Unsupervised Learning

IN3050V22 — Assignment 3

Martin Mihle Nygaard <martimn@ifi.uio.no>

Readme

This one was pretty straight forward, as the guts of the code is spelled out in the Marsland book. I spent about 10% of the time making the code work, 10% cleaning it up, and 80% making the plots pretty*. Jupyter Notebooks still feel like bloat to me, so this is all Python + PDF. Hopefully this report answers the all the assignment questions and RFCs.

The attached scripts `pca.py` and `kmeans.py` should reproduce the figures, and pass the test cases. I quote some of the code here, but it's not meant to run standalone (lacking proper imports and such), only to aid your assessment. I have also not commented on my plotting code.

Principal Component Analysis (PCA)

What is the variance?

The *variance* is a measure of how spread out the data is; how far apart each data point is. This is calculated by the average square deviation from the mean μ, i.e. $\sum_{i=1}^{N}(x_i-\mu)^2/N$, with N as the number of data points. There is some nuance here, though. What I just described is the *population* variance σ^2, and is correct if our data is finite and captures the entire population (Devore and Berk 2012, p. 35).

I would imagine it is usually the case in unsupervised learning applications that our data is a just a sample from greater population, that we assume is somewhat randomly distributed. Then we would prefer to use the *sample variance* $s^2=\sum_{i=1}^{n}(x_i-\bar{x})^2/(n-1)$, where n is the number of samples, and $\bar{x}$ is their mean. The divisor in this formula, $n-1$, is our 'degrees of freedom' (ibid., pp. 34, 35).

In *Numpy* the variance is by default calculated with zero 'delta degrees of freedom' (DDOF), a divisor of $N-0$, that is (*Numpy Reference* 2022, `numpy.var`). In Python terms:

```
mean = lambda x: x.sum() / (len(x) - ddof)
var  = lambda x: mean(abs(x-mean(x))**2))
```

What is the covariance?

The *covariance* is a measure of how strongly related, or dependent, two (random) variables are to each other. It is defined as (Marsland 2015, p. 32; Devore and Berk 2012, p. 247)†:

$$\mathrm{Cov}(X,Y)=E\big[(X-\mu_X)(Y-\mu_Y)\big]$$

The sign of the covariance is indicative of the variables relationship: if $(X-\mu_X)$ and $(Y-\mu_Y)$ tends in opposite directions, there is a negative relationship; if they share

*Essentially emulating Seaborn, I belatedly realised.

†I suspect there is a typo in Marsland, as he suggest $\mathrm{Cov}(X,Y)=E\big[(X-\mu_X)\big]\,E\big[(Y-\mu_Y)\big]$, in contradiction to how he uses it in the covariance matrix on the next page, where he uses Devore and Berks definition instead.

sign, the covariance and relationship will be positive. The magnitude of $\text{Cov}(X, Y)$ is difficult to interpret unless the variables are normalized. Also, notice that

$$\text{Cov}(X, X) = E\Big[(X - \mu_X)^2\Big] = \text{Var}(X)\,.$$

How do we compute the covariance matrix?

If we let $\mathbf{X}$ be a N-dimentional vector of data points such that $\mathbf{X} = [x_1, x_2, \ldots, x_N]^T$, then the elements of the *covariance matrix* $\text{Cov}(\mathbf{X})_{ij}$ is simply the covariances of each pair points, x_i and x_j. You can also simplify this with some matrix operations, where $E\{\cdot\}$ indicates element wise expected value (Marsland 2015, p. 33; Devore and Berk 2012, p. 711):

