EE4-68 Pattern Recognition: Coursework 2

Alvaro Robledo Vega
Imperial College London
CID: 01076364

Mariam Sarfraz
Imperial College London
CID: 01069329

1. Summary

On this report, we analyse a popular dataset (CUHK03) consisting of 14096 images of 1360 pedestrians. The images are captured from 2 cameras, which results in substantial differences in the images captured due to varying angles, image resolution, lighting, etc. This is reflected as large distances between the feature vectors of the same pedestrian.

While we focused on standard facial recognition in the previous coursework, we now face the challenge of recognizing pedestrians despite each one having a set of *dissimilar* feature vectors.

We propose a baseline approach and then try to improve on it by using different distance metrics, metric learning and non-linear feature transformations (kernels).

2. Problem Formulation

The problem we want to solve can be visualized in Figure 1. The aim of this coursework is to minimize the distance between features of the same labels and maximize the distance between features of different labels, where each label represents an individual pedestrian.

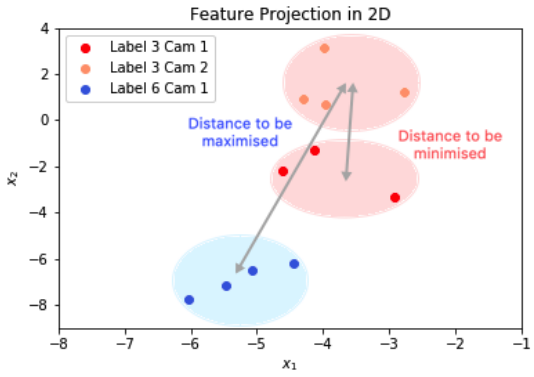

Figure 1: Feature Projection in 2D Space (using PCA)

The first thing to note is that the *distance* seen in Figure 1 can be interpreted and measured in numerous ways. Before trying to minimize or maximize these distances, it is vital to understand *how* we want to measure the distances between the data points.

A distance metric [1] is a way of measuring the distance/ similarity between two features. It is worth noting that similarity and distance represent the same concept as a large distance implies little similarity between objects. The aim of this coursework is to find an optimal metric, which will correctly classify feature vectors using k-Nearest Neighbours.

As standard practice, we initiate our search for the *right* metric by using the Euclidean Distance as our baseline. We will then evaluate the performance of different distance metrics and compare them to the baseline results. Lastly, we will explore the concept of Distance Metric Learning [1], a technique used to automatically construct optimal distance metrics, and Kernelization [1], an approach for mapping data to a higher dimensional feature space using non-linear transformation.

This formulated problem is neither a classification (supervised) nor a clustering (unsupervised) problem. We have images of the same pedestrian i.e. two views of the same person, which we know should be similar. This will be used as a 'soft' constraint for our learning algorithm (further details in sub-section 4.3). If the identity for each pedestrian was known, it would be a classification problem while if no labels were given, it would be a clustering problem.

3. Baseline Approach & Evaluation

3.1. k-Nearest Neighbours (kNN)

Our main baseline approach is kNN on the provided set of features. This algorithm simply looks at the *euclidean* distance between the query image and each gallery image, and returns the k closest neighbors (after deletion of those representing the same person with the same camera) [1].

$$d_{euc}(x, y) = ||x - y||$$

These k images make up our *ranklist*, on which we measure accuracy by determining whether or not the true label is actually included in any of the *ranklist* components. The average accuracy for each rank is calculated by:

$$Accuracy\ (\%) = \frac{Correctly\ Classified\ Data\ Points}{Total\ Number\ of\ Data\ Points} \times 100$$

Figure 2 shows two sample *rank-10* lists produced by our baseline (kNN) algorithm for two different query images. In both cases at least 1 of the 10 images contains the correct label, so we consider them as correctly classified.

By following the above methodology, we evaluate the performance of the algorithm for different values of k. This is depicted in Figure 3, where we can see that accuracy is a monotonically increasing function of k, as expected, since, intuitively, the bigger the ranklist, the higher the chances that it includes a correct match.

