

2.4 Ensemble Methods

This section presents techniques for improving classification accuracy by **aggregating the predictions of multiple classifiers**. These techniques are known as **ensemble methods**. An ensemble method constructs a set of base classifiers from training data and performs a classification. There are a few kinds of ensemble methods, many of them **outperform** a single classifier.

We will present the rationale of the ensemble methods using majority vote and later move on to the topics of **bagging**, **random forests**, and **boosting** – more practical and powerful ensemble learning techniques to construct classification models. Most of the techniques presented herein are also applicable to regression problems.

2.2.1 Motivation: Majority Vote

Let us walk through an example by taking a **majority vote** on the predictions made by each base classifier.

Example: Consider an ensemble of 3 binary classifiers, each of which has an error rate¹ of $e = 0.35$. The ensemble classifier predicts the class label of a test example by taking a majority vote on the prediction made by the base classifiers. Assume that the base classifiers are independent of each other and the ensemble makes a wrong prediction only if more than half of the base classifiers predict incorrectly.

The following excel table lists all the possibilities.

A	B	C	Error Rate	
0.65	0.65	0.65	0.274625	
0.35	0.65	0.65	0.147875	
0.65	0.35	0.65	0.147875	
0.65	0.65	0.35	0.147875	
0.35	0.35	0.65	0.079625	
0.65	0.35	0.35	0.079625	
0.35	0.65	0.35	0.079625	
0.35	0.35	0.35	0.042875	
			1	0.28176
				Ensemble Error Rate

¹ Error rate means how often it is wrong. For a binary classifier, error rate is defined as $\varepsilon = \frac{\text{Number of wrong prediction}}{\text{Total Number of prediction}}$.

Formally, the error rate for each combination can be computed by:

$$\varepsilon_i = \binom{3}{i} e^i (1 - e)^{3-i}$$

$\binom{n}{k}$ is read ***n* choose *k***. $\binom{n}{k}$ is called binomial coefficient and can be computed as $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.

And the ensemble error rate through majority vote is:

$$\varepsilon_{\text{ensemble}} = \sum_{i=2}^3 \binom{3}{i} e^i (1 - e)^{3-i} = 0.28175$$

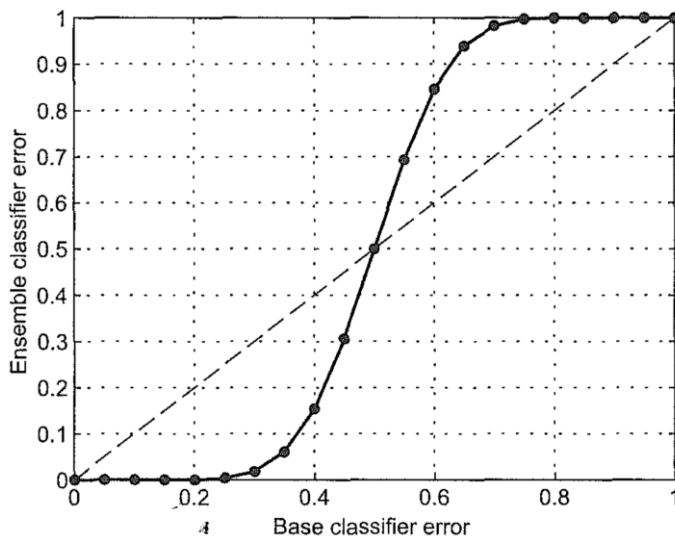
which is slightly better than the error rate of a base learner $e = 0.35$.

Example: Consider an ensemble of 25 binary classifiers, each of which has an error rate of $\varepsilon = 0.35$. The ensemble classifier predicts the class label of a test example by taking a majority vote on the prediction made by the base classifiers. Assume that the base classifiers are independent of each other and the ensemble makes a wrong prediction only if more than half of the base classifiers predict incorrectly.

