`

10.1 KDE: An Introduction

Kernel density estimation (KDE) is a technique to estimate (probability) density of a variable or joint probability density of a group of variables from data without any prior knowledge about the form of the underlying distribution. Figure 10.1 shows a 1D example where the estimated KDE density curve is superimposed on the histogram built using observed data. As is evident, KDE is a robust way of estimating probability densities. Using the KDE curve, we can estimate the likelihood of new variable measurement falling in any specified range.

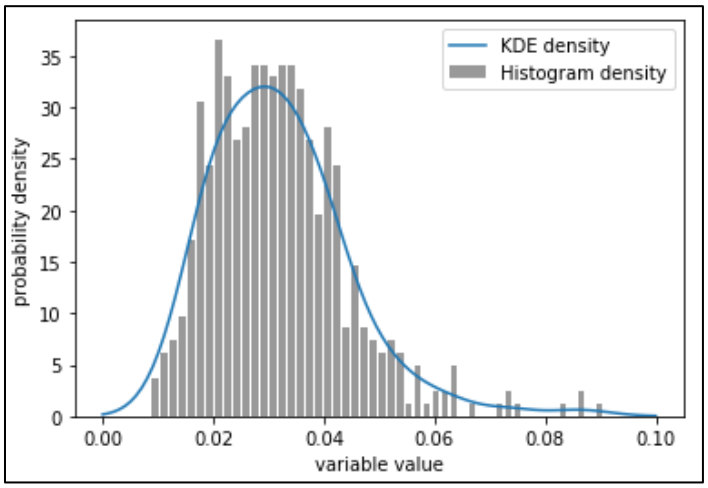


Figure 10.1: Probability density via histogram and KDE

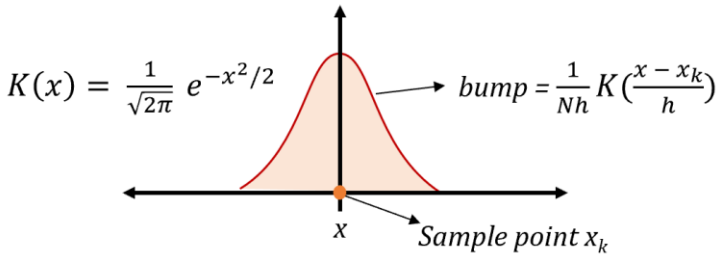
To see one of the utilities of KDE, recollect the two ways we have employed to estimate the threshold of a process monitoring metric till now. We computed the percentile of either the samples (for ICA) or an assumed distribution (such as a F-distribution for PCA T^2 metric). While the first approach is inappropriate when number of samples is low, the second approach is impractical for complex systems (or systems where process variables do not follow gaussian distribution) where the metric distribution is not known beforehand. In these scenarios, KDE can be employed for control limit estimation.

One can use histogram as well for estimating probability densities. However, as seen in Figure 10.1, non-smooth values are obtained, i.e., densities of neighboring points have sharp jumps. These sharp jumps are not due to the underlying true distribution but are just an artifact of the choice of bin locations. The density estimates from histogram strongly depend on the choice of the number of bins and bin locations. These problems become more serious for high-dimensional datasets. Therefore, KDE is preferred over histograms for density estimation.

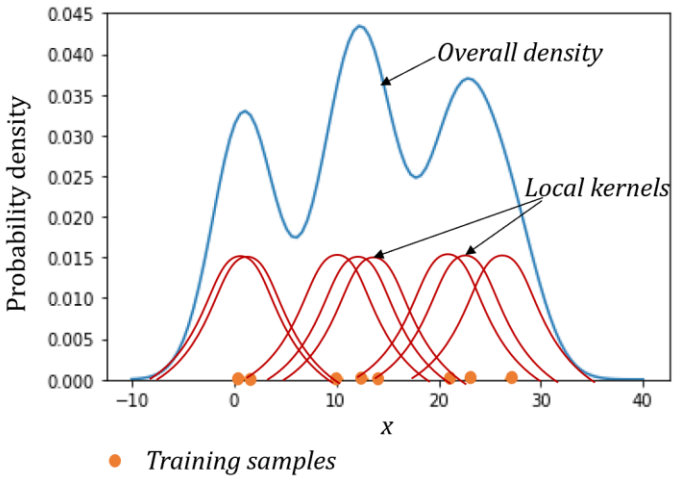
In the beginning of the chapter, we classified KDE and k-NN as non-parametric methods. Parametric vs non-parametric is an important machine learning classification. Parametric methods assume a fixed form of the underlying model where only the model parameters are estimated using data; non-parametric methods do not make any such assumptions. A parametric approach for density estimation would be to assume a gaussian density function and then estimate mean and standard deviation from data. Other popular non-parametric methods that you have already studied are decision trees and SVMs.

Mathematical background

In Gaussian mixture modeling, the overall density was approximated as a sum of local gaussians centered at different clusters. KDE takes this idea to its extreme and puts a local density function (also called kernel) at each data point as shown below for a 1D case, where a popular choice for kernel, *K*, Gaussian kernel is used, and N is number of training samples.



