Chapter 11

Multiclass and Bias-Variance decomposition

Multiclass classification. Bootstrapping, Bias-Variance decomposition

11.1 Multiclass classification

So far we have always focused on binary classification but classification problems with more than two classes are common. A classical example is the classification of flowers of three species of the *Iris* genus: *Iris setosa*, *Iris versicolor* and *Iris virginica*, shown in Figure 11.1. The data set describes each flower with four features: sepal length and width and petal length and width[1].

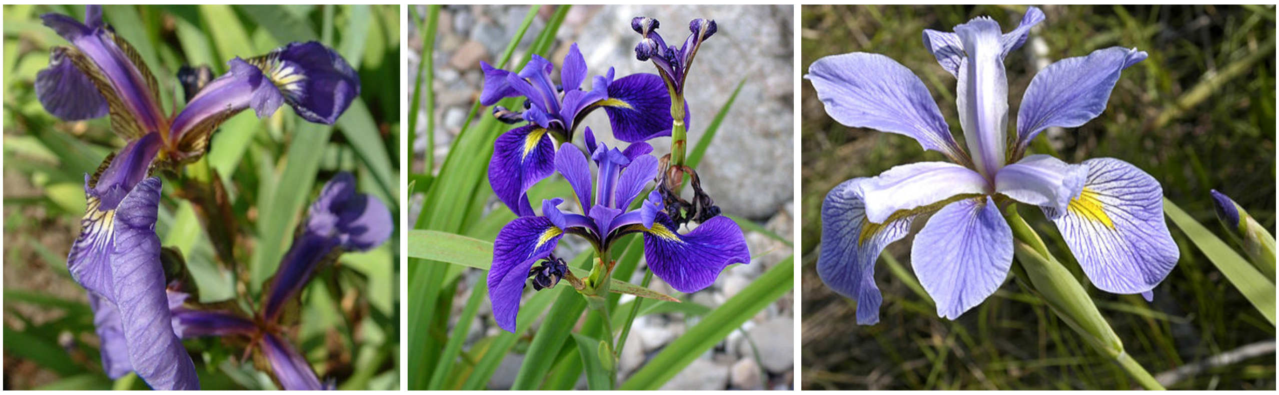


Figure 11.1: Iris flowers: setosa, versicolor and virginica. Images CC BY-SA: Szczecinkowaty; Gordon abd Robertson; Mayfield

For classifiers like Naïve Bayes or k-Nearest Neighbours the number of classes makes no difference, since the classifier is used in exactly the same way. For Naïve Bayes, we choose the class that maximizes the conditional probability of the feature values:

$$C^{Nave\ Bayes} = \underset{k\in\{0,1,\ldots,K\}}{\operatorname{argmax}} \ln p(C_k) + \sum_{j=1}^{N} \ln p(x_j|C_k)$$

and for k-Nearest Neighbours we classify each new point according to the majority in its k-neighbourhood. Figure 11.2 illustrates the data set and its use for creating a k-NN classifier.

[1] The data set can be downloaded from the MIST repository: `https://archive.ics.uci.edu/ml/datasets/Iris`

