Chapter 16

Feature Extraction

Dimensionality reduction: feature extraction with PCA; self-organizing maps.

16.1 Dimensionality Reduction

In Chapter 15 we saw how to reduce the dimensionality of a data set by selecting only a subset of features, whether by filtering, using a wrapper to evaluate the performance of the learner for each subset of features or using learning algorithms that embed feature selection. In this chapter, we will see a different approach, *feature extraction*, which consists of computing new features using a function of the original features in the dataset. This approach is very useful in many cases, such as image processing, text mining or voice recognition. The main idea is to transform the original data into a more useful data set.

There are many domain-specific algorithms for feature extraction. Identifying regions of interest in an image requires different methods from extracting specific frequencies from a sound file, for example. But in this chapter we will focus on some generic approaches that do not depend on the type of problem. One widely used, and useful, approach is *Principal Component Analysis* (PCA).

16.2 Principal Component Analysis

Formally, PCA is a procedure for finding a transformation of a data set into an orthogonal set of coordinates chosen so that the values along each new coordinate are linearly uncorrelated. Another way of imagining PCA, is that we are going to choose the direction along which the data points have the greatest variance — that is, are more "spread out" — and then project the data in this direction, the *principal component*. Then we iteratively choose a new direction, orthogonal to all previous ones, using the same criterion of maximum variance.

Figure 16.1 illustrates this process. On the left panel, we see a set of points in three dimensions, and can note that the distributions over the different coordinates are not uncorrelated, since the point cloud is spread along a diagonal. If we compute the vector along this diagonal, one of the three vectors represented in red, and project the data in that direction, we can then find the next *principal component* by doing the same computation on the projected data. We can imagine repeating this proces until there is only one orthogonal direction left, giving the third vector in this case, since we started from three dimensions. The panel on the right shows the result of projecting the three-dimensional data into

the first two *principal components*. Note that, after this transformation, the coordinates are no longer linearly correlated, with the points no longer spread along a diagonal.

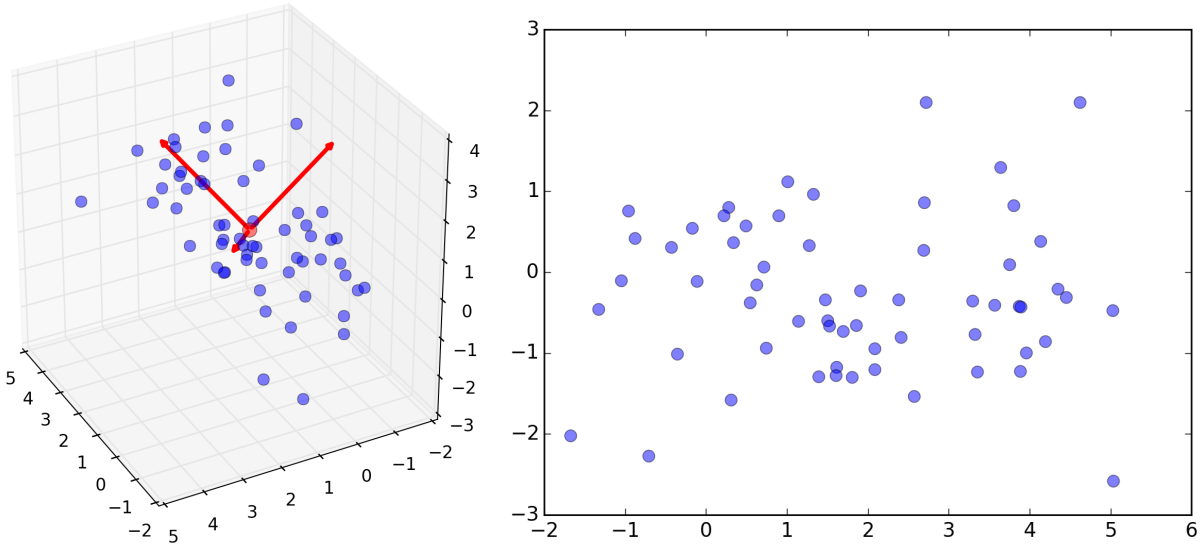

Figure 16.1: The left panel shows a three-dimensional data set with correlated coordinates, and corresponding three principal components. The right panel shows the projection of the original point into the first two principal components.