The parameter *h* in the above illustration is called bandwidth or smoothing parameter and helps determine the width of local bumps. The overall density at any point ($\boldsymbol{p_{KDE}(x)}$) is obtained by simply summing up the local bumps as shown below.



$$p_{KDE}(x) = \frac{1}{Nh} \sum_{k=1}^{N} K(\frac{x - x_k}{h})$$

`

The kernels are chosen such that $\int_{-\infty}^{\infty} K(x)\, dx$, the total area under the KDE curve, is 1. For multidimensional case with *D* dimensions, $p_{KDE}(\boldsymbol{x})$ is given by

$$p_{KDE}(\boldsymbol{x}) = \frac{1}{Nh^D} \sum_{k=1}^{N} K\left(\frac{\boldsymbol{x} - \boldsymbol{x}_k}{h}\right)$$

The above multivariate KDE uses the same bandwidth for all axes; different bandwidths are used to deal with non-uniform spread along different axes. We will deal primarily with univariate problems in this chapter.

Deciding KDE hyperparameters

The kernel K and the bandwidth are the two hyperparameters needed by KDE algorithm. Figure 10.2 shows the kernels available with scikit-learn. The choice of kernel is typically not very crucial and Gaussian kernel is usually the default choice.

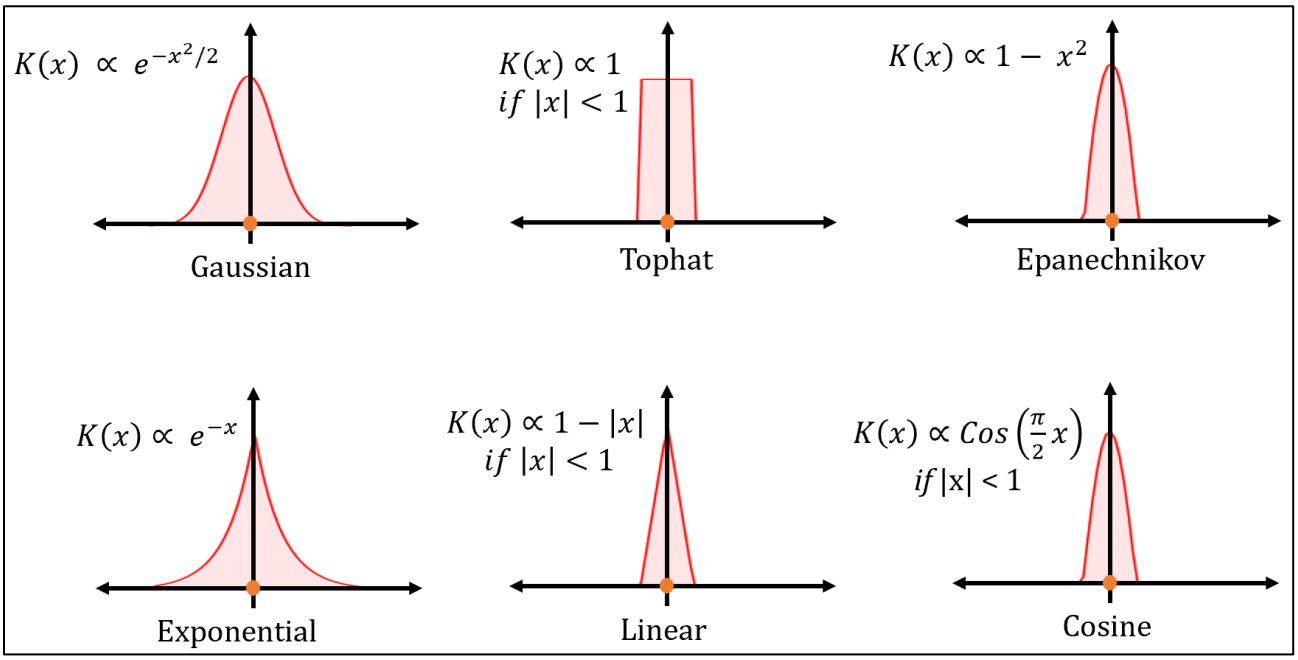


Figure 10.2: Available KDE kernels with scikit-learn

The bandwidth, however, is a crucial parameter and should be chosen carefully. Too small bandwidth leads to spikes in density estimates due to overfitting; absence or presence of a single data-point can significantly impact the density estimate resulting in high variance. Too large bandwidth over-smooths the density curve potentially masking the critical structure in data; the discrepancy between the true and estimated density becomes large resulting in high bias. Figure 10.3 illustrates this bias-variance trade-off.