Additionally, if we look at the truncated mAP (mean average precision) [2], we can see how it stabilises around the 0.48 level as k increases. This is the expected behaviour,

Figure 2: Two randomly selected query images (left) with their respective *rank-10* lists (right), showing correct and incorrect matches

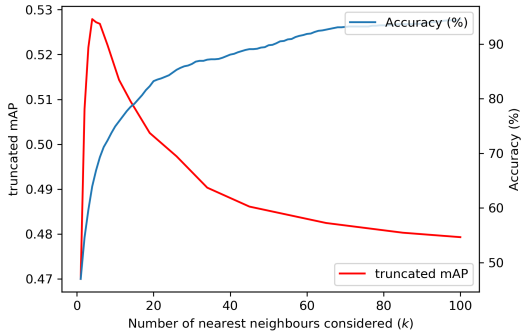

Figure 3: kNN accuracy & truncated mAP vs k

since mAP can be simply explained as the average of the maximum precision at the different recall levels [2], and the higher the k, the lower the impact of a change by an additional recall level. The actual mAP for the entire set is 0.45.

Euclidean distance is known for being quite sensitive to outliers [3], so we will attempt to improve performance by using different distance metrics. Additionally, it should also be possible to modify the given features by using kernels to achieve a better performance. We will explore a selected number of different methods in the following sections of this report.

3.2. k-Means Clustering

As mentioned in Section 2, this is neither completely a classification (supervised) problem, nor a clustering (unsupervised) problem. Therefore, it is interesting to also consider *k-means*, the most classic clustering algorithm, as an additional baseline.

This algorithm first finds structures in the given gallery dataset, dividing the data into k different clusters. Then, for a given query image, it checks which of the k cluster centers is the closest, and then performs kNN considering only the gallery features from that particular cluster.

This algorithm could, in theory, outperform kNN in those

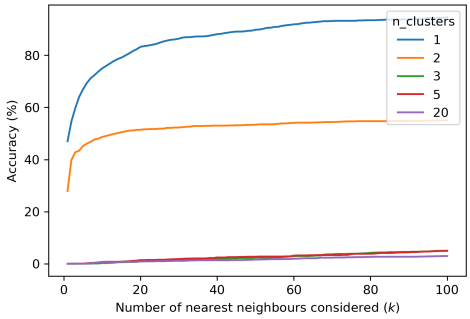

Figure 4: k-means accuracy for different values of k. Each colour represents a different number of clusters

cases when the query image is on the boundary between 2 clusters. However, it will also perform worse as the number of clusters increases, since we start to miss information from the nearby clusters because the cluster size gets smaller and we start considering less data points in the subsequent kNN (even these data points are supposed to be some of the most relevant ones, they will not be *all* of the relevant ones).

However, for our particular case, as depicted in Figure 4, the second fact ends up outperforming the first, and accuracy drops drastically as soon as we start using 3 or more clusters.

This confirms that simple kNN should be our main baseline for the remaining of this report.

4. Improved Approaches

While Euclidean Distance Metric is one way to calculate the distances between data points, there are many other distance metrics to choose from. In this section, a selection of the most commonly used metrics were chosen for evaluation. Cosine and Chi-Square distance metrics are implemented on our test and gallery feature vectors and their performance is then compared to our baseline metric performance.

We then explore *Distance Metric Learning*: the learning of a transformation matrix which rotates the space and rescales the dimensions, to solve our formulated problem. The Mahalanobis Metric is implemented and evaluated.

Finally, we will also explore non-linear feature transformations by using several different *kernels*.

4.1. Cosine Distance Metric

The cosine metric measures the cosine of the angle between two vectors. The distance between two data points, x and y, can be computed using:

$$d_{cos}(x, y) = \frac{x^T y}{||x||_2 ||y||_2}$$

While euclidean distance is used to measure the difference in absolute values of two data points, cosine looks at the angular difference between them. As seen in the equation, the magnitudes of both points are normalized.

The in-built function *Nearest Neighbours* [4] from the python library *scikit-learn* is used to implement cosine metric with a kNN setting. The k-nearest neighbours, from the gallery vectors, for each query vector are extracted and average accuracies are then calculated.

The results, for varying ranks, are summarised in Figure 5. Interestingly enough, cosine distance gives us slightly better results than euclidean. This mild improvement of around 0.3% is not sufficient to conclude that the cosine metric is *better suited* to solve our problem. However, it implies that measurement of orientation is as significant, if not more, than the magnitude of data points.