$$\begin{aligned}
\text{Cov}(\mathbf{X}) &= \begin{bmatrix} \text{Cov}(x_1, x_1) & \text{Cov}(x_1, x_2) & \ldots & \text{Cov}(x_1, x_N) \\ \text{Cov}(x_2, x_1) & \text{Cov}(x_2, x_2) & \ldots & \text{Cov}(x_1, x_N) \\ \vdots & \vdots & \ddots & \vdots \\ \text{Cov}(x_N, x_1) & \text{Cov}(x_N, x_2) & \ldots & \text{Cov}(x_N, x_N) \end{bmatrix} \\
&= \begin{bmatrix} E\big[(x_1 - \mu_1)\,(x_1 - \mu_1)\big] & \ldots & E\big[(x_1 - \mu_1)\,(x_N - \mu_N)\big] \\ \vdots & \ddots & \vdots \\ E\big[(x_N - \mu_N)\,(x_1 - \mu_1)\big] & \ldots & E\big[(x_N - \mu_N)\,(x_N - \mu_N)\big] \end{bmatrix} \\
&= E\left\{ \begin{bmatrix} (x_1 - \mu_1) \\ (x_2 - \mu_2) \\ \vdots \\ (x_N - \mu_N) \end{bmatrix} \begin{bmatrix} (x_1 - \mu_1) & (x_2 - \mu_2) & \ldots & (x_N - \mu_N) \end{bmatrix} \right\} \\
&= E\Big\{ (\mathbf{X} - \mu_\mathbf{X})\,(\mathbf{X} - \mu_\mathbf{X})^T \Big\}
\end{aligned}$$

The covariance definition is somewhat muddied when we again consider our data as samples from a greater population: the expected value, $E[\cdot]$, is not simply the arithmetic mean. We get a better guess of the covariance with a DDOF of 1; the *sample covariance* formula for each jk-entry in the matrix becoming (Wikipedia 2022):

$$\text{Cov}(\mathbf{X})_{jk} = \frac{1}{N-1} \sum_{i=1}^{N} \big(x_{ij} - \bar{x}_j\big)\,\big(x_{ik} - \bar{x}_k\big)\,.$$

And, indeed, this is the default in *Numpy*'s implementation (*Numpy Reference* 2022, `numpy.cov`).

What is the meaning of the principle of maximum variance?

In PCA a *principal component* is a direction (vector) in the data with the largest variation (Marsland 2015, p. 134). This is the direction which, hopefully, contains the most relevant information needed to separate the data, since variation is usually what we use when we want to make classifications. This is a nice way filter out dimensions with little information, potentially also reducing noise.

Why do we need this principle?

Does the principle always apply?

PCA is by no means fail proof, though; the direction with the most variation is not guarantied to contain the signal we are looking for, maybe we even end up filtering out the relevant axis since there happened to be some irrelevant, but extreme, variation in another direction. This is whats happening in Figure 3: the direction with the most variablity does not help us classify the data points, but there is a clear structure pressent (here captured by the second principal component).

Implementation: How Is PCA Implemented?

See the source files for the full implementation, but the vitals are quoted below.

Centering the Data, Computing Covariance Matrix, Computing Eigenvalues and Eigenvectors, Sorting Eigenvalues and Eigenvectors

In the implementation of the helper functions, I've closely followed the book (Marsland 2015, pp. 136–37). They are reduced to one-liners, but are essentially the same.

```
22 def center_data(A):
23   return A - np.mean(A, axis=0)
24
25 def compute_covariance_matrix(A):
26   return np.cov(A.T)
27
28 def compute_eigenvalue_eigenvectors(A):
29   return (eig := np.linalg.eig(A))[0].real, eig[1].real
30
31 def sort_eigenvalue_eigenvectors(eigval, eigvec):
32   return eigval[i := np.argsort(eigval)[::-1]], eigvec[:, i]
```

PCA Algorithm

For the main algorithm, see code below. Again, pretty much copied the book here (ibid., pp. 136–37), though some changes: The algorithm is refactored into several functions, as per assignment specification; and the vectors are *not* normalized, since the assignment tests apparently are not, too.

```
34 def pca(A, m):
35   A = center_data(A)
36   C = compute_covariance_matrix(A)
37   eigval, eigvec = compute_eigenvalue_eigenvectors(C)
38   eigval, eigvec = sort_eigenvalue_eigenvectors(eigval, eigvec)
39   eigvec = eigvec[:,:m]   # Slice `m` dims with lowest variance
40   return eigvec, np.dot(eigvec.T, A.T).T
```

All functions in this section should pass the test cases from the assignment text without issue.