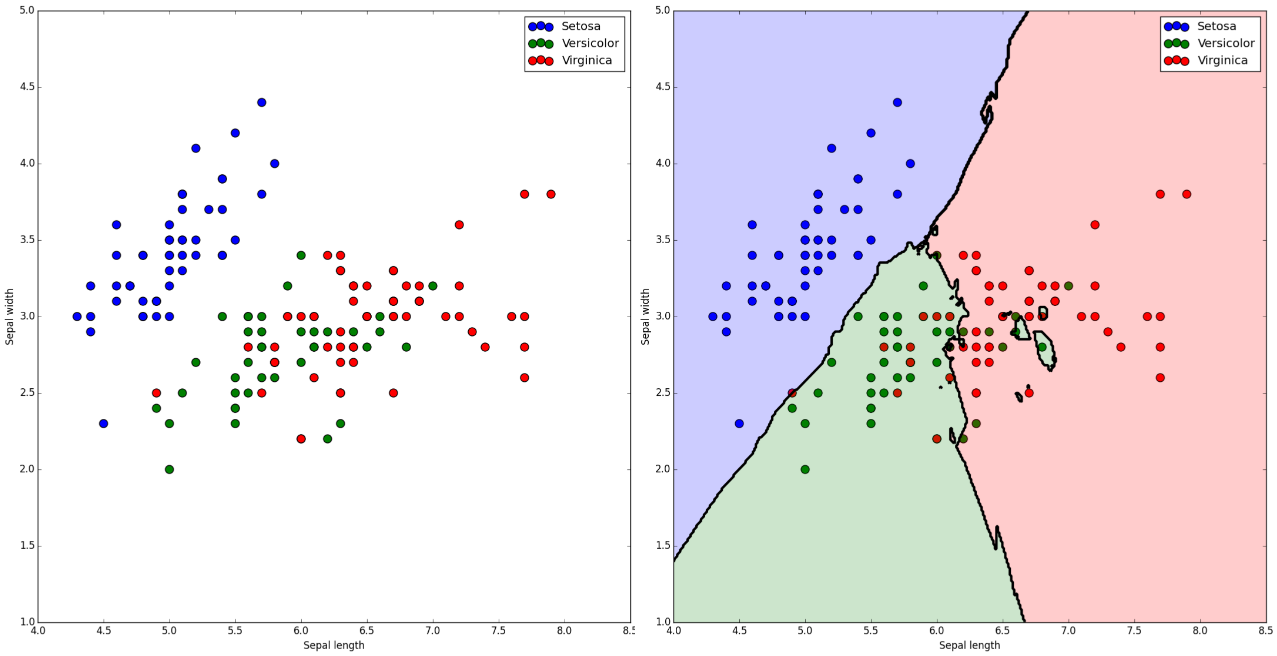


Figure 11.2: The left panel shows the Iris data set projected on the sepal length and width features. The right panel shows the classification with k-NN. Each point is classified according to the majority of the classes of neighbouring points.

However, for classifiers based on binary discriminant functions, like Logistic Regression, perceptrons or SVM, extension to more than two classes requires some meta-algorithm to obtain the necessary binary discriminants. One possible way of separating K classes is to train $K-1$ binary classifiers, each one to discriminate between one class and all other examples. This is an example of a *one versus the rest* classification scheme. An example is assigned to the class corresponding to the classifier that identifies it as being in the classifier's class, or to class K if none of the $K-1$ classifiers identifies it. Figure11.3 shows this process. One problem with the $K-1$ *one versus the rest* classification scheme is that there are ambiguous results wherever classifiers overlap. In this example, there are points that are classified both as *setosa* and *versicolor*.

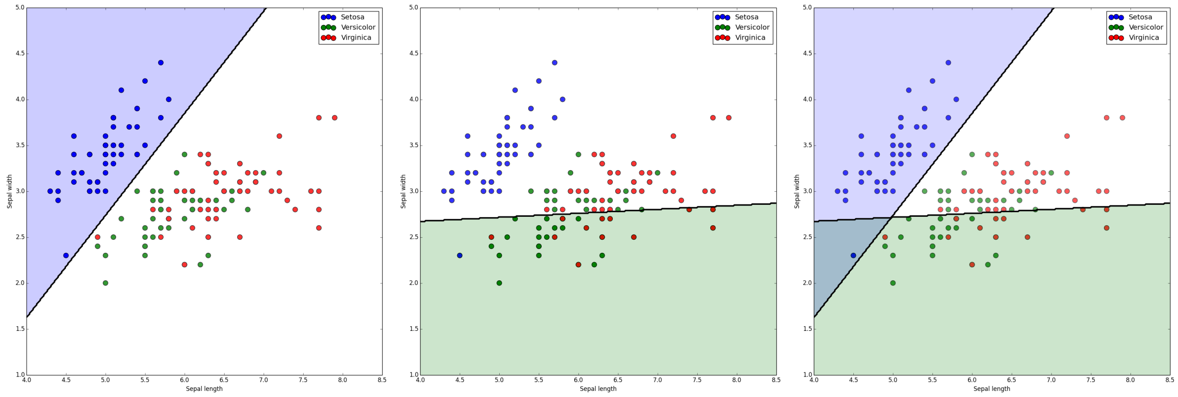


Figure 11.3: One versus the rest with K-1 classifiers. The first two classifiers distinguish, respectively, setosa and versicolor examples from all others. The last panel shows the final classification.

