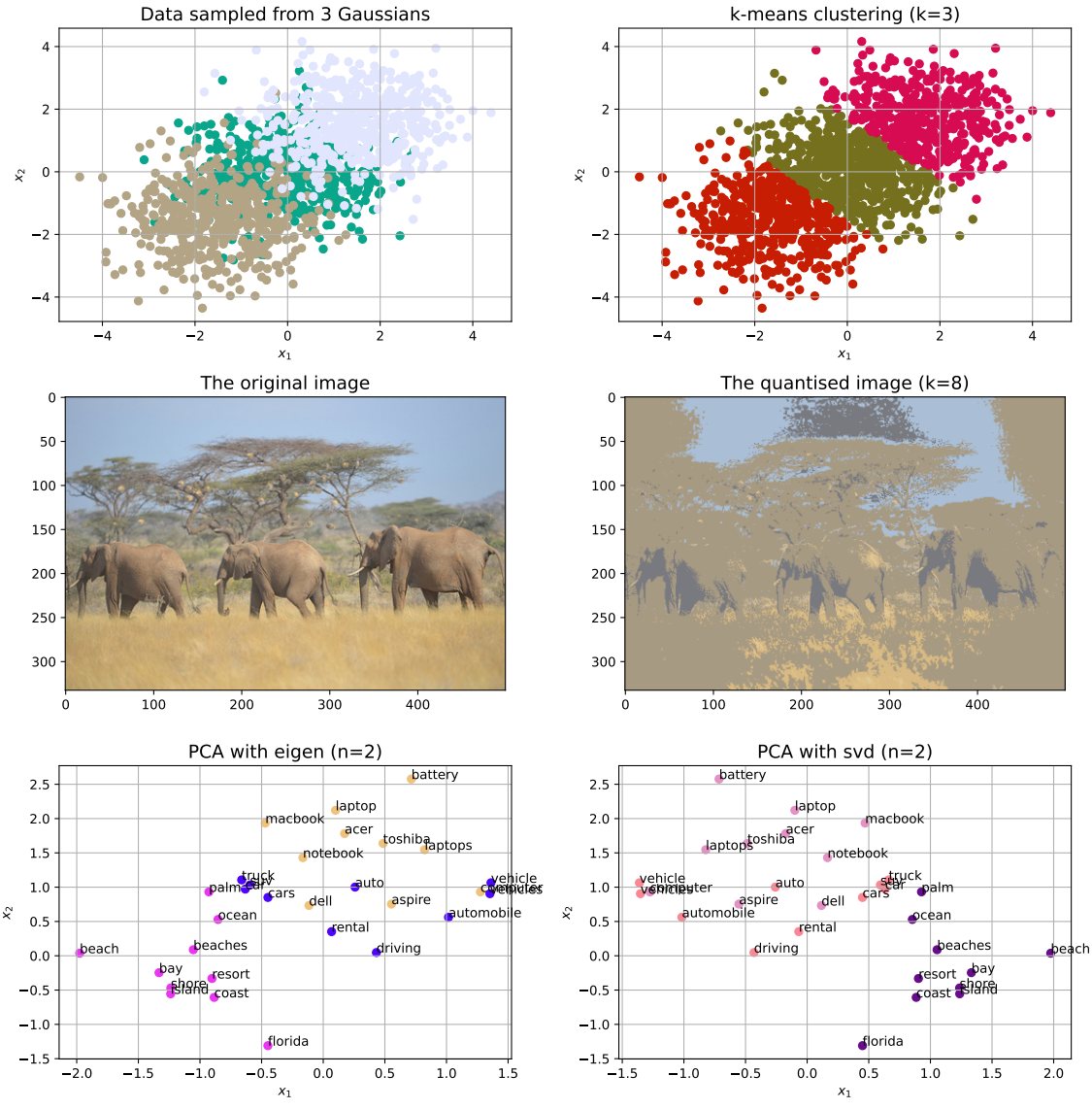

Figure 1: Example output from the `week_7.py` script

1 k-Means Clustering

In this section you will implement the k-means clustering algorithm as introduced in the lecture. The algorithm receives a set $\boldsymbol{X} \in \mathbb{R}^{N \times D}$ of N samples (each with dimension D) and a given number k of clusters into which to categorise the data. To do this, the algorithm learns a set $\boldsymbol{C} = \{\boldsymbol{c}_1, \ldots, \boldsymbol{c}_k\}$ of k cluster centroids and then assigns the N samples in the datasets to the cluster whose centroid lies closest to each sample. We denote the subsets $\boldsymbol{S}_i \subseteq \boldsymbol{X},\ i \in \{1, \ldots, k\}$ as the samples in $\boldsymbol{X}$ currently assigned to cluster i.