In practice, we do not compute the principal components in this iterative manner. This is just to make it easier to imagine the process. The way PCA is done is by computing the *eigenvectors* of the *covariance matrix* or, more precisely, of the *scatter matrix*[1]. The scatter matrix S can be computed by adding the matrices obtained by the outer products of all data vectors with themselves, after subtracting the mean vector:

$$m = \frac{1}{n}\sum_{k=1}^{n} x_k \qquad S = \sum_{k=1}^{n}(x_k - m)(x_k - m)^T$$

Using the Numpy library, we can compute the scatter matrix by computing the mean vector and then the outer products of the data points minus the mean vector. Note that, with the Numpy library, the mean vector cam be computed in a single instruction. This implementation is only to make it clearer how the vector is computed.

```
1  import numpy as np
2  mean_x = np.mean(data[0,:])
3  mean_y = np.mean(data[1,:])
4  mean_z = np.mean(data[2,:])
5  mean_v = np.array([[mean_x],[mean_y],[mean_z]])
6  scatter = np.zeros((3,3))
7  for i in range(data.shape[1]):
8      scatter += (data[:,i].reshape(3,1) - mean_v).dot((data[:,i].reshape(3,1) - mean_v).T)
9
10 print mean_v
11 [[ 1.07726488]
```

[1]The scatter matrix divided by the number of samples is the maximum likelihood estimator of the covariance matrix but, for our purposes, this scaling factor is not important, so we can use the scatter matrix directly. This explanation is based on the PCA demo authored by Sebastian Raschka, available at http://sebastianraschka.com/Articles/2014_pca_step_by_step.html

```
 [ 1.11609716]
 [ 1.03600411]]
print scatter
 [[ 110.10604771   39.91266264   52.3183266 ]
  [  39.91266264   80.68947748   34.48293948]
  [  52.3183266    34.48293948   97.58136923]]
```

Once we have the scatter matrix, we can compute the *eigenvectors* and corresponding *eigenvalues*. The eigenvectors of a matrix are those vectors which, after multiplication by the matrix, retain the same direction, changing only by a scalar factor. Thus, if v is an eigenvector of matrix A,

$$Av = \lambda v$$

The scaling factor λ is the corresponding *eigenvalue*, which can be used to sort the *eigenvectors* in order to give us the principal components in order of importance. The details of this computation fall outside the scope of this course, but we can use the `eig` function from the `linalg` module in the Numpy library. This function returns a vector with the eigenvalues and a matrix with the corresponding normalized eigenvectors, in columns (the first column of the matrix is the eigenvector corresponding to the first eigenvalue, and so on):

```
eig_vals, eig_vecs = np.linalg.eig(scatter)
print eig_vals
[ 183.57291365   51.00423734   53.79974343]
print eig_vecs
[[ 0.66718409  0.72273622  0.18032676]
 [ 0.45619248 -0.20507368 -0.8659291 ]
 [ 0.58885805 -0.65999783  0.46652873]]
```

The two largest eigenvalues are, in order, the first and the third. This means that these are the first two principal components of our data set, and the two best directions do choose to project the three-dimensional data into two dimensions, as shown in Figure 16.1. To do this, we combine these two vectors into a transformation matrix, then transform the data and plot it.

```
transf = np.vstack((eig_vecs[:,0],eig_vecs[:,2]))
t_data = transf.dot(data.T)
fig = plt.figure(figsize=(7,7))
plt.plot(t_data[0,:], t_data[1,:], 'o', markersize=7, color='blue', alpha=0.5)
plt.gca().set_aspect('equal', adjustable='box')
plt.savefig('L16-transf.png',dpi=200,bbox_inches='tight')
plt.close()
```

By plotting the first principal component in the x axis we get most of the variance in this axis, with the values ranging from -2 to 6. The second principal component, in the y axis, corresponds to the direction, orthogonal to the first, that has the largest of the remaining variance. In this case, the range is now only from -3 to 3. It is also worth noting that the projected points are no longer in a diagonal distribution, as the new coordinates now are linearly uncorrelated due to the transformation using the principal components.