An alternative is to train binary classifiers to distinguish between all pairs of classes by training $K(K-1)/2$ classifiers and then classifying each new example with a majority vote, assigning it to the class with the largest number of votes among the classifiers. However, with this approach there are also ambiguous classifications whenever there is an equal number of votes for more than one class.

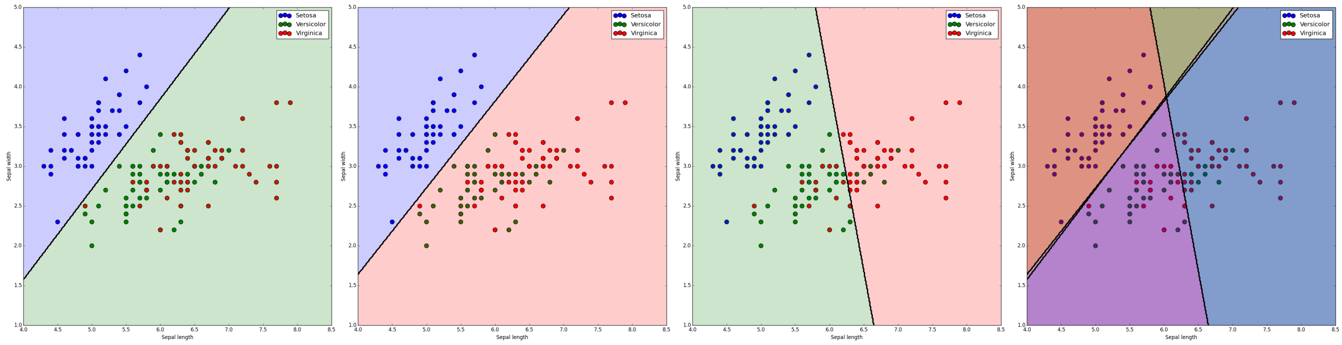


Figure 11.4: One versus one classification schemes. After training $K(K-1)/2$ classifiers, points are classified by majority vote.

A better alternative is to use a *one versus the rest* classification scheme with K classifiers. If each classifier can provide a value for the decision function, points can be classified according to the maximum of the decision functions of the K classifiers. This solves the problem of ambiguous classification, as illustrated in Fig 11.5. However, for the *one versus the rest* scheme it is necessary to train each classifier with an unbalanced sample in which the majority of points fall outside the respective class. For example, if our training set has 10 evenly balanced classes, then each of the 10 classifiers will have only 10% of the points in the positive class and 90% in the negative class. Furthermore, the decision function values for the different one-vs-rest classifiers may not be directly comparable, and these differences may affect the performance of this multiclass classification heuristic.

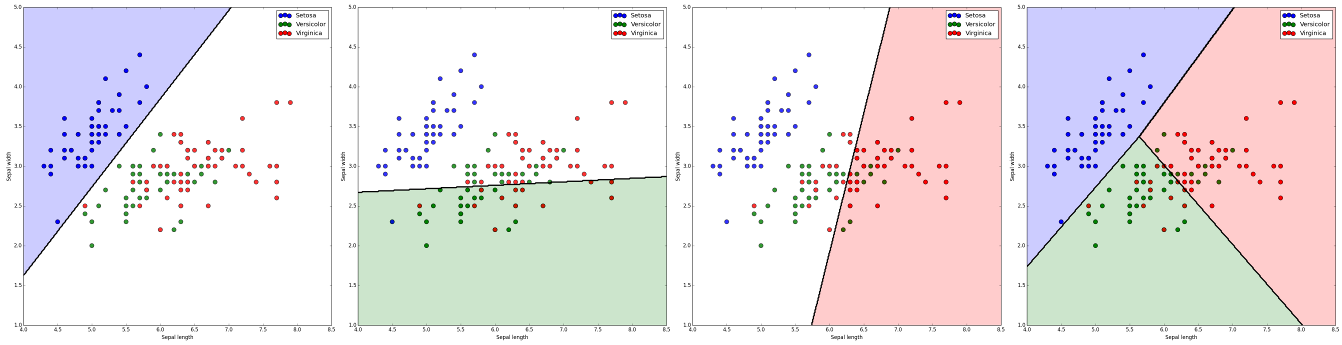


Figure 11.5: One versus the rest classification scheme with K classifiers. Points are classified by the maximum value of the decision function.