Understanding: How Does PCA Work?

Visualize the PCA Projection

I've visualized the original iris data, the centered data, the principal eigenvector, and its one-dimensional projection in Figure 1. Plotting code is always an ugly mess, so see `pca.py` for implementation details.

Evaluation: When Are the Results of PCA Sensible?

Running PCA with Labels

See Figure 2 for a visual of both the original data and the principal component projected to one dimension. The data is now centered around the origin, and is one-dimensional. It seems the information necessary to separate the labels is preserved. The variability perpendicular to the principal component looks like noise, and would probably just make a classifier overfit.

Loading the Second Set of Labels

In Figure 3 I've plotted the data with the second set of labels. Both before and after doing PCA. I've plotted 2 components here. The first component, the one with most variability, is not suitable for separating the two classes, but the second one is—not *perfect* by any means, but better. Using both could potentially yield even better results, but might overfit if not careful, I think, since the data appears rather noisy.

Case Study 1: PCA for Visualization

Visualizing the Data by Selecting Features

In Figure 4 I've presented a scatter plot matrix of all the iris features in the data set, with histograms on the diagonal.

Visualizing the Data by PCA

In Figure 5 I've plotted the 4 principal components of the dataset. Also, see Figure 9 for a 2D representation of the first and second principal components. Honestly, unless

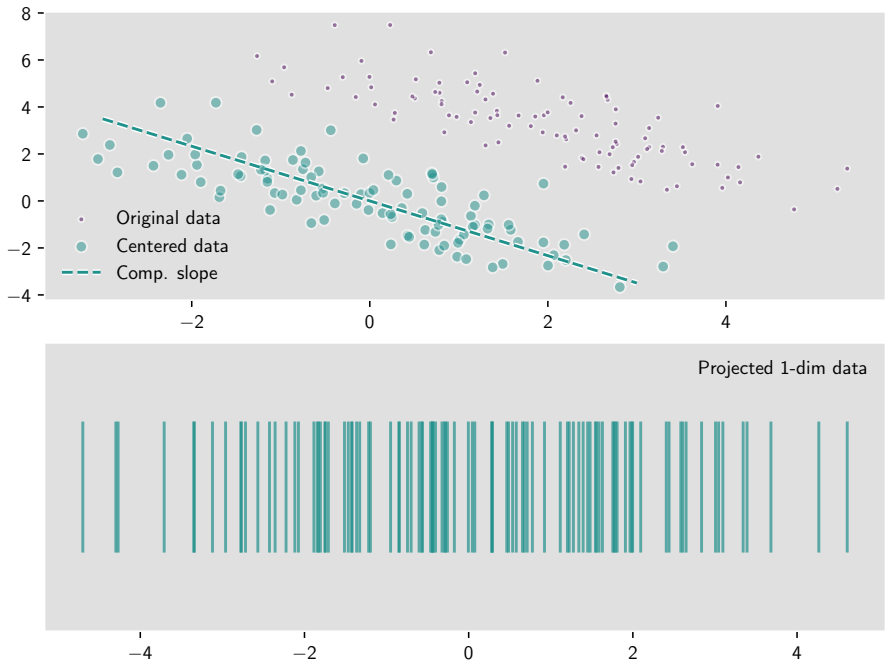

Figure 1: Visualizing PCA.