(1) What is the error rate of the ensemble for the case with 13 base classifiers predicting incorrect answer?

(2) What is the error rate of the ensemble method?

Figure below shows the error rate of an ensemble of 25 binary classifiers through the majority vote ($\varepsilon_{\text{ensemble}}$) for different base classifier error rate (ε).



$e > 0.5$ 發散
 < 0.5 收斂
 $= 0.5$ 不變

Q: what have you observed?

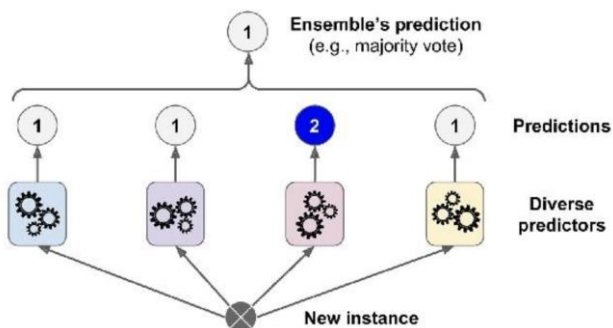
A:

Remark: Two necessary conditions for an ensemble classifier to perform better than a single classifier:

1. The base classifier should be independent of each other.
2. The base classifier should do better than a classifier that performs random guessing.

In practice, it is difficult to ensure total independence among the base classifiers. Nevertheless, even that the base classifiers are correlated, improvements on classification accuracy has been observed in the ensemble methods. In fact, the winning solutions in machine learning competitions often involve several ensemble methods.

Python example: majority vote



Let us use `make_moon` to make two interleaving half circles and creates and trains a voting classifier in `Scikit-Learn`, composed of three diverse classifiers. Let us first create and plot the data:

```

%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

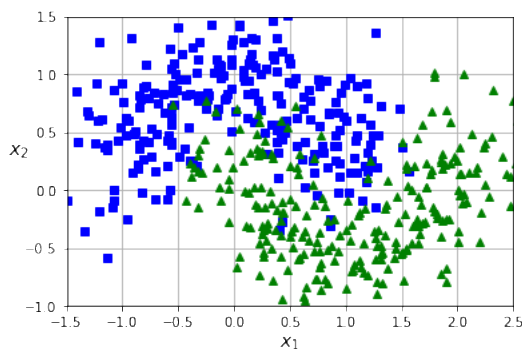
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=500, noise=0.30, random_state=42)

def plot_dataset(X, y, axes):
    plt.plot(X[:, 0][y==0], X[:, 1][y==0], "bs")
    plt.plot(X[:, 0][y==1], X[:, 1][y==1], "g^")
    plt.axis(axes)
    plt.grid(True, which='both')
    plt.xlabel(r"$x_1$", fontsize=14)
    plt.ylabel(r"$x_2$", fontsize=14, rotation=0)

plot_dataset(X, y, [-1.5, 2.5, -1, 1.5])
plt.show()

```



Now the training and testing:

```

X_train, X_test, y_train, y_test = train_test_split(X, y,
random_state=42)
from sklearn.ensemble import VotingClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

log_clf = LogisticRegression(random_state=42)
rnd_clf = RandomForestClassifier(random_state=42)
svm_clf = SVC(random_state=42)

voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='hard')

voting_clf.fit(X_train, y_train)
for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
    clf.fit(X_train, y_train)

```

```
y_pred = clf.predict(X_test)
print(clf.__class__.__name__, accuracy_score(y_test, y_pred))
```

```
LogisticRegression 0.864
RandomForestClassifier 0.872
SVC 0.888
VotingClassifier 0.896
```

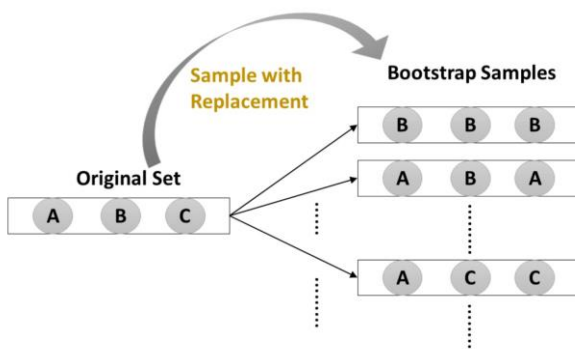
There you have it! The voting classifier slightly outperforms all the individual classifiers. You can download the above source code `Ch4_MajorityVote.ipynb` from the course website.