1.1 Implement the within cluster sum of squares (WCSS)

Performance of k-means clustering can be measured with the within cluster sum of squares (WCSS). The WCSS is defined by

$$WCSS = \frac{1}{k} \sum_{i=1}^{k} \frac{1}{|\boldsymbol{S}_i|} \sum_{\boldsymbol{x} \in \boldsymbol{S}_i} ||\boldsymbol{x} - \boldsymbol{c}_i||^2$$

Implement the following function in `week_7.py`.

```
def compute_wcss(data, centroids, assignments):
    """
    Compute the within cluster sum of squares (WCSS).

    # Arguments:
        data: input data of shape (N, D).
        centroids: array of shape (k, D) containing the centroids for
            each cluster.
        assignments: array of shape (N) containing the assigned clusters
            for each row in data.

    # Returns:
        wcss: the WCSS across all clusters.
    """
    # TODO: implement the within cluster sum of squares
    return None
```

The return value here should be a scalar (i.e., the WCSS). You can make use of the NumPy function linalg.norm to compute Euclidean distances between vectors.

1.2 Implement k-means clustering

As discussed in the lecture, k-means clustering comes with different methods for initialising the cluster centroids. In this exercise, you are asked to implement the algorithm with the following two initialisation options (these can be specified with the `--k_means_init` argument).

Forgy initialisation. This initialisation method randomly selects k samples from the data to initialise the cluster centroids.

Naive initialisation. This initialisation method samples cluster centroids from a uniform distribution in the range of the minimum and maximum values observed across the dataset.

Implement the following function in `week_7.py`:

```python
def k_means_clustering(data, rng, k=3, init="forgy", max_iter=500):
    """
    Method for implementing k-means clustering.

    # Arguments:
        data: input data of shape (N, D).
        rng: an instance of numpy.random.Generator.
        k: the number k for the algorithm.
        init: the initialisation method (one of "forgy", "naive").
        max_iter: the maximum number of iterations to run the algorithm.

    # Returns:
        assignments: list of shape (N) of cluster assignments
            for each data point.
        n_iter: the number of iterations needed by the algorithm.
        centroids: the centroids after convergence.
    """
    # TODO: implement the k-means clustering algorithm
    return None, None, None
```

Your implemented algorithm will be tested on synthetic data in the script. The data are generated from a set of multivariate Gaussian distributions. The first set of data points will be sampled from a standard normal distribution. The subsequent points are sampled from mean-shifted standard normals. You can specify the mean shift values (and thereby the number of additional Gaussians to sample from) using the `--mean_shifts` argument. This argument expects a string of space-separated floating point numbers, each number representing a Gaussian distribution to draw samples from (and its corresponding mean shift value). The `--n_samples` argument specifies the number of data points to sample from for each Gaussian. For example, running the script with

```
$ python week_7_working.py --mean_shifts -3 -1.5 1.5 3 --n_samples 50
```

will sample 50 points each from five Gaussian distributions (the standard normal one and four additional, mean-shifted ones where the means are shifted by -3, -1.5, 1.5, and 3).

1.3 Image colour quantisation with *k*-means

In this subtask, we take your implementation to a practice test. Specifically, your implemented algorithm will be applied to the task of image colour quantisation. That is, for a given input image and a specified number of clusters k (these can be specified with the `--n_clusters_q` argument), k-means clusters the individual image pixels based on their RGB-values. For each run, your implementation will be tested on one of five images (randomly selected) from the Visual Genome dataset (**?**). The images are stored in the `week_7_data/img` directory.