Some classifiers allow specific alternatives to multiclass classification. For example, Logistic Regression can be extended to multiclass classification by fitting K discriminant hyperplanes simultaneously, by using the *cross entropy* of all classes and predictions considering, for each of the K discriminants, that the points belong to class 1 if they are in class k and to class 0 otherwise:

$$p(T|w_1, ..., w_K) = \prod_{n=1}^{N} \prod_{k=1}^{K} p(C_k|\phi_n)^{t_{nk}} = \prod_{n=1}^{N} \prod_{k=1}^{K} y_{nk}^{t_{nk}}$$

In this expression, the $t_n k$ matrix gives this *one vs rest* classes, assigning a 1 to all elements in class k and 0 otherwise. In practice, we minimize the logarithm of the *cross entropy* as an error function.

$$E(w_1, ..., w_K) = -\sum_{n=1}^{N} \sum_{k=1}^{K} t_{nk} \ln y_{nk}$$

With the `sklearn` library, we can use either the *one vs rest* classification scheme (`'ovr'`) or the *cross entropy* one (`'multinomial'`) in the `LogisticRegression` class:

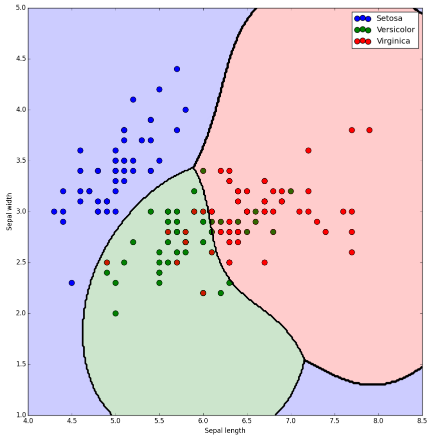
```
from sklearn.linear_model import LogisticRegression

#One versus rest, max
logreg = LogisticRegression(C=1e5,multi_class='ovr')
logreg.fit(X, Y)
#Cross entropy
logreg = LogisticRegression(C=1e5,multi_class='multinomial')
logreg.fit(X, Y)
```

The multilayer perceptron also can be easily adapted to multiclass classification by having one output neuron for each class and training the MLP to output a 1 on the neuron corresponding to the class of the example and a 0 on all other output neurons. The activation function on the output layer, in this case, is usually the *softmax* function, mapping a vector of K input values into a vector of K values all between 0 and 1 and adding up to 1. This can be interpreted as the probability of the example belonging to each class.

$$\sigma : \mathbb{R}^K \rightarrow [0,1]^K \qquad \sigma(\vec{x})_j = \frac{e^{x_j}}{\sum\limits_{k=1}^{K} e^{x_k}} \qquad \sigma_j \in [0,1]; \sum_{k=1}^{K} \sigma_k = 1$$

For binary classifiers in general, the `sklearn` library offers a useful class to perform *one versus rest* classification by training K classifiers and classifying each example according to the maximum of the decision function:



```
from sklearn.multiclass import OneVsRestClassifier
ovr = OneVsRestClassifier(SVC(kernel='rbf',
                              gamma=0.7, C=10))
ovr.fit(X, Y)
ovr.predict(test_set)
```

Figure 11.6: One-vs-rest classification of the Iris data set using SVM.

To use this class, we need only provide it with the class of the binary classifier, which must implement the `fit` and `decision_function` methods. The `fit` method of `OneVsRestClassifier` generates K classifiers, training each to distinguish one class from all others. Then the `predict` method returns the class corresponding to the classifier that outputs the largest value in the `decision_function`. Figure 11.6 shows the result of this process using SVM on the Iris dataset.