2.2.2 Bagging

One way to get a diverse set of classifiers is to use very different training algorithms, as just discussed in Section 2.2.1. Another approach is to use the same training algorithm for every feature, but to train them on different random subsets of the training set. This approach is called **bagging** (short for **bootstrap** (自助抽樣) **aggregating**) based on the **bootstrap**, a widely applicable and extremely powerful statistical tool.

The basic idea of **bootstrap** or bootstrapping is that inference about a population from sample data (sample \rightarrow population) can be modelled by resampling the sample data and performing inference about a sample from resampled data (resampled \rightarrow sample). In short, bootstrap resamples the data with replacement as illustrated in the figure below. When doing properly, we will have:

sample \rightarrow population \approx resampled \rightarrow sample



Bagging refers to bootstrap aggregating. It is a general-purpose procedure for reducing the variance of a statistical learning method. Basically, we resample the data with replacement and then train a classifier on the newly sampled data. Then, we combine the outputs of each of the individual classifiers using a majority-voting scheme or other similar schemes.

Theoretical Minimum for Bagging

Let us consider a given a set of n **independent** (or uncorrelated) observations Z_1, Z_2, \dots, Z_n , each with variance σ^2 . The variance of the mean \bar{Z} of the observations is given by $\frac{\sigma^2}{n}$. **In other words, averaging a set of observations reduces variance.**

Hence a natural way to reduce the variance and hence increase the prediction accuracy of a statistical learning method is to take many training sets from the population, build a separate prediction model using each training set, and average the resulting predictions. In other words, we could calculate $g^{(1)}(x), g^{(2)}(x), \dots, g^{(B)}(x)$ using B separate training sets, and average them in order to obtain a single low-variance statistical learning model,

$$g_{\text{avg}}(x) = \frac{1}{B} \sum_{b=1}^B g^{(b)}(x)$$

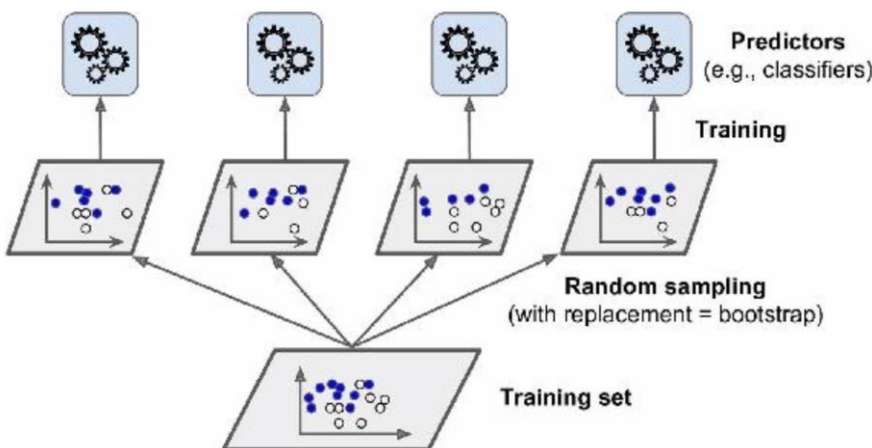
Q: Is this (to take many training sets from the population) practical?