To do this, implement the following function in `week_7.py`:

```python
def image_colour_quantisation(img, k, max_iter, rng):
    """
    Perform image colour quantisation on the provided input image.

    # Arguments:
```

```
        img: an array of shape (N, M, C) with N x M pixels and
            C colour channels per pixel.
        k: the number of clusters k to use for k-means.
        max_iter: the maximum number of iterations to run the
            clustering algorithm for.
        rng: an instance of numpy.random.Generator.

    # Returns:
        img_seq: an array of shape (N * M, C) representing the
            sequentialised image after the colour channels C have
            been assigned to the k clusters using the k-means algorithm.
    """
    # TODO: implement image colour quantisation using the k-means algorithm.
    return None
```

The `week_7.py` test driver will plot four subplots for this section. The first row of the output file visualises the data sampled from the Gaussians and the corresponding result of applying k-means to the data. The second row visualises the result of image colour quantisaton, by showing both the original input image and the quantised one.

2 Principal Component Analysis

In this section you will implement principal component analysis (PCA) as introduced in the lecture, based on two methods: eigendecomposition and singular value decomposition. In order to do this, we use word embeddings as the data.

2.1 Enter word embeddings

Word embeddings are numerical representations of words in a collection of textual documents that encode semantic relationships between individual words based on the context in which they appear. That is, words that are semantically related are located close to each other in a learned embedding space. Word embeddings are trained in an unsupervised fashion, and popular methods to achieve this are *word2vec* (**??**) and *GloVe* (**?**).

We will not implement our own word embedding models here, but instead utilise a subset of pre-trained *GloVe* embeddings.[1]

The subset of word embeddings used in this assignment can be found in the `embeds.txt` file in the `week_7_data` directory. The function

```
def load_word_embeddings(path):
```

takes care of loading the word embeddings and does not have to be implemented.

The first subtask is to implement a function that retrieves the nearest neighbours for a given word in the pre-trained embedding space.

Implement the following function in `week_7.py`:

```
def get_nearest_neighbours(query, embeddings, labels, n=10):
    """
    Computes nearest neighbours (in terms of Cosine similarity) for
        a given input query in embedding space.

    # Arguments:
        query: the query word.
        embeddings: list of word embeddings of shape (N, D).
        labels: list of N labels for the word embeddings.
        n: number of nearest neighbours to return.

    # Returns:
        nearest: list of words representing the n nearest neighbours
            for query (INCLUDING query itself).
    """
    assert query in labels, print(f"No embedding for word {query}")
    # TODO: find the nearest neighbours for word 'query' in 'embeddings'
    return None
```

The distances between the embedding for `query` and all other embeddings should be computed using the Cosine distance. You can use the SciPy function spatial.distance.cosine to compute these distances.

[1] Sourced from https://nlp.stanford.edu/projects/glove/.

2.2 PCA with eigenvalue decomposition

Your next task will be to implement PCA with eigenvalue decomposition.

Implement the following function in `week_7.py`:

```
def pca_eigen(data, n):
    """
    Computes PCA by eigendecomposition.

    # Arguments:
        data: input data of shape (N, D).
        n: number of principal components to retain.

    # Returns:
        The transformed data, truncated at n along the
            second dimension shape (N, n).
    """
    # TODO: transform the input data using PCA and
    # return only the n principal components
    return None
```

In order to do this, you will have to compute eigenvectors and eigenvalues of the covariance matrix. You can use the linalg.eig NumPy function for this.

2.3 PCA with singular value decomposition

Finally, you will implement PCA with singular value decomposition.

Implement the following function in `week_7.py`:

```
def pca_svd(data, n):
    """
    Computes PCA by singular value decomposition.

    # Arguments:
        data: input data of shape (N, D).
        n: number of principal components to retain.

    # Returns:
        The transformed data, truncated at n along the
            second dimension (shape (N, n)).
    """
    # TODO: transform the input data using PCA and
    # return only the n principal components
    return None
```

In order to do this, you will have to compute a singular value decomposition of the data matrix. You can use the linalg.svd NumPy function for this.

The `week_7.py` test driver will plot two subplots for this section in the third row of the output file, in which the word embeddings are visualised. Specifically, two plots will be produced (one

for eigendecomposition and one for singular value decomposition) that show the embeddings for the words *car*, *beach* and *laptop*, along with their nine nearest neighbours each.