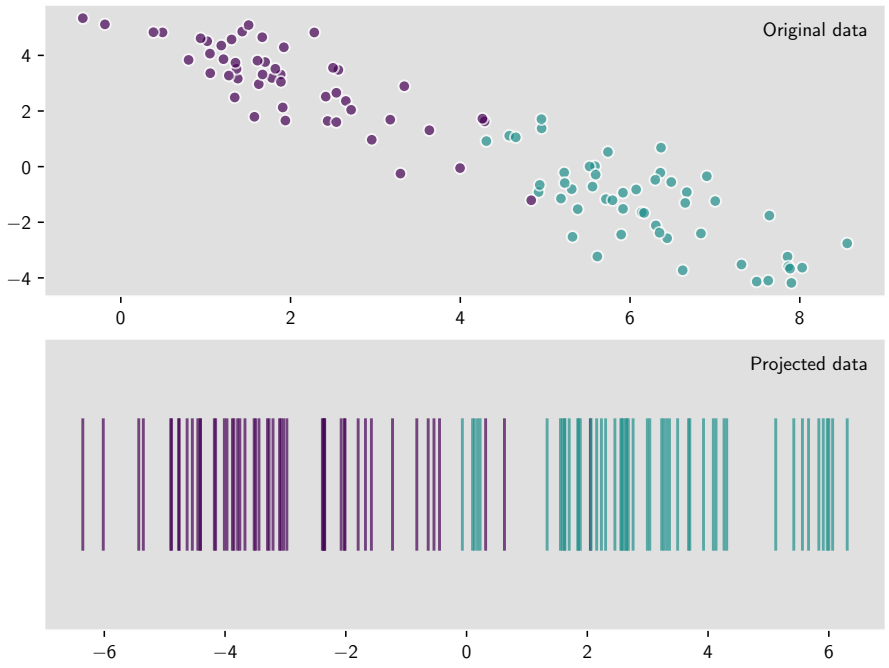

Figure 2: Visualizing PCA with labeled data (1)

Figure 3: Visualizing PCA with labeled data (2)