`

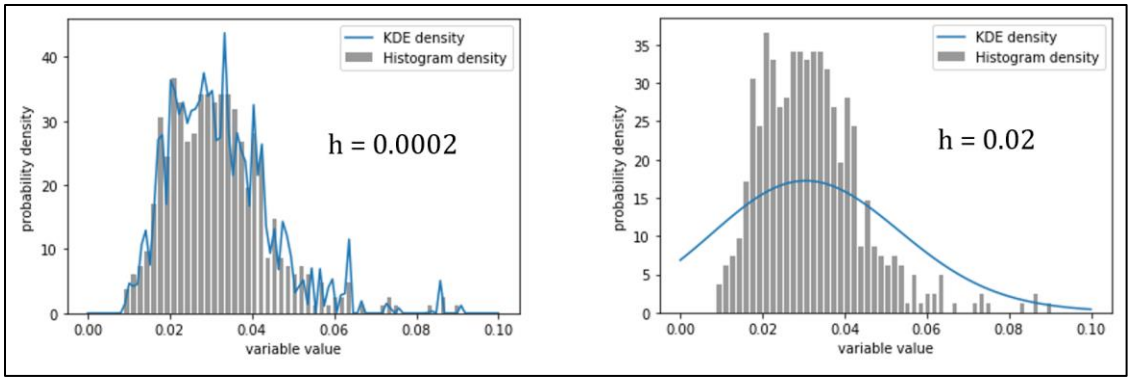


Figure 10.3: Overfitted KDE (left) and underfitted KDE (right)

An empirical approach to choosing *h* is to assume some underlying standard distribution and then minimize the error between KDE estimate and the assumed distribution. With gaussian assumption, bandwidth is given by

$$h = 1.06\sigma N^{-1/5}$$

where σ is sample standard deviation. A more robust modified version is given by

$$h = 0.9AN^{-1/5} \; ; A = min(\sigma, \frac{IQR}{1.34})$$

where IQR is the interquartile range. These empirical approaches, however, may not give good results. A more systematic approach is to use cross validation to choose the bandwidth value that maximizes the probability or likelihood of validation data. For this purpose, GridSearchCV can be used which automatically returns the optimal bandwidth among supplied bandwidth values. GridSearchCV uses score method of KernelDensity estimator which returns the log-likelihood (logarithm of density) at the validation data-points.

To apply KDE, let us revisit our ICA example from Chapter 6. We will determine the monitoring control limits for the TEP process using KDE. Let us first see how to fit an optimal KDE curve to the monitoring metric values. For illustration, we will use $\boldsymbol{I_e^2}$ values from the training samples. Note that we use the empirical expression of h to decide the list of bandwidth values for grid search.

```
# finding optimal KDE bandwidth by GridSearchCV
from sklearn.neighbors import KernelDensity
from sklearn.model_selection import GridSearchCV

N = len(ICA_statistics_train[:,1])
empirical_h = 1.06*np.std(ICA_statistics_train[:,1])*N**(1/5)
h_grid = np.linspace(0,5,50)*empirical_h  # 50 candidate values
```

`

```
grid = GridSearchCV(KernelDensity(), {'bandwidth': h_grid}, cv=N) # leave-one-out CV
grid.fit(ICA_statistics_train[:,1][:,None]) # 2D array is expected

print('Empirical h {:.4f}: optimal h {:.4f}'.format(empirical_h, grid.best_params_['bandwidth']))

>>>  Empirical h 0.0461: optimal h 0.0047

# optimal KDE and histogram
kde = grid.best_estimator_
metric_grid = np.linspace(0,0.1,100)[:,None]
density = np.exp(kde.score_samples(metric_grid)) # exponential taken because score_samples
                                                 method returns logarithm of density

plt.hist(ICA_statistics_train[:,1], bins=50, color='grey', histtype='bar', alpha=0.8, density=True,
rwidth=0.7, label='Histogram density')
plt.plot(metric_grid, density, alpha=1, label='KDE density')
plt.ylabel('probability density'), plt.xlabel('Ie2 values')
```

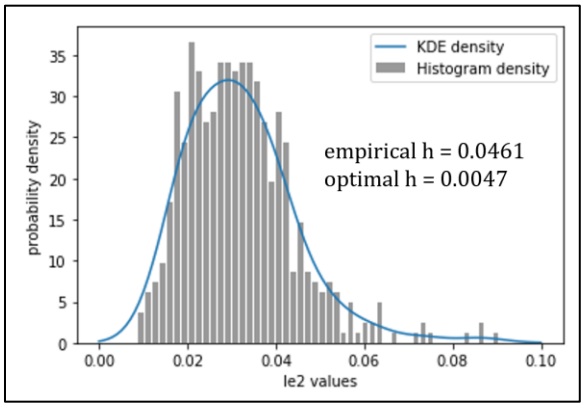


Figure 10.4: Optimal KDE curve fitted to the I_e^2 monitoring metric

We can see that GridSearchCV returns an optimal KDE that correctly reflects the distribution of the underlying data without overfitting the spurious details.