A:

Instead, we can **bootstrap**, by taking repeated samples from the single training data set. In this approach we generate B different bootstrapped training data sets. We then train our method on the b^{th} bootstrapped training set in order to get $g^{(*b)}(x)$, and finally average all the predictions to obtain

$$g_{\text{bag}}(x) = \frac{1}{B} \sum_{b=1}^B g^{(*b)}(x)$$

This is called **bagging**. In other words, bagging allows training instances to be sampled several times. This sampling and training process are illustrated in figure below.



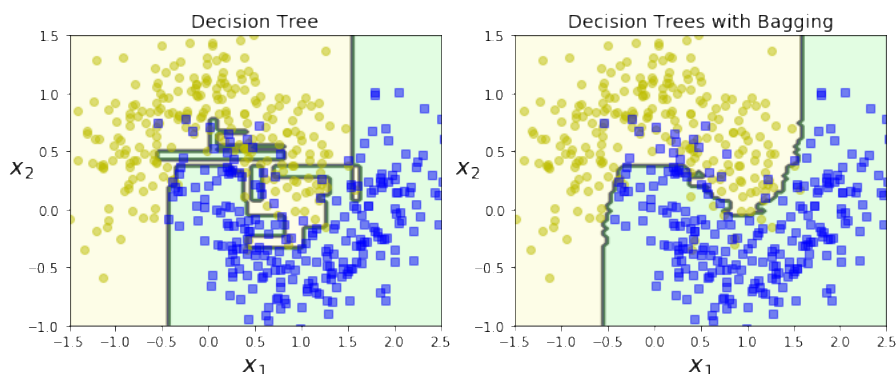
Python Example

Scikit-Learn offers a simple API for bagging with the `BaggingClassifier` class (or `BaggingRegressor` for regression). The following code trains an ensemble of 500 Decision Tree classifiers, each trained on 100 training instances, randomly sampled from the 500 data generated from `make_moon` with replacement. The `n_jobs` parameter tells Scikit-Learn the number of CPU cores to use for training and predictions (-1 tells Scikit-Learn to use all available cores):