11.2 Bias and Variance

Statistically, *bias* is the difference between the expected value of an estimator and the true value being estimated. Thus, the *bias* of a model at some point is the difference between the true value and the expected prediction of the model for that point. The *bias* for the model is the average of the *bias* values measured for all points::

$$bias_n = (\bar{y}(x_n) - t_n)^2 \qquad bias = \frac{1}{N}\sum_{n=1}^{N}(\bar{y}(x_n) - t_n)^2$$

Figure 11.7 shows an example of a model that cannot adequately fit the data. The estimates for the point marked as a large blue circle are all tendentiously above the true value and thus there is a difference between the average and the true value.

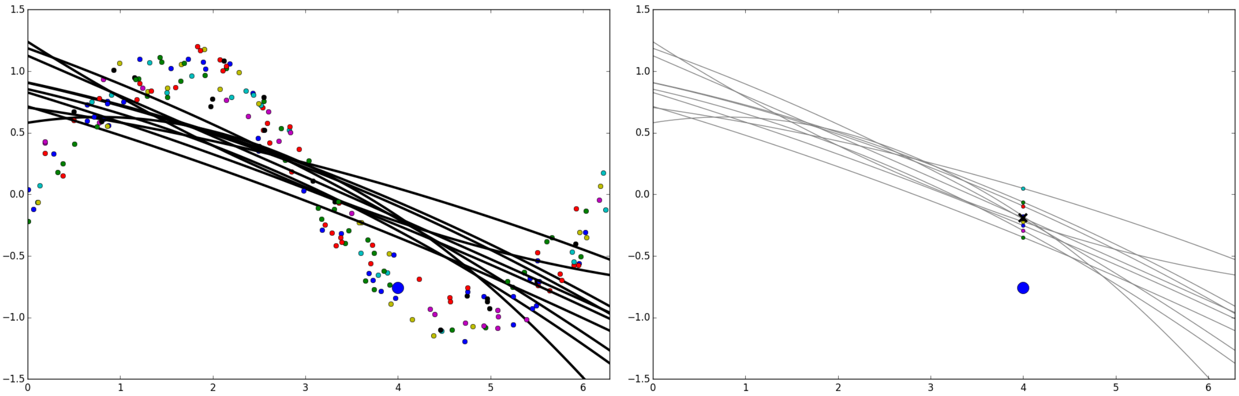


Figure 11.7: This model cannot adjust to the data and thus has a large *bias* in some points.

In statistics, variance is a measure of the dispersion of values. Applying this concept to a regression model, the *variance* of the model at some point is the expected variance of the predicted values for that point when the model is trained over any data set. The *variance* for the model is the average of the variances for all points. To estimate the variance of a point and on N points of a model trained on M data sets, we compute:

$$\frac{1}{M}\sum(\bar{y}(x_n) - y_m(x_n)) \qquad var = \frac{1}{NM}\sum_{n=1}^{N}\sum_{m=1}^{M}(\bar{y}(x_n) - y_m(x_n))^2$$

where $\bar{y}(x_n)$ is the average of the predictions for point x_n. Figure 11.8 shows a model that overfits the data, which results in a large *variance*, showing that, for the point marked as a large circle, the predictions of individual hypotheses are spread in a broad range around their average.

11.3 Bootstrapping

To estimate the *bias* and *variance* of a model we need to train the model over different training sets. However, in general we only have one training set, and so we need to resample our training set in order to generate different sets from the same distribution. One widely used resampling method is *bootstrapping*, which consists of creating replicas of the original set by sampling at random with reposition until we have a new set with the same number of points as the original. On average, the replica set will have around two thirds of the points of the original set, with some repetitions, leaving