10.2 Determining Monitoring Metric Control Limit via KDE

Let's say we want to find the 99% control limit for our monitoring metric. With an optimal KDE curve on hand, the required control limit would be the metric value such that the area under

`

the KDE curve to the right of this value is 0.01 as illustrated in Figure 10.5. By definition, there would be only 1% probability that metric value for test sample (which comes from the same distribution from which the training samples were obtained) is higher than the 99% control limit. In process monitoring parlance, this implies that there is a 1% probability that a fault alarm is a false alarm.

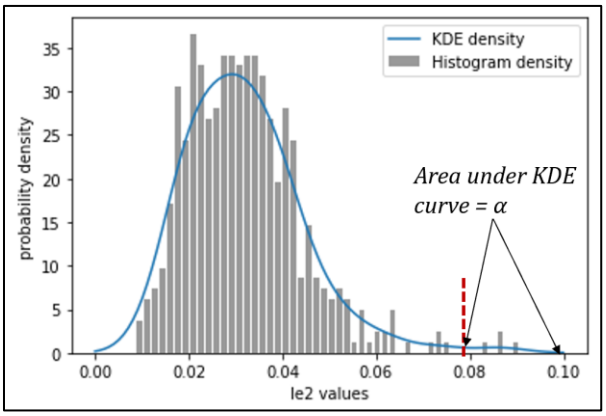


Figure 10.5: 100(1-α) percentile or control limit of a monitoring metric. α = 0.01 for a 99% control limit or 99th percentile.

Numerical integration is needed to find the area under KDE curve between any two metric values. Let us define a function that takes in metric values and returns a control limit corresponding to a specified percentile. As you can see, we divided the metric axis into a grid (metric_grid) with 100 intervals and computed the area under the KDE curve to the left of each of these grid points. The grid point with the required minimum area (as specified by the percentile parameter) becomes our control limit.

```
def get_CL_via_KDE(metric_values, percentile, metricLabel):
    """ calculate control limit for a monitoring statistic via KDE

    parameters
    ----------
    metric_values: numpy array of shape = [n_samples,]
    """

    # find optimal KDE bandwidth via CV
    N = len(metric_values)
    empirical_h = 1.06*np.std(metric_values)*N**(1/5)
    h_grid = np.linspace(0,5,50)*empirical_h

    CV_grid = GridSearchCV(KernelDensity(),{'bandwidth': h_grid}, cv=N) # leave-one-out CV
    CV_grid.fit(metric_values[:,None]) # 2D array is expected
    optimal_h = CV_grid.best_params_['bandwidth']
```

`

```
    # compute density values via KDE
    kde = CV_grid.best_estimator_
    metric_grid = np.linspace(0, np.max(metric_values), 100)[:,None]
    metric_pdf_values = np.exp(kde.score_samples(metric_grid)) # pdf = probability density function

    # Find control limit via numerical integration
    metric_cdf_values = [np.trapz(metric_pdf_values[:i, None], metric_grid[:i,:], axis=0)[0] for i in range(100)] # cdf = cumulative density function
    metric_CL = metric_grid[np.argmax(np.array(metric_cdf_values) > percentile/100),0]

    return metric_CL
```

We can now utilize this function to compute the 99% control limits for our ICA monitoring metrics. We can see that the control limits are similar to what we had obtained earlier in Chapter 6.

```
I2_CL = get_CL_via_KDE(ICA_statistics_train[:,0], 99, 'I2')
Ie2_CL = get_CL_via_KDE(ICA_statistics_train[:,1], 99, 'Ie2')
SPE_CL = get_CL_via_KDE(ICA_statistics_train[:,2], 99, 'SPE')

draw_ICA_monitoring_charts(ICA_statistics_train, [I2_CL, Ie2_CL, SPE_CL], 'training')
```

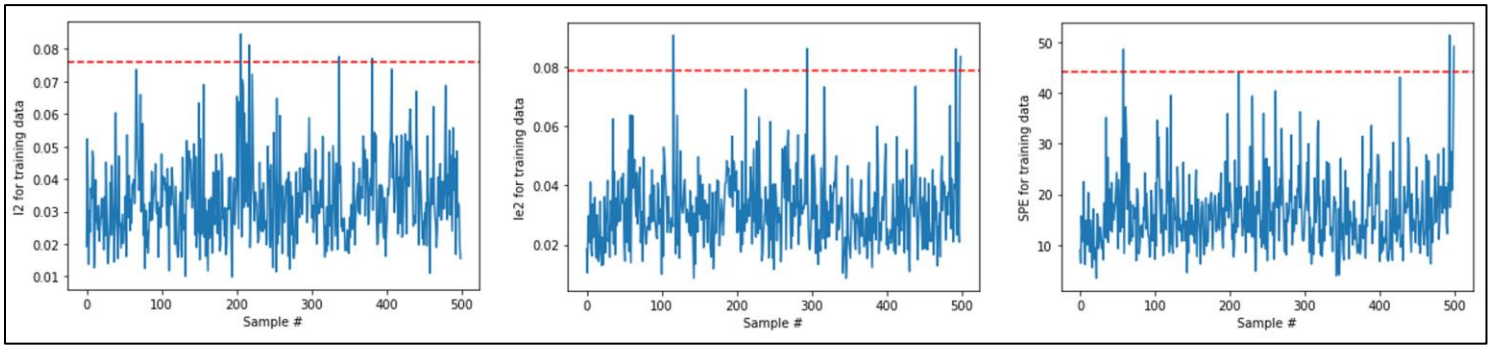


Figure 10.6: ICA monitoring charts for training data with control limits determined via KDE