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

bag_clf = BaggingClassifier(
    DecisionTreeClassifier(random_state=42), n_estimators=500,
    max_samples=100, bootstrap=True, n_jobs=-1, random_state=42)
bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)
```

Figure below compares the decision boundary of a single Decision Tree with the decision boundary of a bagging ensemble of 500 trees (from the preceding code).



Q: what have you observed?

A:

You can download the complete source code `Ch4_Bagging.ipynb` from the course website.

2.2.3 Random Forests

An ensemble of randomized decision trees is known as a **random forest**. Random forests attempt to improve the generalization performance by constructing an ensemble of *decorrelated* decision trees.

Random forests build on the idea of **bagging** to use different bootstrap samples of a single training data for learning decision trees.

Remark: a key distinction of random forests from bagging is that at every internal node of a tree, the best splitting criterion is chosen among a small set of randomly selected features. In this way, the random forests construct ensembles of decision trees by not only manipulating training instances (*aka*, bagging), but also the input features.

Theoretical Minimum: Random Forests and Decorrelating Decision Trees

Random forests provide an improvement over bagged trees by way of a small tweak that decorrelates the trees and let us study the rationale behind.

As in bagging, we build a number of decision trees on bootstrapped training samples. But when building these decision trees, each time a split in a tree is considered, a random sample of m features is chosen as split candidates from the full set of p features. The split is allowed to use only one of those m features. A fresh sample of m features is taken at each split, and typically we choose $m \approx \sqrt{p}$. For example, if we have 13 features in total, each time we will only use 4 randomly selected from the 13 features and only use 1 from the 4 features to split the tree.

We now explain the clever rationale behind random forests. Suppose that there is one very strong feature in the data set, along with a number of other moderately strong features. What will likely happen when we do the top split?

A:

Q: What is the consequence?

A: 强化预测力

Random forests overcome this problem by forcing each split to consider only a subset of the features. Therefore, on average $(p - m)/p$ of the splits will not even consider the strong feature, and so other features will have more of a chance. **We can think of this process as decorrelating the trees**, thereby making the average of the resulting trees less variable and hence more reliable.

Python Example: Random Forest for Classifying Digits

To demonstrate the capability of random forests, let's consider one piece of the optical character

recognition problem: **the identification of hand-written digits**. In the wild, this problem involves both locating and identifying characters in an image. Here we'll take a shortcut and use `Scikit-Learn`'s set of pre-formatted digits, which is built into the library.

Loading and visualizing the digits data

We'll use `Scikit-Learn`'s data access interface and take a look at this data:

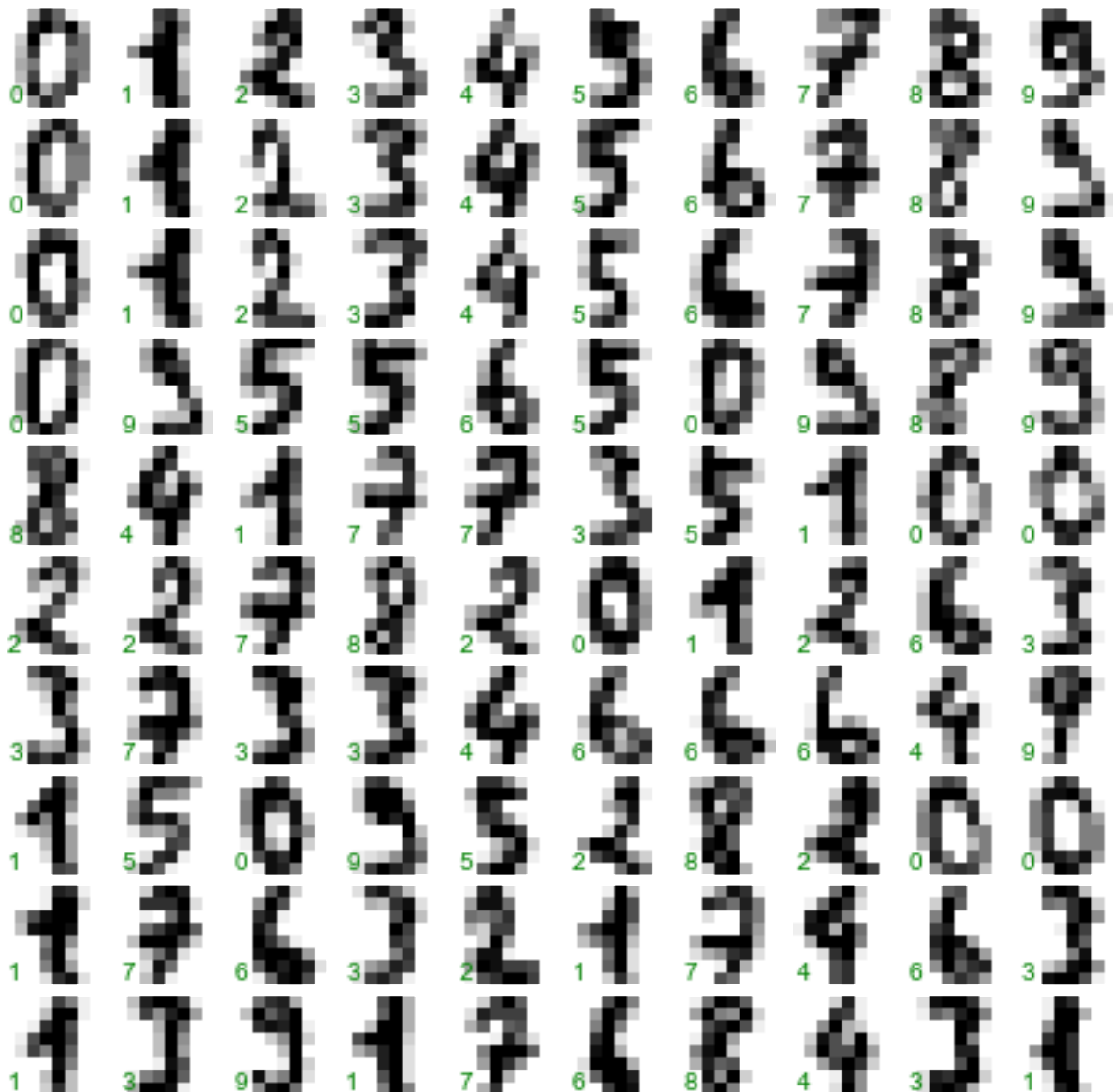
```
from sklearn.datasets import load_digits
digits = load_digits()
digits.images.shape
(1797, 8, 8)
```

The images data is a three-dimensional array: 1,797 samples each consisting of an 8×8 grid of pixels. Let's visualize the first hundred of these:

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()

fig, axes = plt.subplots(10, 10, figsize=(8, 8),
                        subplot_kw={'xticks':[], 'yticks':[]},
                        gridspec_kw=dict(hspace=0.1, wspace=0.1))

for i, ax in enumerate(axes.flat):
    ax.imshow(digits.images[i], cmap='binary', interpolation='nearest')
    ax.text(0.05, 0.05, str(digits.target[i]),
           transform=ax.transAxes, color='green')
```