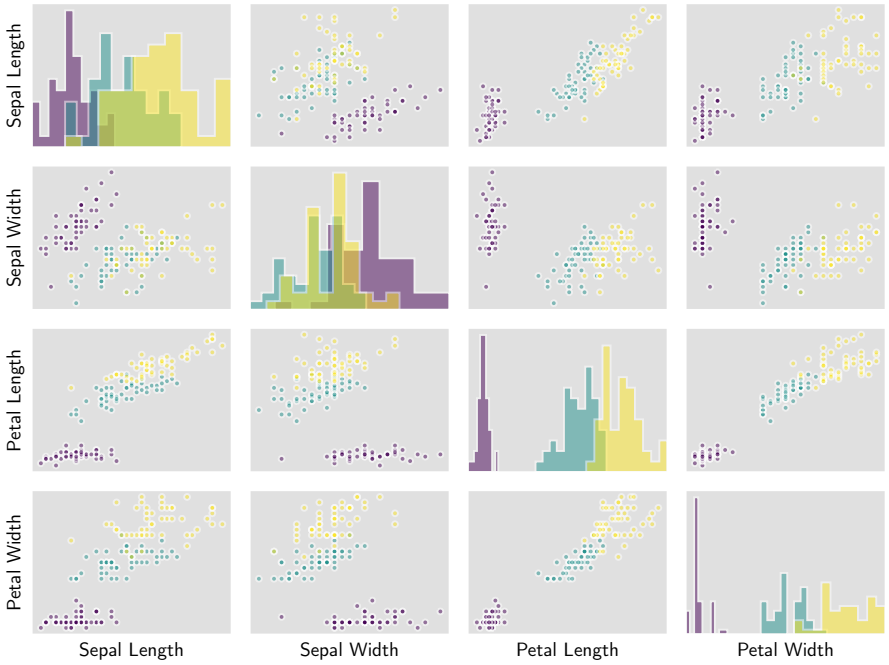

Figure 4: Scatter plot matrix of iris dataset features

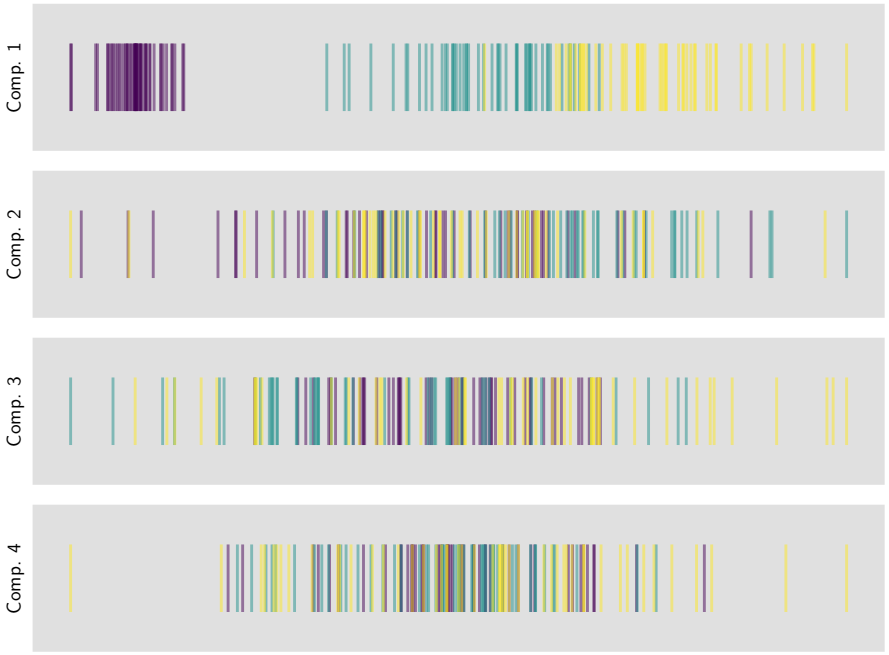

Figure 5: Visualizing iris dataset using PCA

I've made some mistake, visualizing by selecting features is much more informative for me. The first component contains useful information for sure (the others less so), but it is difficult to interpret. Whereas in the previous plot, I'm able to tell *why* an iris is classified the way it is.

Case Study 2: PCA for Compression

Inspecting the Data

A sample image (the first) from the *Faces in the Wild* dataset is shown in Figure 6.

Implementing a Compression-Decompression Function

I've followed the algorithm in Marsland 2015, page 137, but I had to do an additional transpose of the encoded matrix to line up the dimensions properly. I also used the matrix multiplier '`@`', instead of the dot product, since it produces the same results in this case[‡]. See the function definition below (from `pca.py`):

```
42  def encode_decode_pca(A, m):
43    return ((X := pca(A,m))[0] @ X[1].T).T + np.mean(A, 0)
```

Inspecting the Reconstructed Data

5 samples of images from data compressed down to 200 dimensions, then decompressed back to the original amount, is shown in Figure 7. It works surprisingly well, actually, considering the compressed data is approx. $200/2914 \approx 6.9\%$ of the original—if you disregard the vectors needed to apply the (de)compression, that is.

Evaluating Different Compressions

Figure 8 shows examples of images from dataset compressed down to m dimensions, with $m \in \{100, 200, 500, 1000\}$, then decompressed back to the original. I find it difficult to spot any difference between the originals and 1000 dimension compression. Even at 500 I have to squint to notice some slight changes in texture. At 200 dimension compression, though, it's really noticeable: sharp lines gets smoothed, textures scrambled, and some details disappears entirely. 100 is worse, but the essence of the original is still there. I think the 6th column from the right is interesting: the original picture almost in profile, the face slightly turned; but it looks like with heavier compression, the face

[‡]Not sure why, though.

Figure 6: Sample image (uncompressed)

Figure 7: Compressed–decompressed images using PCA, 200 dimensions

Figure 8: Compressed–decompressed images using PCA

turns more and more towards the camera, and the background gets reinterpreted as part of the face. I'm guessing this is because the original camera angle is novel, and in lower dimensions is discarded as "noise".

k-Means Clustering

See `kmeans.py` for the full script.

Qualitative Assessment

Projecting the Data Using PCA

A plot of the original data with the 'true' labels, along with 2 principal components, are shown in Figure 9.

Running k*-means*

I do the k-means clustering for $k \in \{2, 3, 4, 5\}$ like so[§]:

```
38  Ks = (2, 3, 4, 5)
39  yhats = {k: KMeans(k).fit_predict(P) for k in Ks}
```

Plots of the resulting labels are shown in Figure 10. The k-means predictions are not *too* bad. Especially for $k = 3$: though the boundary between class 1 and 2 (as y labels them) is not correct, it is not far off, and—in my opinion—looks smoother than the 'correct' y labeling. There are some minor, but glaring and annoying, 'mistakes' (at least by human standards) in the $k = 2$ case. For $k \in \{4, 5\}$ the clustering seems also logical, if there actually was more classes.