Data density estimation via KDE has other uses as well. Samples in low-density regions can be segregated from high density regions and discarded as outliers. You can use KDE for the scenario shown in Figure 4.15 for your data pre-processing. Additionally, a process monitoring tool can be built using KDE as well by employing the logic that a test sample falling in low density regions (as estimated from training data) would be an abnormal sample. For all these applications, the procedure for building KDE models remains the same.

`

10.3 KNN: An Introduction

The k-nearest neighbor (k-NN or KNN) is a versatile technique based on a simple intuitive idea that the label/value for a new sample can be obtained from the labels/values of closest neighboring samples (in the feature space) from the training dataset. The parameter *k* denotes the number of neighboring samples utilized by the algorithm. As shown in Figure 10.7, k-NN can be used for both classification and regression. For classification, KNN assigns test sample to the class that appears the most amongst the *k* neighbors. For regression, the predicted output is the average of the value of the *k* neighbors. Due to its simplicity, k-NN is widely used for pattern classification and was included in the list of top 10 algorithms in data mining.[48]

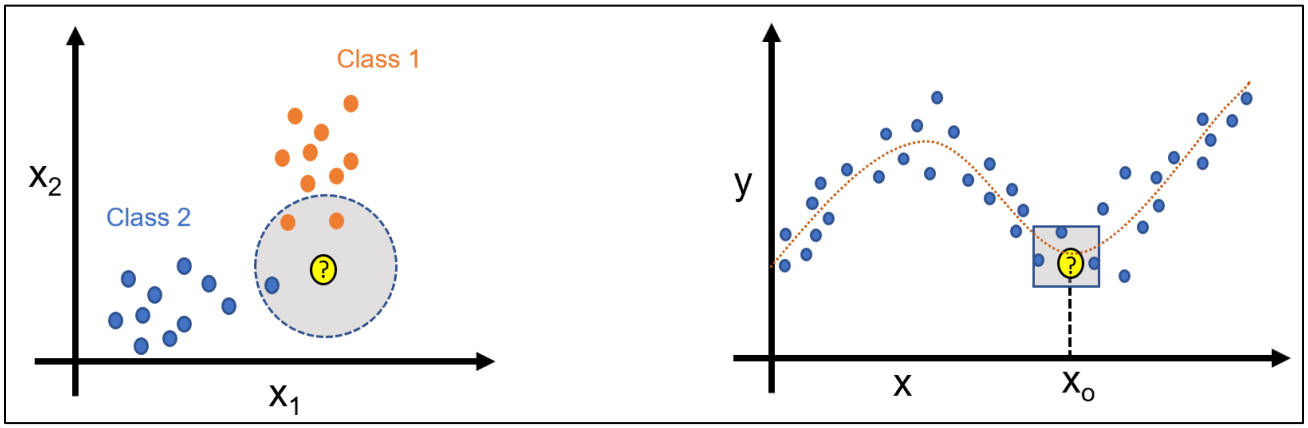


Figure 10.7: KNN illustration for classification (left) and regression (right). Yellow data point denotes unknown test sample. The grey-shaded region represents the neighborhood with 3 nearest samples.

KNN belongs to the class of lazy learners where models are not built explicitly until test samples are received. At the other end of the spectrum, eager learners (like, SVM, decision trees, ANN) 'learn' explicit models from training samples. Unsurprisingly, training is slower, and testing is faster for eager learners. KNN requires computing the distance of the test sample from all the training samples, therefore, KNN also falls under the classification of instance-based learning. Instance-based learners make predictions by comparing the test sample with training instances stored in memory. On the other hand, model-based learners do not need to store the training instances for making predictions.

[48] Wu et al., Top 10 algorithms in data mining. Knowledge and Information systems, 2008.

`

Conceptual background

Apart from an integer *k* and input-output training pairs, k-NN algorithm needs a distance metric to quantify the closeness of a test sample with the training samples. The standard Euclidean metric is commonly employed. Once the nearest neighbors have been determined, two approaches, namely uniform and distance-based, can be employed to decide weights assigned to each neighbor which impacts the neighbor's contribution in prediction. In uniform weighting, all *k* neighbors are treated equally while, in distance-based weighting, each of the *k* neighbors is weighted by the inverse of their distance from the test sample so that closer neighbors will have greater contributions. The figure below illustrates the difference between the two weight schemes for a classification problem