The `decomposition` module of the Scikit-Learn library offers classes `PCA` and `RandomizedPCA` for principal component analysis. The `RandomizedPCA` is suitable for large datasets, using random samples of the data for the PCA instead of the complete dataset.

16.3 Self Organizing Maps

Another way of projecting a high dimension data set into a smaller set of dimensions is to use a *Self Organizing Map* (SOM). We can imagine the SOM as an artificial neural network whose neurons are arranged in a two-dimensional matrix, with each neuron in the SOM having a set of coefficients of the same dimension as the data set. This gives us two distance measures: we can measure the distance from the coefficients vector of any neuron to any point in the data set, and we can measure the distance within the neuron matrix from any neuron to its neighbours.

The SOM is trained by first assigning random values to the coefficients of the neurons. Then, iteratively, we start by finding the neuron closest to a data point, called the *Best Matching Unit* (BMU), and shifting the coefficients vector of the BMU neuron closer to the data point. Neurons that are close to the BMU in the SOM matrix are also moved in the same direction, though by a smaller amount decreasing with the distance to the BMU in the SOM matrix. The magnitude of these changes is a function of a learning coefficient that decreases monotonically during training. Figure 16.2

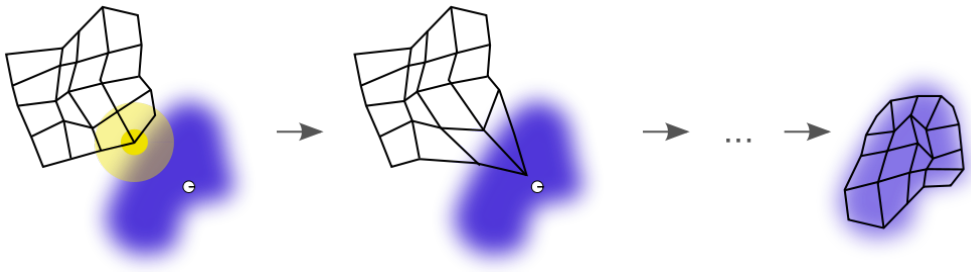

Figure 16.2: Training the SOM. As the coefficient vector of each neuron is changed, it "pulls" on the vectors of neighbouring neurons, making the neuron matrix adjust to the data set in the space of the data points. Image source: Wikipedia

To illustrate the use of a SOM, we will project the three-dimensional colour space into a two-dimensional matrix. Each colour is defined by a vector of 3 values, for the red, green and blue components. We will use the minisom module[2] to train a SOM of 20 by 30 neurons, for a total of 600 neurons[3]. We start by creating a labelled set of colors,

```
colors = np.array(
         [[0., 0., 0.],
          [0., 0., 1.],
          ...
          [.5, .5, .5],
          [.66, .66, .66]])
color_names = \
        ['black', 'blue', 'darkblue', 'skyblue',
         'greyblue', 'lilac', 'green', 'red',
         'cyan', 'violet', 'yellow', 'white',
         'darkgrey', 'mediumgrey', 'lightgrey']
```

The `MiniSom` class is initialized by providing the dimensions of the SOM. In order, the number of neurons in the x and y dimensions and the dimension of the input space. The `learning_rate` is the

[2] Available at https://github.com/JustGlowing/minisom

[3] This example is based on a SOM demo at the Multivariate Pattern Analysis in Python site: http://www.pymvpa.org/examples/som.html

multiplier for the adjustment in the neuron coefficients and `sigma` is a parameter defining the neighbourhood function on the SOM matrix. The methods `random_weights_init` and `train_batch` serve, respectively, to initialize the coefficients and train the SOM. The initialization consists of assigning random points from the training set to the SOM neurons as coefficients.

```
from minisom import MiniSom
import matplotlib.pyplot as plt
import numpy as np

plt.figure(1, figsize=(7.5, 5), frameon=False)
som = MiniSom(20, 30, 3, learning_rate=0.5, sigma = 2)
som.random_weights_init(colors)
som.train_batch(colors,10000)
```