4.2. Chi Square Distance Metric

One of the key problems with the Euclidean Metric is that it is highly sensitive to outliers [3]. We attempt to tackle this problem by using the Chi-Square Metric [1]. Each dimension of the data points is normalized hence, is resistant to noisy features that can potentially deteriorate our results. Chi-Square Distance is calculated using:

$$d_{chi}(x, y) = \frac{1}{2} \sum_i \frac{(x_i - y_i)^2}{x_i + y_i}$$

where i represents the number of dimensions.

Chi-Square distance between each query vector and all gallery images is computed and again, a kNN setting is used. The results can be seen in Figure 5. The performance is seen to be worse than both euclidean and cosine metrics. This is due to the fact that Chi-Square metric accords high weightage to features with low frequency and conversely, nullifies the effect of frequent features. Despite normalizing outliers, Chi-Square is sensitive to the frequencies of features and thus this fluctuation can justify the results obtained [5].

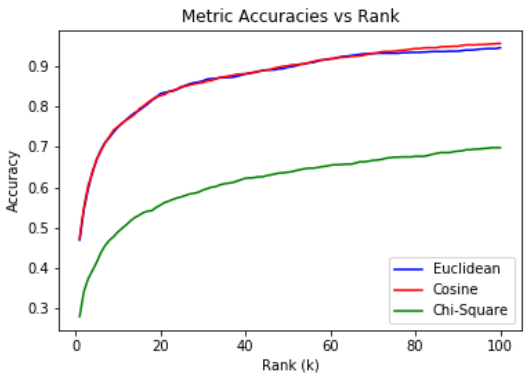

Figure 5: Metric Accuracy Plots for varying Ranks

The time taken to calculate distances using the three metrics explored so far, followed by kNN classification can be seen in Table 1. We note that all metrics take approximately the same time. The minor variations are due to different code implementations of the metrics, i.e. Cosine and Euclidean distances are calculated using in-built scikit-learn functions whereas a custom function was used for Chi-Square.

	Time Taken (s)
Euclidean	128.0
Cosine	102.0
Chi-Square	99.0

Table 1: Time Taken for Different Ranks & Distance Metrics

None of the explored metrics so far *learnt* from the training set to *transform* the query and gallery set to a space well suited to solve our formulated problem. In the next subsection, we will address this challenge.

4.3. Mahalanobis Distance Metric Learning

As opposed to the distance metrics explored above, where only the *way* of measuring distances was changed, we now focus on distance metric learning. Mahalanobis Distance Metric is defined as:

$$d_{md}(x, y) = (x - y)^T A (x - y)$$

where A is a symmetric, positive definite matrix (see Appendix A for proof), which is *learnt* using the training dataset provided. This matrix rotates the space and re-scales the dimensions of points x and y. In our case, it is responsible for transforming the query and gallery images to a space where the distance between features of the same label are brought closer together, while keeping the differently labelled features further apart. The distribution of points is effectively made circular so that each dimension equally contributes to the distance, making the metric extremely robust to outliers [1].

Two approaches for learning the matrix A and consequently, obtaining an accuracy for Mahalanobis Metric will be evaluated.

Original Covariance Based Approach

The matrix A is learnt by calculating the inverse of the covariance matrix of the training set. A is a measure of the variance of dimensions and the covariances between different dimensions. It signifies how a change in one dimension affects another one. In general, this matrix estimates the degree of linear dependency between different features.

On calculating A using our training set, we observe that the variances are exceedingly higher than the covariances for our features. To calculate the Mahalanobis Distances between each query feature vector with all gallery feature vectors, we use an in-built function *cdist* imported from a Python Library called *SciPy*. The result is a distance matrix of size $(N_{test}, N_{gallery})$, where $N_{test} = 1400$ and $N_{gallery} = 5328$. Each ijth element of the matrix is the Mahalanobis distance between query vector i and gallery vector j. Computation of this matrix, however, is unfeasible as the time complexity is $O(N_{test}\ D\ N_{gallery})$, where D is

the dimension of our feature vectors i.e. 2048. This problem is solved by using PCA (Principal Component Analysis) to reduce the dimensions of our feature vectors. A PCA analysis (refer to Appendix B for more details) showcased that 93% of the data variance was captured by the top 100 dimensions (with the largest eigenvalues). We therefore reduce our dimensions from 2048 to 100.