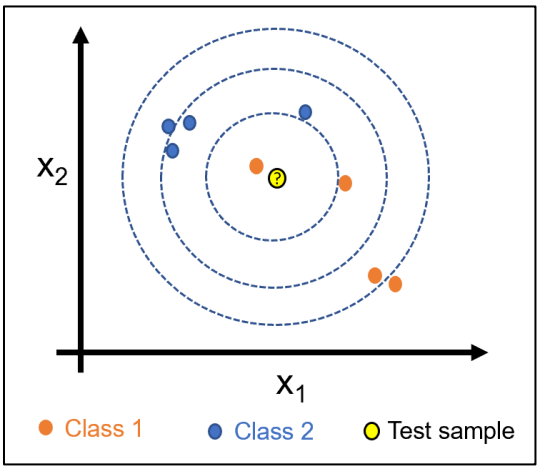


Figure 10.8: Illustration on impact of weight-scheme on k-NN output. Dashed circles are for distance references.

In Figure 10.8, with uniform weighting (also called majority voting for classification problems), the test sample is assigned to class 1 for $k = 1$ or 3 and class 2 for $k = 6$. For $k = 8$, no decision can be made. With distance-weighting, test sample is always classified as class 1 for k = 1, 3, 6, or 8. This illustration shows that distance weighting can help reduce the prediction dependence on the choice of *k*.

For predictions, k-NN needs to compute the distance of test samples from all the training samples. For large training sets, this computation can become expensive. However, specialized techniques, such as KDTree and BallTree, have been developed to speed up the extraction of neighboring points without impacting prediction accuracies. These techniques utilize the structure in data to avoid computing distances from all training samples. The NearestNeighbors implementation in scikit-learn automatically selects the algorithm best suited to the problem at hand. The KNeighborsRegressor and KNeighborsClassifier modules are provided by Scikit-learn for regression and classification, respectively.

`

A couple of things to pay careful attention in k-NN include variable selection and variable scaling. Variables that are not important for output predictions should be removed; otherwise unimportant variables will undesirably impact the determination of nearest neighbors. Further, the selected variables should be properly scaled to ensure that variables with large magnitudes do not dwarf the contribution of other variables during distance computations.

Deciding k-NN hyperparameters

The choice of the hyperparameter, *k*, is critical and is the knob to balance underfitting and overfitting. Small *k* can lead to overfitting where noisy measurements unfavorably impact the predictions. Large *k* can reduce the effect of noise but can lead to underfitting as it over-smooths the predictions for regression or make classification boundaries less distinct. Unfortunately, there is no standard guidance on the choice of *k*. The optimal value depends on the data and can be estimated via cross-validation.

Applications of k-NN for process systems

As alluded to before, k-NN method is often employed for equipment condition monitoring[49]. For process-level monitoring, k-NN classification can be used to classify process abnormalities into distinct fault classes if sufficient historical faulty samples are available. As opposed to k-NN fault classification, an interesting adaptation of k-NN for fault detection was proposed by He & Wang that made use of only normal operation data[50]. We will study this application in more detail in the next section.

A few other notable applications of k-NN for process systems include the work of Facco et al. on automatic maintenance of soft sensors[51], Borghesan et al. on forecasting of process disturbances[52], Cecilio et al. on detecting transient process disturbances[53], and Zhou et al. on fault identification in industrial processes[54]. These applications may not utilize the k-NN method directly for classification or regression, but use the underlying concept of similarity of nearest neighbors.

[49] Dong Wang, K-nearest neighbors based methods for identification of different gear crack levels under different motor speeds and loads: Revisited, Mechanical Systems and Signal Processing, 2016

[50] He and Wang, Fault detection using k-nearest neighbor rule for semiconductor manufacturing processes, IEEE Transactions on Semiconductor Manufacturing, 2007.

[51] Facco et al., Nearest-neighbor method for the automatic maintenance of multivariate statistical soft sensors in batch processing, Industrial Engineering & Chemistry Research, 2010.

[52] Borghesan et al., Forecasting of process disturbances using k-nearest neighbors, with an application in process control, Computers and Chemical Engineering, 2019.

[53] Cecilio et al., Nearest neighbors methods for detecting transient disturbances in process and electromechanical systems, Journal of Process Control, 2014.

[54] Zhou et al., Fault identification using fast k-nearest neighbor reconstruction, Processes, 2019

`

10.4 Process Fault Detection via KNN for Semiconductor Manufacturing Process

Fault detection by k-NN[28] (FD-KNN) is based on a simple idea that distance of a faulty test sample from the nearest training samples (obtained from normal operating plant conditions) must be greater than a normal sample's distance from the neighboring training samples. Incorporating this idea into the process monitoring framework, a monitoring metric (termed KNN squared distance) is defined for each training sample as follows

$$D_i^2 = \sum_{j=1}^{k} d_{ij}^2$$

where d_{ij}^2 is the distance of i^{th} sample from its j^{th} nearest neighbor. After computing k-NN squared distances for all the training samples, a threshold corresponding to the desired confidence limit can be computed. A test sample would be considered faulty if its k-NN squared distance is greater than the threshold.