To view the result, we can draw the matrix using the colours corresponding to the three coefficients of each SOM neuron, each coefficient corresponding to a colour channel. We can also draw the colour labels on the SOM matrix by placing them at the position of the SOM neuron whose coefficients are closer to the colour values. To do this, we use the `winner` method of the SOM object to obtain the coordinates, in the SOM matrix, of the Best Matching Unit for the colour vector. The code below details this process and Figure 16.3 shows the resulting image.

```
for ix in range(len(colors)):
    winner = som.winner(colors[ix])
    plt.text(winner[1], winner[0], color_names[ix], ha='center', va='center',
             bbox=dict(facecolor='white', alpha=0.5, lw=0))
plt.imshow(som.weights, origin='lower')
plt.savefig('L6-colors.png',dpi=300)
plt.close()
```

16.4 An example of feature extraction

To illustrate the process of feature extraction and data projection with a SOM, we'll examine data from the Gapminder site[4]. We have data on a set of indicators: *per capita* GDP, life expectancy, infant mortality and unemployment. Each indicator is available in an Excel spreadsheet file with one year in each column and one country in each row. Figure 16.4 illustrates the structure of these files.

The problem here is that the data is not uniform in quality. For each country and indicator there may be data for some years and not others, so there are different numbers of data points for different countries, as illustrated in Figure 16.5. This makes it hard to organize the information. So the first step will be to extract from these heterogeneous sets of data a set of features with a fixed dimension for all countries. We can do this by fitting each curve with a third degree polynomial. This will allow us to represent each country as a set of 16 features, with four features for the curve of each four indicators. Figure 16.6 shows examples of polynomial curves obtained from the standardized indicator values, with years and indicator values rescaled to a range of $[0, 1]$.

With this dataset with 16 dimensions, with a 16 dimensional vector describing each country, we can train a SOM in order to project the countries into a two-dimensional image according to their similarity in the pattern of the four indicators. We start by normalizing these data by subtracting the mean value

[4] http://www.gapminder.org

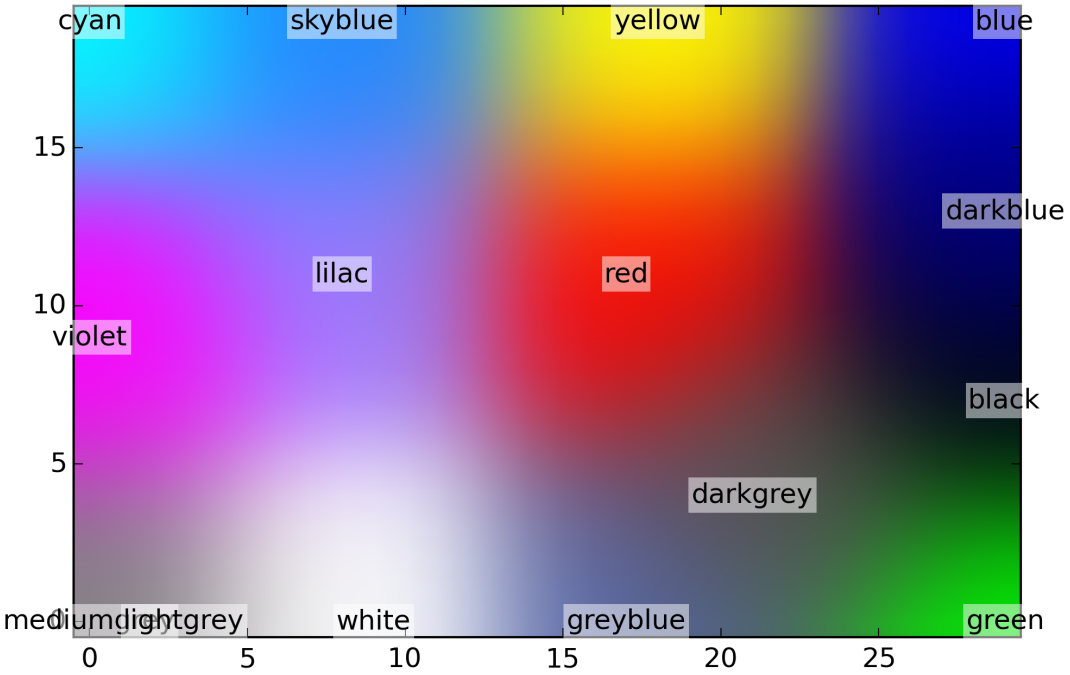

Figure 16.3: Result of training the SOM with the set of colours and labelling the colours at the respective SOM neurons.