In order to work with this data within `Scikit-Learn`, we need a two-dimensional, `[n_samples, n_features]` representation. We can accomplish this by treating each pixel in the image as a feature: that is, by flattening out the pixel arrays so that we have a length-64 array of pixel values representing each digit. Additionally, we need the target array, which gives the previously determined label for each digit. These two quantities are built into the digits dataset under the `data` and `target` attributes, respectively:

```
| X = digits.data
| X.shape
(1797, 64)

| y = digits.target
| y.shape
(1797,)
```

We see here that there are 1,797 samples and 64 features.

We can quickly classify the digits using a random forest as follows:

```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier

Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, random_state=0)
model = RandomForestClassifier(n_estimators=1000)
model.fit(Xtrain, ytrain)
ypred = model.predict(Xtest)
```

We can take a look at the classification report for this classifier:

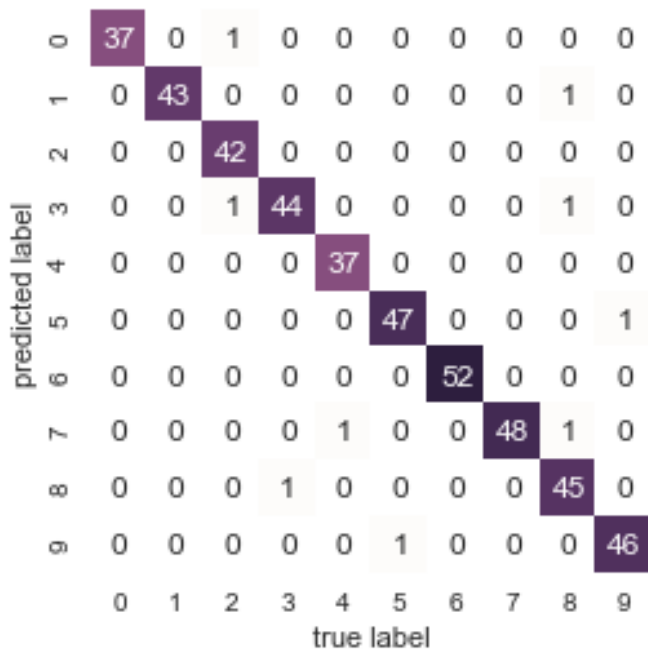
```
from sklearn import metrics
print(metrics.classification_report(ytest, ypred))
```

	precision	recall	f1-score	support
0	0.97	1.00	0.99	37
1	0.95	0.98	0.97	43
2	1.00	0.95	0.98	44
3	0.96	0.98	0.97	45
4	1.00	0.97	0.99	38
5	0.96	0.98	0.97	48
6	1.00	1.00	1.00	52
7	0.96	1.00	0.98	48
8	0.98	0.94	0.96	48
9	0.98	0.96	0.97	47
avg / total	0.98	0.98	0.98	450