An advantage of FD-KNN is that it is applicable to process systems with complex characteristics, such as multimodality, non-gaussianity, and non-linearity. To illustrate a complete application of FD-KNN, we will use the metal-etch dataset that we used previously in Chapter 8 for GMM-based process monitoring. If you recall, the batch data was unfolded before scaling and PCA application. We will do the same pre-processing transformations here using Sklearn's Pipeline feature. Following He & Wang's work, *k* was set to 5. To compute the monitoring metric threshold, we will use KDE.

```
# scale data & fit PCA model via pipeline
from sklearn.pipeline import Pipeline

pipe = Pipeline([('scaler', StandardScaler()), ('pca', PCA(n_components = 3))])
score_train = pipe.fit_transform(unfolded_dataMatrix)

#  k-nearest neighbors of each training sample in score space
from sklearn.neighbors import NearestNeighbors

nbrs = NearestNeighbors(n_neighbors=6).fit(score_train) # a data-point is its own neighbor
d2_nbrs, indices = nbrs.kneighbors(score_train)
d2_sqrd_nbrs = d2_nbrs**2
D2 = np.sum(d2_sqrd_nbrs, axis = 1)
D2_log = np.log(D2)
```

`

```
D2_log_CL = get_CL_via_KDE(D2_log, 95, 'D2_log')
```

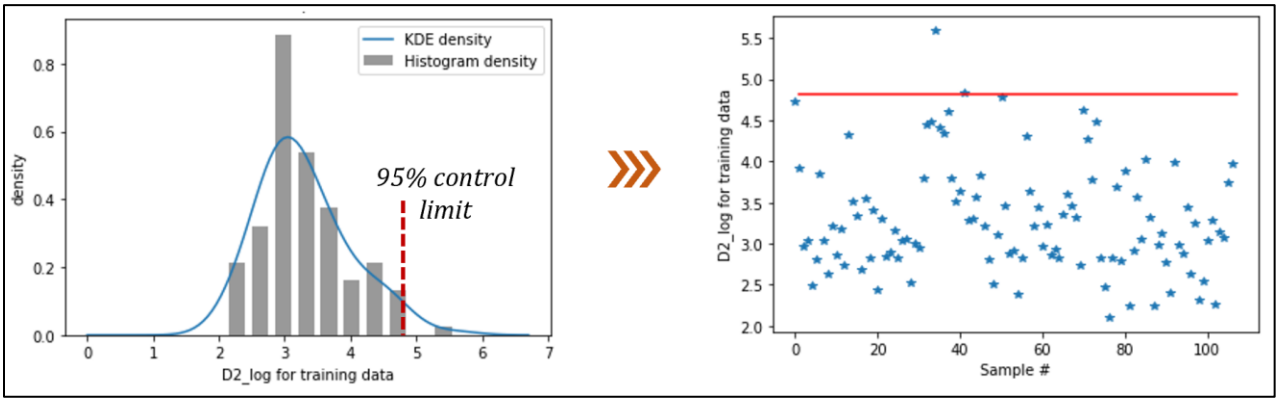


Figure 10.9 FD-KNN monitoring chart for metal-etch training samples

Figure 10.9 shows the output from fitting KDE to k-NN squared distances distribution and the resulting monitoring chart for the training samples. Note that we log-transformed the k-NN squared distances because application of KDE to D^2 resulted in non-zero densities for D^2 below 0. Figure 10.10 shows the monitoring chart for the faulty test samples. 15 out of 20 samples are correctly flagged as faulty while 5 samples are misdiagnosed as normal.

```
# scale and PCA on faulty test data
score_test = pipe.transform(unfolded_TestdataMatrix)

# D2_log_test
d2_nbrs_test, indices = nbrs.kneighbors(score_test)
d2_nbrs_test = d2_nbrs_test[:,0:5] # we want only 5 nearest neighbors
d2_sqrd_nbrs_test = d2_nbrs_test**2
D2_test = np.sum(d2_sqrd_nbrs_test, axis = 1)
D2_log_test = np.log(D2_test)
```

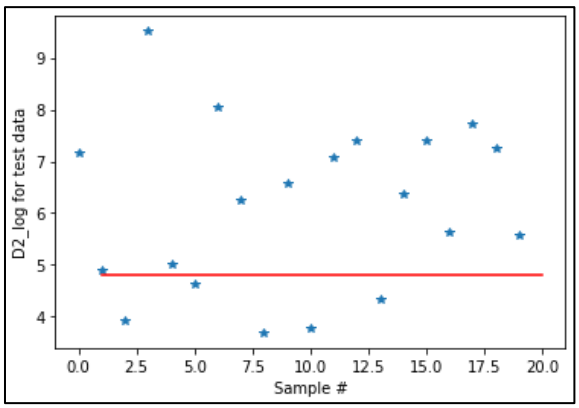
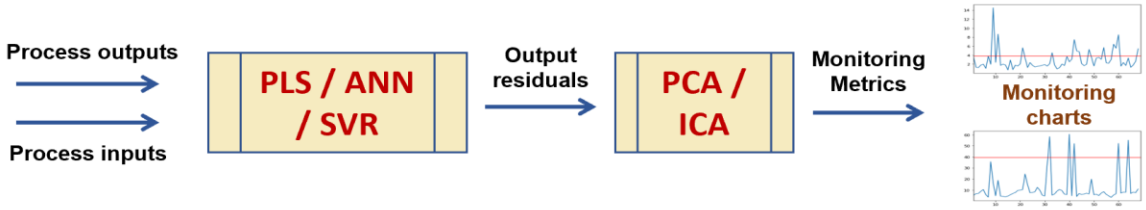