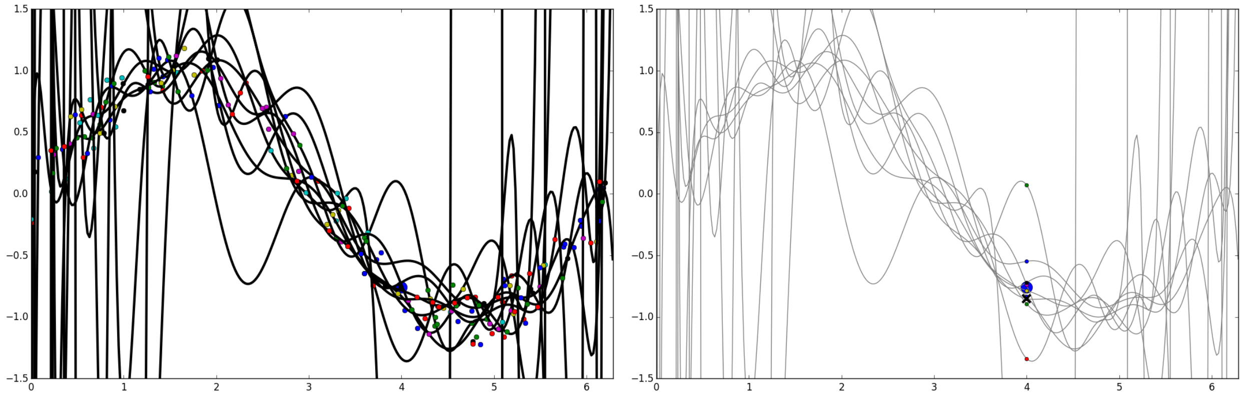


Figure 11.8: This model overfits the data and thus has a large *variance* in some points.

out about one third of the original points. With this method we can generate a large number of data sets and use them to estimate the *bias* and *variance* of our model.

The function below shows how we can create a number of replicas from a training set data matrix using bootstrapping. For each replica, the function generates a random vector with indexes of the rows of the data matrix to be copied to the replica. This random vector will contain repetitions (sampling with reposition), so some points will be left out and others may be repeated.

```
def bootstrap(samples,data):
    train_sets = np.zeros((samples,data.shape[0],data.shape[1]))
    for sample in range(samples):
        ix = np.random.randint(data.shape[0],size=data.shape[0])
        train_sets[sample,:] = data[ix,:]
    return train_sets
```

With the replicas, we can now estimate the bias and variance of a model by training on each replica and evaluating the errors outside the training set, using a separate test set (or validation set if we use it to select a model). For this example, we'll use polynomial regression models. We start by creating and filling a matrix with all the predictions of all polynomials fit to all the replicas of the training set. This is the `predicts` matrix in the source code below.

```
def bv_poly(degree, train_sets, test_set):
    samples = train_sets.shape[0]
    predicts = np.zeros((samples,test_set.shape[0]))
    for ix in range(samples):
        coefs = np.polyfit(train_sets[ix,:,0],
                           train_sets[ix,:,1],degree)
        predicts[ix,:] = np.polyval(coefs,test_set[:,0])
    mean_preds = np.mean(predicts,axis=0)
    bias_per_point = (mean_preds-test_set[:,-1])**2
    bias = np.mean(bias_per_point)
    var_per_point = np.mean((predicts-mean_preds)**2,axis=0)
    var = np.mean(var_per_point)
    return bias,var
```

Then we compute the average predicted values over all predictions and use this vector, with one mean prediction for each point in the test set, to predict the bias values for all points in the test set. The bias will be the mean of these values. For the variance the procedure is similar, but now the variance for each point in the data set is given by the average quadratic distance between each individual prediction and the mean prediction value.

Since we are estimating *bias* and *variance* on each hypothesis with points that were not used to train that particular hypothesis, our estimates are unbiased. This is why it is important to avoid using the same examples for training and evaluating *bias* and *variance*.

11.4 Bias-variance decomposition

With a quadratic error function, the error is the expected square of the difference between the predicted values and the true values. This the loss function that is generally used in regression, so in regression we can decompose the error into:

$$E\left((y-t)^2\right) = (E(y)-E(t))^2 + E\left((y-E(y))^2\right) + E\left((t-E(t))^2\right)$$

The term $(E(y)-E(t))^2$ is the square of the difference between the expected prediction and the true value, which is the *bias*. $E\left((y-E(y))^2\right)$ is the *variance* and $E\left((t-E(t))^2\right)$ is the expected squared error between the expected value for each point and the value in the training set. This last term is the *noise* in our data set, which we will generally assume to be zero. Thus, assuming there is no random noise in our data, we can decompose the quadratic error into a sum of *bias* and *variance*. Figure 11.9 shows this decomposition used to examine the source of the error for polynomials of different degrees.