BK25 28,799

	A	AR	AS	AT	AU	AV	AW	AX	AY
1	Life expectancy with projections	1807	1808	1809	1810	1811	1812	1813	1814
2	Afghanistan	28,139273	28,129027	28,11878	28,108533	28,098287	28,08804	28,077793	28,067547
3	Albania	35,4	35,4	35,4	35,4	35,4	35,4	35,4	35,4
4	Algeria	28,8224	28,8224	28,8224	28,8224	28,8224	28,8224	28,8224	28,8224
5	American Samoa								
6	Andorra								
7	Angola	26,98	26,98	26,98	26,98	26,98	26,98	26,98	26,98
8	Anguilla								
9	Antigua and Barbuda	33,536	33,536	33,536	33,536	33,536	33,536	33,536	33,536
10	Argentina	33,2	33,2	33,2	33,2	33,2	33,2	33,2	33,2
11	Armenia	33,995	33,995	33,995	33,995	33,995	33,995	33,995	33,995
12	Aruba	34,419	34,419	34,419	34,419	34,419	34,419	34,419	34,419
13	Australia	34,05	34,05	34,05	34,05	34,05	34,05	34,05	34,05
14	Austria	34,4	34,4	34,4	34,4	34,4	34,4	34,4	34,4
15	Azerbaijan	29,165	29,165	29,165	29,165	29,165	29,165	29,165	29,165

Figure 16.4: Data available for each indicator.

of each feature and dividing by the standard deviation. This is necessary because the coefficients of the polynomials can span a wide range of values.

```
descs = np.zeros((len(countries),len(data_names)*(degree+1)))
features = len(data_names)*(degree+1)
for ix in range(len(countries)):
    c = countries[ix]
    c_desc = c.descriptors.reshape((1,features))
    descs[ix,:] = c_desc
descs = (descs-np.average(descs,axis=0))/np.std(descs,axis=0)
```

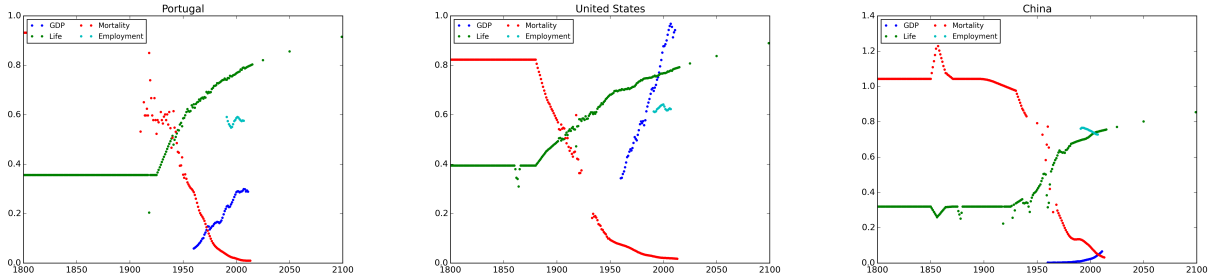

Figure 16.5: Examples of the data points available for the four indicators in three different countries