Quantitative Assessment

First, I find the accuracy of a classifier trained on the true labels:

[§]I'm wondering: why would I use $\mathbf{P}$, the projection, and not $\mathbf{X}$, the original data, here? Doesn't $\mathbf{X}$ contain more information for the k-means algorithm to detect? Or would this just be noise?

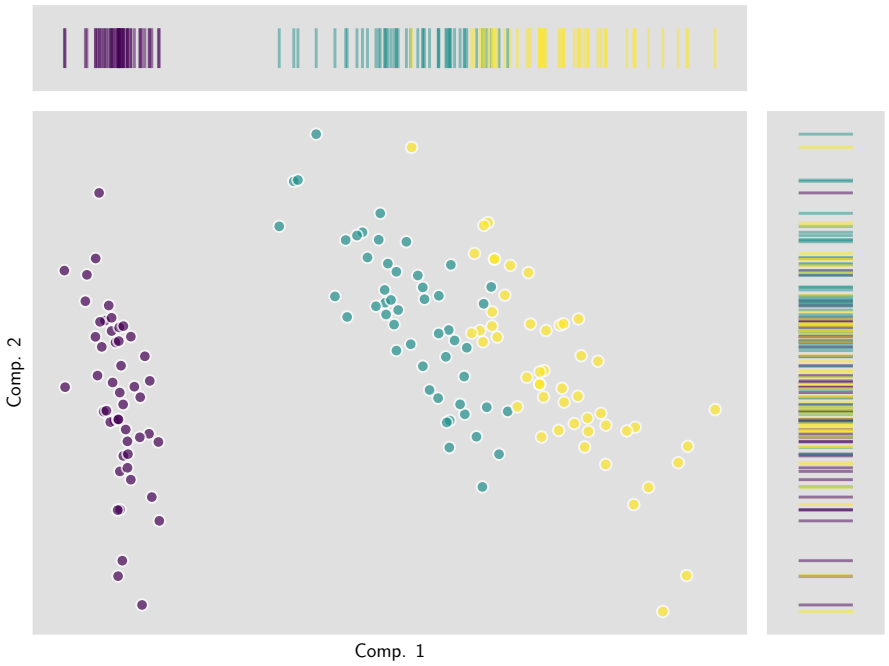

Figure 9: PCA on iris dataset

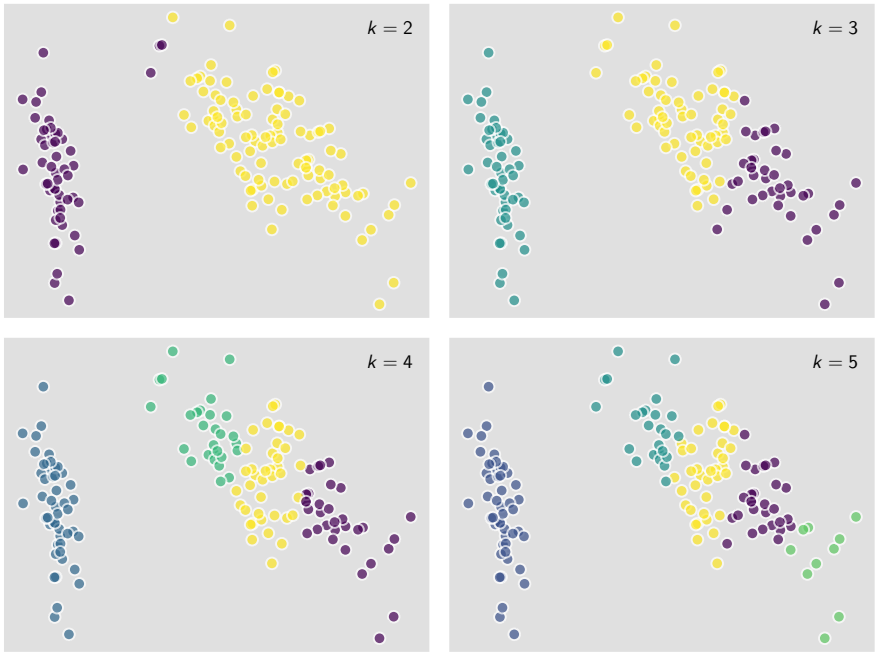

Figure 10: k-means clustering on iris dataset

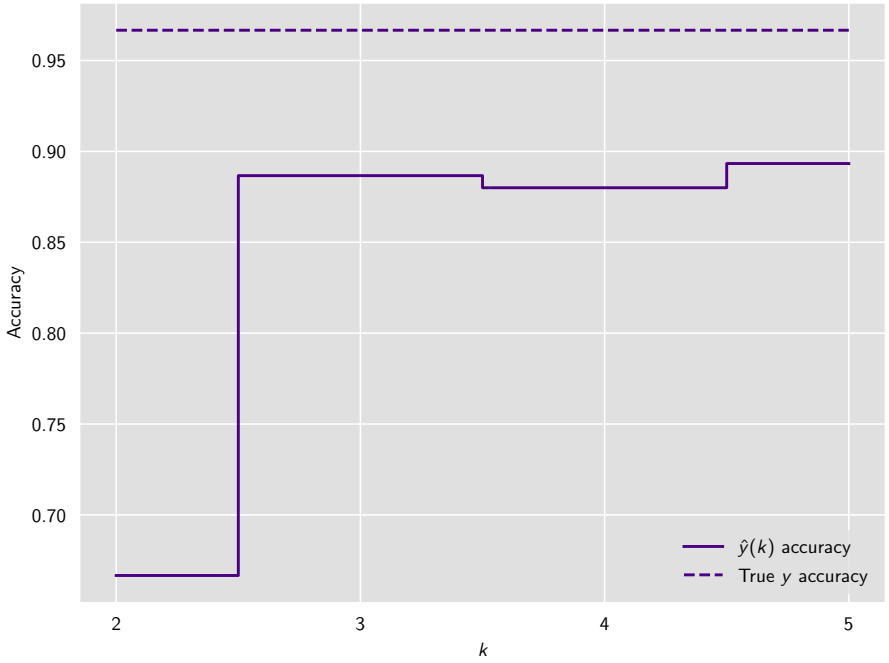

Figure 11: Accuracy of logistic classifiers trained on k-means output

```
60  y_acc = accuracy_score(y, LogisticRegression().fit(P, y).predict(P))
```

I train new classifiers on the k-means output from the previous section: (1) one-hot-encode, (2) fit classifiers, and (3) measure accuracy.

```
66  yhats = {k: np.eye(max(yhats[k] + 1))[yhats[k]] for k in Ks}
67  LMs   = {k: LogisticRegression().fit(yhats[k], y) for k in Ks}
68  acc   = {k: accuracy_score(y, LMs[k].predict(yhats[k])) for k in Ks}
```

The accuracy results are shown in Figure 11.

References

Devore, Jay L. and Kenneth N. Berk (2012). *Modern Mathematical Statistics with Applications*. eng. Second Edition. Springer Texts in Statistics. New York, NY: Springer New York. ISBN: 9781461403906.

Marsland, Stephen (2015). *Machine Learning: an Algorithmic Perspective*. 2nd edition. Chapman & Hall / CRC Machine Learning & Pattern Recognition Series. Boca Raton, FL: CRC Press. ISBN: 9781466583283.

Numpy Reference (14th Jan. 2022). URL: https://numpy.org/doc/stable/reference/ (visited on 21/04/2022).

Wikipedia, The Free Encyclopedia (2022). *Sample mean and covariance*. URL: https://en.wikipedia.org/w/index.php?title=Sample_mean_and_covariance&oldid=1063504049 (visited on 21/04/2022).