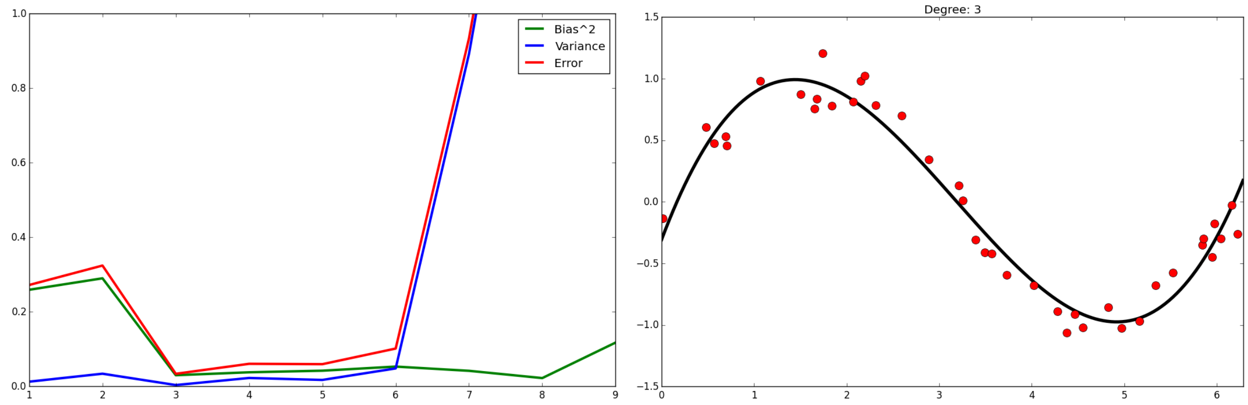


Figure 11.9: The left panel plots the *bias*, *variance* and total error (assuming zero noise). The right panel shows the result of training the best model.

As we can see in Figure 11.9, there is a trade-off between *bias* and *variance*. If the model is underfitting, unable to adjust to the data, *bias* is the largest component of the error. But when overfitting, *variance* becomes the dominant factor. The optimal choice is the one that minimizes the total contribution of *bias* and *variance*.

So far, we've seen how to decompose the error into *bias* and *variance* for models using a quadratic error function. However, although this is the norm with regression problems, a quadratic error function is not ideal for classifiers. In these cases, we generally evaluate the error using a 0/1 loss function, giving an error of 1 if the predicted class is different from the true class, or 0 if they are equal. With this error function, the decomposition into *bias* and *variance* is different. First of all, the *main prediction* in this case is the prediction that is most common, or the mode of the predictions, considering all hypotheses. So the *bias* for example i with a 0/1 loss function is the error of the *main prediction* with respect to the true class of point i:

$$bias_i = L(Mo(y_{i,m}), t_i)$$

where $Mo(y_{i,m})$ is the mode of predictions for point i over all m hypotheses and L is the loss function returning 0 if the values are equal or 1 if the values differ. The *variance* is the expected error of all predictions for example i with respect to the *main prediction*:

$$var_i = E\left(L(Mo(y_{i,m}), y_{i,m})\right)$$

So far, this is essentially the same as we saw for the quadratic error function used in regression problems. However, the error decomposition is fundamentally different because whether the *variance* increases or decreases the error depends on the *bias* for that error. If the *bias* is 0, meaning the *main prediction* is correct, then the *variance* increases the error, since any deviation from the main prediction increases the error. On the other hand, if the *bias* is 1, then this means the *main prediction* is incorrect and so any deviation from this prediction will decrease the expected total error. Thus, the error decomposition into *bias* and *variance* (assuming no noise in the data) is:

$$E\left(L(t, y)\right) = E\left(B(i)\right) + E\left(V_{unb.}(i)\right) \breve{}\, E\left(V_{biased}(i)\right)$$

where $V_{unb.}$ is the variance for points with *bias* of 0 and V_{biased} corresponds to the variance for points with *bias* of 1. Or, alternatively, we can consider the *variance* to be the net variance $E_x\left(V_{unb.}(i)\right) - E_x\left(V_{biased}(i)\right)$.