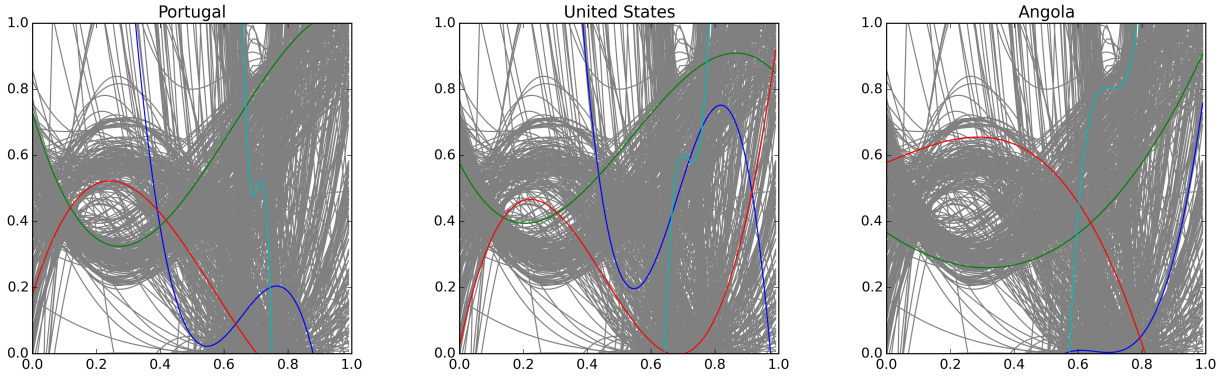

Figure 16.6: Polynomial curves adjusted to the indicator data points

Then we train the SOM and read a list with the set of countries to label on the neuron matrix.

```
1 som = MiniSom(30, 45, features, learning_rate=0.5, sigma = 2)
2 som.random_weights_init(descs)
3 som.train_batch(descs,10000)
4 to_plot = open('countries_to_plot.txt').readlines()
5 for ix in range(len(to_plot)):
6     to_plot[ix]=to_plot[ix].strip()
```

Finally, we can represent the SOM colouring each neuron on the matrix in a lighter colour the larger its average distance to its neighbours. Figure 16.7 shows the result, indicating the position in the SOM of the neurons closest to the selected countries.

```
1 plt.figure(1, figsize=(7.5, 5), frameon=False)
2 plt.bone()
3 plt.pcolor(som.distance_map()) # average dist. to neighs.
4 for ix in range(len(descs)):
5     if countries[ix].name in to_plot:
6         winner = som.winner(descs[ix])
7         plt.text(winner[1], winner[0], countries[ix].name,
8                  ha='center', va='center',color='lime')
9 plt.savefig('L6-countries_som.png',dpi=300)
10 plt.close()
```

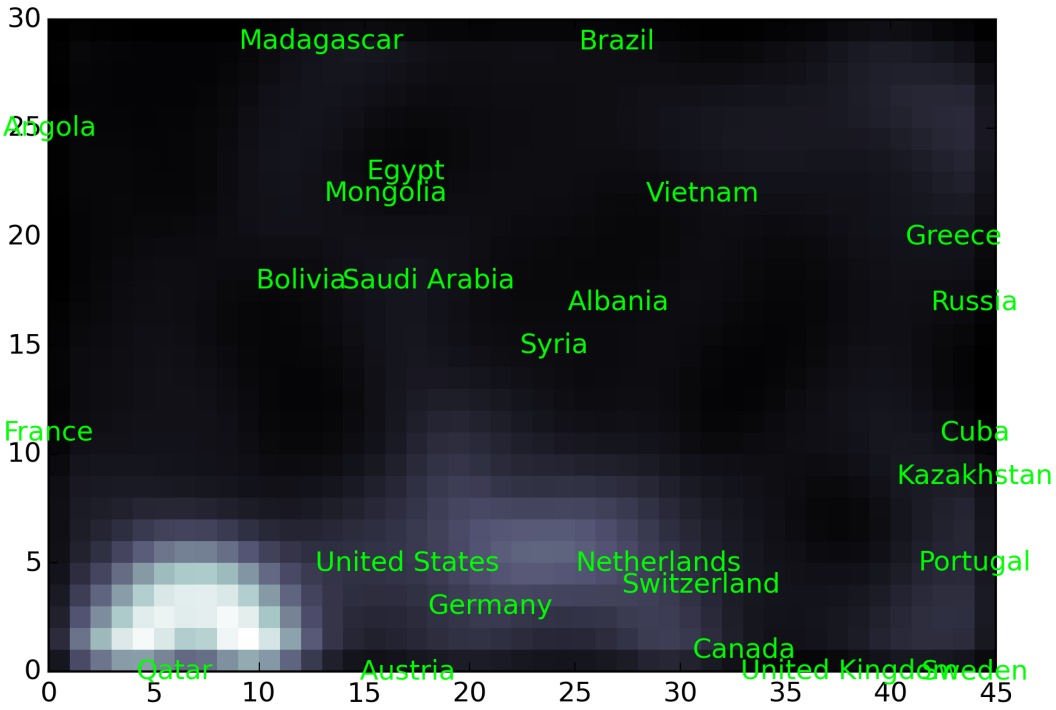

Figure 16.7: SOM with the projected planets.