	Accuracy (%)		
Time Taken after PCA (s)	**Rank 1**	**Rank 5**	**Rank 10**
75.0	41.8	62.2	70.3

Table 2: Results for Mahalanobis Metric Learning using Inverse Covariance Matrix

The results, as seen in Table 2, show that Mahalanobis metric performs worse than our baseline model. This is due to the fact that the correlation between dimensions is extremely low in magnitude. A being a diagonal matrix, only re-weights and re-scales the dimensions, without rotating the space. This re-scaling may be resulting in increased distance in similarly label vectors and decreased distance in differently labeled vectors.

Optimized Approach

Rather than using the inverse covariance matrix of the training set, we will now proceed to learn A with the aim to bring the vectors of the same label together, while keeping vectors of different labels apart.

For this purpose, we use the Python package called *metric learn*. This package comprises of several metric learning algorithms, out of which we will use *Large Margin Nearest Neighbour (LMNN)*. LMNN learns a Mahalanobis distance metric in the kNN classification setting [6]. It aims to learn the metric in a way to keep k-nearest neighbours in the same class, while keeping vectors from different classes separated by a large margin. This complies perfectly with our desired solution.

We first fit the training vectors and labels to our model. We then transform the query and gallery vectors onto our transformed space. On learning A, we observe that it is an identity matrix! Intuitively, if A is an identity matrix, the Mahalanobis Distance reduces to Euclidean Distance. On calculating the accuracies for rank 1, 5 and 10, we obtain the exact same results as the ones achieved for our baseline model. This implies that the Euclidean Distance in a kNN classification setting gives us optimal results. While the inverse covariance matrix gave us extremely low covariances between dimensions, the optimized approach indicates that all dimensions are, in fact, uncorrelated.

To confirm these results, we try to optimize the matrix A by defining an objective function and a set of constraints. The constraints will be on the distances between the training data points, while the objective function will define the loss function that we aim to minimize in general. Our chosen objective function is as follows:

$$min \quad ||A||_F^2$$

This is the Squared Frobenius Norm of matrix A. We aim to minimize the squared values of all elements in the matrix as this avoids overfitting and ensures that the learning is *generalized*.

We formulate the following two constraints:

$$c(A) = \sum_{(x_i,x_j)\in S} d_A(x_i, x_j) \leq 1 \quad (1)$$

$$c(B) = d_A(x_i, x_k) - d_A(x_i, x_j) \geq 1, \quad \forall (i, j, k) \in R \quad (2)$$

Constraint (1) ensures that the sum of all vectors belonging to the same label must not be greater than 1. Constraint (2) ensures that there must be a marginal distance of 1 between similarly labelled vector distances and differently labelled ones.

We use an in-built function named *optimize*, which is part of the Python package *SciPy* [7]. This function takes the objective function and set of constraints as inputs, along with the optimization algorithm we would like to use. Gradient Descent is chosen as our optimization algorithm as it guarantees convergence to a minimum. Optimized matrix A is then found to be equal to an identity matrix, thus confirming that the optimal metric is the Euclidean Metric.

4.4. Kernels: Non Linear Transformation of Data

Kernel methods make use of the so-called *kernel functions*, which allow to operate in a *high-dimensional*, implicit feature space by simply computing the inner products between all pairs of data points instead of computing the actual coordinates of the data in that space, which would be much more computationally intensive [8]. This approach is also referred as the *kernel trick*.

We have tested several well-known kernel functions, such as the *polynomial*, *gaussian* and *sigmoid* kernels.

	Accuracy (%)		
	Rank 1	**Rank 5**	**Rank 10**
Poly. deg. 2	47.1	67.3	75.1
Gaussian	47.2	67.5	74.9
Sigmoid	47.6	67.4	75.1

Table 3: Accuracies for Different Ranks & Kernels

They all perform in a very similar way in terms of accuracy, even improving our kNN baseline by a few decimal points. However, once again, none of them manage to make a significant improvement over our baseline model. This is probably due to the initial features having been trained on the training set, which implies that there are not any more optimal results which can be achieved other than the ones we already have.