Figure 10.10: FD-KNN monitoring chart for metal-etch test samples

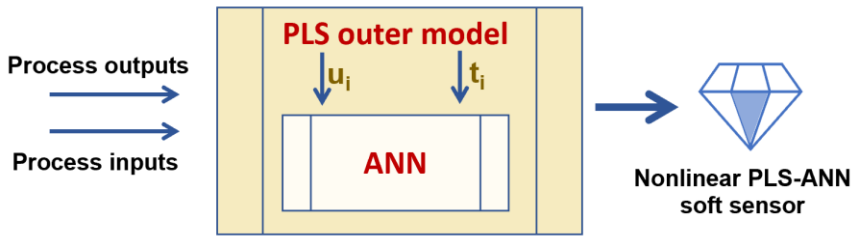
`

10.5 Combining ML Techniques

In this book, we have looked at several different ML techniques. We attempted to gain conceptual understanding of these methods and not just learn how to apply them via Sklearn modules. This enables us to tweak the standard algorithms of these techniques to suit our specific needs. An area where this can prove very useful is when you need to combine the different ML techniques together to leverage their respective strengths. A common combination scheme is to use PCA for feature extraction and any then apply other method on the extracted features. However, if you dig deeper into the ML for PSE literature, you can find several other interesting combinations. One such scheme (also called external analysis[55] [56]) is illustrated below. In this scheme, the impact of normal plant operating condition changes (as indicated by process inputs) are removed from the output variables before the application of PCA or ICA-based fault detection. The corresponding authors have shown that fault detection performance increases significantly through this scheme.



The secret behind the success of external analysis is that nonlinear or dynamic characteristics of process data can be handled by specialized modeling tools and then output residuals (difference between output measurements and predictions) can be handled efficiently by PCA. Another noteworthy scheme includes the integration of artificial neural networks (ANN) and PLS[57]. The powerful dimension reduction capabilities of PLS are combined with nonlinearity modeling capabilities of ANN to obtain a high-fidelity soft-sensor model.



[55] Kano et al., Evolution of multivariate statistical process control: application of independent component analysis and external analysis, Computers & Chemical Engineering, 2004

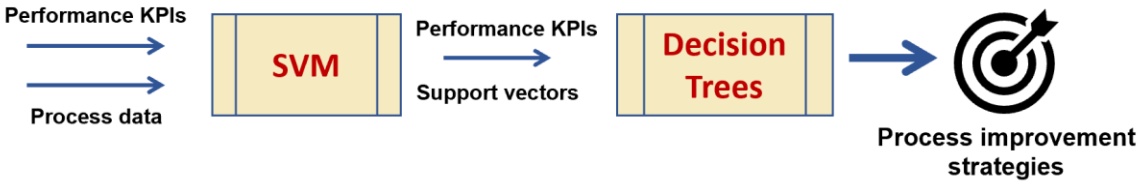
[56] Ge et al., Robust online monitoring for multimode processes based on nonlinear external analysis, Industrial Engineering & Chemistry Research, 2008

[57] Qin & McAvoy, Nonlinear PLS modeling using neural networks, Computers & Chemical Engineering, 1992

`

Jemwa & Aldrich[58] have proposed a scheme that combines SVM and decision trees. This scheme involves judicious use of support vectors to reduce the number of samples that goes into building decision trees model. The final result comprises of optimal operating strategies of the following type

$$\text{If} \quad 0.6 \leq reactant\ mole\ fraction \leq 0.9$$
$$Then\ \ keep\ reactant\ temperature\ within\ 270 - 290\ K$$



These were just a few examples to illustrate how very powerful models can be built by careful and conceptually sound combination of different ML techniques. Note that these combination strategies are more ad-hoc and situation-dependent compared to the more systematic ensemble learning techniques. The key takeaway is that as long as you are aware of the underlying mechanism and the combination makes theoretical sense, it does not hurt to try out various combinations if your problem cannot be satisfactorily handled by a standalone technique.

Summary

With this chapter we have come to the end of our journey into the world of classical ML methods. We hope that you already feel empowered and your interest in machine learning has increased. The methods you have learnt so far should enable you to tackle majority of the problems you will face as process data scientists. With the classical techniques conquered, let us now get ready to step into the amazing world of artificial neural networks.

[58] Jemwa & Aldrich, Improving process operations using support vector machines and decision trees, AIChE Journal, 2005