16.5 Further Reading

1. PCA with Scikit-Learn: http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html

2. Wikipedia article on Self Organizing Maps: https://en.wikipedia.org/wiki/Self-organizing_map

Bibliography

[1] Uri Alon, Naama Barkai, Daniel A Notterman, Kurt Gish, Suzanne Ybarra, Daniel Mack, and Arnold J Levine. Broad patterns of gene expression revealed by clustering analysis of tumor and normal colon tissues probed by oligonucleotide arrays. *Proceedings of the National Academy of Sciences*, 96(12):6745–6750, 1999.

[2] Ethem Alpaydin. *Introduction to Machine Learning*. The MIT Press, 2nd edition, 2010.

[3] David F Andrews. Plots of high-dimensional data. *Biometrics*, pages 125–136, 1972.

[4] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, New York, 1st ed. edition, oct 2006.

[5] Deng Cai, Xiaofei He, Zhiwei Li, Wei-Ying Ma, and Ji-Rong Wen. Hierarchical clustering of www image search results using visual. Association for Computing Machinery, Inc., October 2004.

[6] Guanghua Chi, Yu Liu, and Haishandbscan Wu. Ghost cities analysis based on positioning data in china. *arXiv preprint arXiv:1510.08505*, 2015.

[7] Le Cun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Handwritten digit recognition with a back-propagation network. In *Advances in Neural Information Processing Systems*, pages 396–404. Morgan Kaufmann, 1990.

[8] Pedro Domingos. A unified bias-variance decomposition. In *Proceedings of 17th International Conference on Machine Learning. Stanford CA Morgan Kaufmann*, pages 231–238, 2000.

[9] Hakan Erdogan, Ruhi Sarikaya, Stanley F Chen, Yuqing Gao, and Michael Picheny. Using semantic analysis to improve speech recognition performance. *Computer Speech & Language*, 19(3):321–343, 2005.

[10] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[11] Brendan J Frey and Delbert Dueck. Clustering by passing messages between data points. *science*, 315(5814):972–976, 2007.

[12] Arthur E Hoerl and Robert W Kennard. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, 12(1):55–67, 1970.

[13] Patrick Hoffman, Georges Grinstein, Kenneth Marx, Ivo Grosse, and Eugene Stanley. Dna visual and analytic data mining. In *Visualization'97., Proceedings*, pages 437–441. IEEE, 1997.

[14] Chang-Hwan Lee, Fernando Gutierrez, and Dejing Dou. Calculating feature weights in naive bayes with kullback-leibler measure. In *Data Mining (ICDM), 2011 IEEE 11th International Conference on*, pages 1146–1151. IEEE, 2011.

[15] Stuart Lloyd. Least squares quantization in pcm. *Information Theory, IEEE Transactions on*, 28(2):129–137, 1982.

[16] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA., 1967.

[17] Stephen Marsland. *Machine Learning: An Algorithmic Perspective*. Chapman & Hall/CRC, 1st edition, 2009.

[18] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[19] Yvan Saeys, Iñaki Inza, and Pedro Larrañaga. A review of feature selection techniques in bioinformatics. *bioinformatics*, 23(19):2507–2517, 2007.

[20] Roberto Valenti, Nicu Sebe, Theo Gevers, and Ira Cohen. Machine learning techniques for face analysis. In Matthieu Cord and Pádraig Cunningham, editors, *Machine Learning Techniques for Multimedia*, Cognitive Technologies, pages 159–187. Springer Berlin Heidelberg, 2008.

[21] Giorgio Valentini and Thomas G Dietterich. Bias-variance analysis of support vector machines for the development of svm-based ensemble methods. *The Journal of Machine Learning Research*, 5:725–775, 2004.

[22] Jake VanderPlas. Frequentism and bayesianism: a python-driven primer. *arXiv preprint arXiv:1411.5018*, 2014.