As an example, we'll decompose the *bias* and *variance* of K-Nearest Neighbours classifiers (assuming the data has no noise). We start, as in the regression example, by fitting the classifier to each of the replicas and storing the predictions.

```
def bv_knn(neighs, train_sets, test_set):
    samples = train_sets.shape[0]
    predicts = np.zeros((samples,test_set.shape[0]))
    for ix in range(samples):
        sv = KNeighborsClassifier(n_neighbors=neighs)
        sv.fit(train_sets[ix,:,:-1],train_sets[ix,:,-1])
        predicts[ix,:] = sv.predict(test_set[:,:-1])
    main_preds = np.round(np.mean(predicts,axis=0))
    bias_per_point = np.abs(test_set[:,-1]-main_preds)
    var_per_point = np.mean(np.abs(predicts-main_preds),axis=0)
    u_var = np.sum(var_per_point[bias_per_point == 0])/test_set.shape[0]
    b_var = np.sum(var_per_point[bias_per_point == 1])/test_set.shape[0]
    print(u_var,b_var)
    return bias,u_var-b_var
```

Next, we compute the main prediction for each example, which is the more common prediction. This can be done by rounding the mean of all predictions for each example to 0 or 1. The *bias* is then computed from the difference between the main prediction and the true class, which can be 0 or 1, and the variance from the fraction of predictions that differ from the main prediction. Finally we average the *bias* of each point over all the points to estimate the *bias* of the model. Since this is a classification problem, we must distinguish the *variance* contributed by the unbiased points from the *variance* contributed by the biased points, since these affect the error differently. Thus we decompose these two contributions to compute the net variance.

Figure 11.10 shows the *bias* and *variance* decomposition of a K-NN classifier with the number of neighbours varying from 1 through 17. When the classifier averages classes over a larger neighbourhood it has a larger *bias* and tendentiously smaller net *variance*. With a smaller number of neighbours, the *bias* decreases but the *variance* starts increasing. The right panel shows the classifier that best balances *bias* and *variance*, with 5 neighbours.

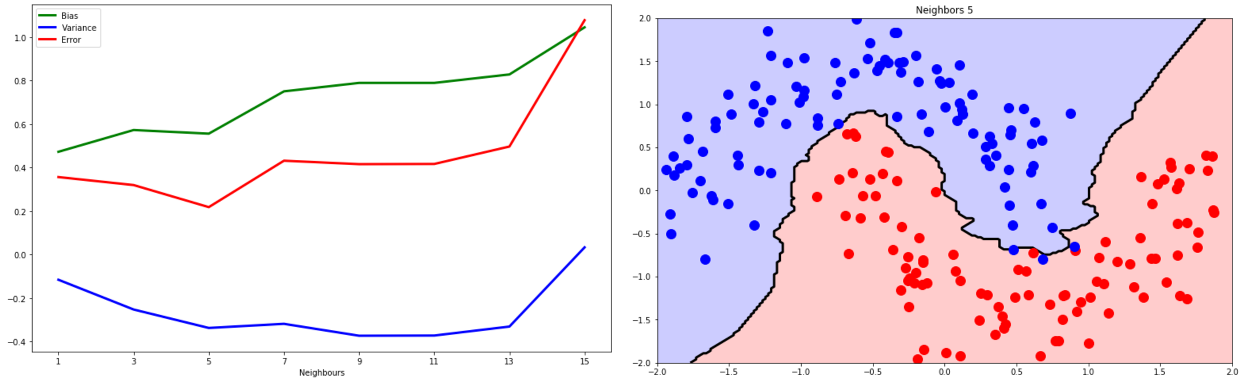


Figure 11.10: The left panel plots the *bias*, *variance* and total error (assuming zero noise) for different values of the number of neighbours considered. The right panel shows the result of training the best model with 5 neighbours.

11.5 Further Reading

1. Alpaydin [2], Section 4.3
2. Bishop [4], 4.1.2, 4.3.4, 7.1.3
3. (Optional: Valentini and Dietterich. Bias-variance analysis of support vector machines for the development of SVM-based ensemble methods [21])
4. (Optional: Domingos, P. A unified bias-variance decomposition [8])

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.