Appendices

A. Appendix

Matrix A must be symmetric and positive definite. If A is not positive definite, the distance calculated using A, d_A could be negative.

$$d_A(x, y) = (x - y)^T A(x - y)$$

$$d_A(x, y) = v^T A v$$

$$d_A(x, y) = \lambda v^T v = \lambda < 0$$

where v is an eigenvector corresponding to a negative eigenvalue λ of A.

B. Appendix

We note that almost all variance is captured using 100 PCA bases.

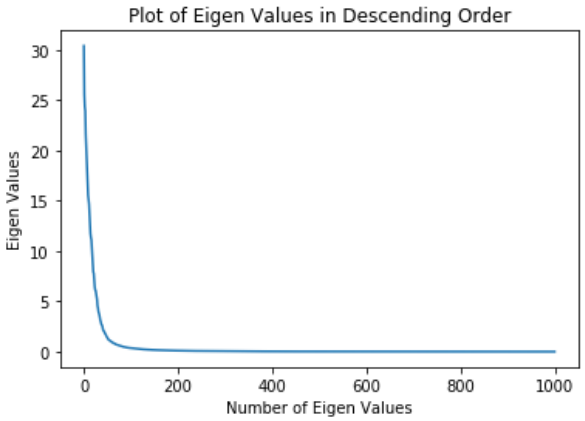

Figure 6: Accuracy for Varying PCA Bases

Accuracy using kNN for varying PCA Bases to ensure that results are not affected drastically by using PCA.

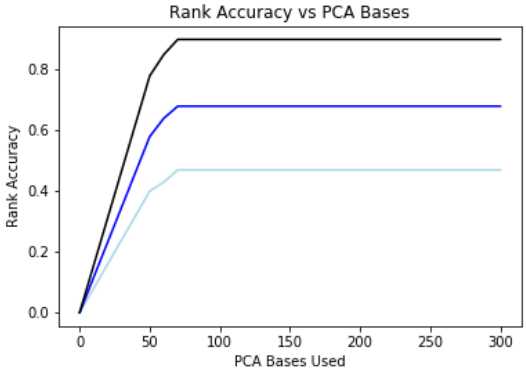

Figure 7: Accuracy for Varying PCA Bases

C. Appendix

Find the source code in a zip file in the following link `https://www.dropbox.com/s/ny8201jigr6r4vv/ar5115_ms5715.zip?dl=0`.

In the zip file, you will find a Jupyter Notebook named *CW2.ipynb*. The notebook contains all the relevant code written and executed for this coursework.

A folder in the directory, named *PR_Data*, is required which must have the following two files:

1. *feature_data.json*
2. *cuhk03_new_protocol_config_labeled.mat*

As we use a variety of in-built python packages, the following dependencies need to installed:

1. *scipy*
2. *sklearn*
3. *matplotlib*
4. *metric_learn*
5. *pandas*
6. *collections*
7. *pylab*

References

[1] Krystian Mikolajczyk. *EE4-68 Pattern Recognition - Lecture Notes*. 2018.

[2] Jonathan Hui. *mAP (mean Average Precision) for Object Detection*. Mar. 2018. URL: `https://medium.com/@jonathan_hui/map-mean-average-precision-for-object-detection-45c121a31173`.

[3] Michael E. Houle et al. "Can Shared-Neighbor Distances Defeat the Curse of Dimensionality?" In: *Scientific and Statistical Database Management*. Ed. by Michael Gertz and Bertram Ludäscher. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 482–500. ISBN: 978-3-642-13818-8.

[4] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[5] McCune Grace. *Distance Measures*.

[6] Wikipedia contributors. *Large margin nearest neighbor — Wikipedia, The Free Encyclopedia*. [Online; accessed 14-December-2018]. 2018. URL: `https://en.wikipedia.org/w/index.php?title=Large_margin_nearest_neighbor&oldid=869579376`.

[7] Eric Jones, Travis Oliphant, Pearu Peterson, et al. *SciPy: Open source scientific tools for Python*. [Online; accessed ¡today¿]. 2001–. URL: `http://www.scipy.org/`.

[8] Ping Li. “A Comparison Study of Nonlinear Kernels”. In: *arXiv e-prints*, arXiv:1603.06541 (Mar. 2016), arXiv:1603.06541. arXiv: 1603 . 06541 [stat.ML].