And for good measure, plot the confusion matrix:

```
from sklearn.metrics import confusion_matrix

mat = confusion_matrix(ytest, ypred)
cmap = sns.cubehelix_palette(light=1, as_cmap=True)
sns.heatmap(mat.T, square=True, cmap=cmap, annot=True, fmt='d',
            cbar=False)
plt.xlabel('true label')
plt.ylabel('predicted label');
```



We find that a simple, untuned random forest results in a very accurate classification of the digits data.

Feature Importance

One great quality of Random Forests (and other ensemble tree methods) is that they make it easy to measure the relative importance of each feature (or attribute). Scikit-Learn measures a feature's importance by looking at how much the tree nodes that use that feature to reduce impurity on average (across all trees in the forest). More precisely, it is a weighted average, where each node's weight is equal to the number of training samples that are associated with it.

Scikit-Learn computes this score automatically for each feature after training, then it scales the results so that the sum of all importance is equal to 1. You can access the result using the `feature_importances_` variable. For example, the following code trains a `RandomForestClassifier` on the iris dataset and outputs each feature's importance.

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import load_iris
from sklearn.ensemble import RandomForestClassifier

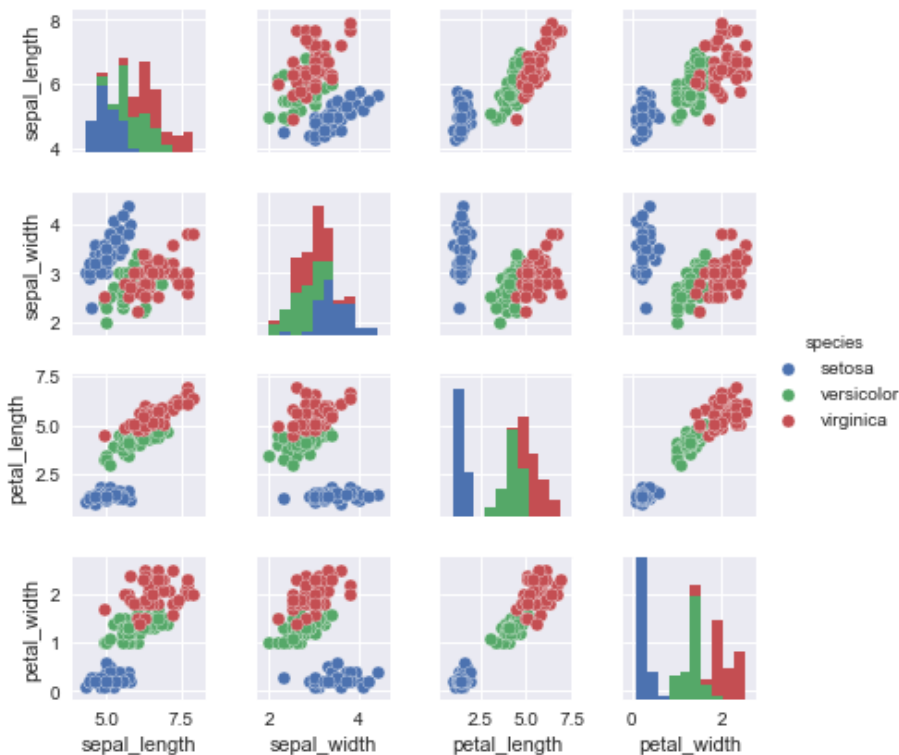
iris = load_iris()
rnd_clf = RandomForestClassifier(n_estimators=500, n_jobs=-1)
rnd_clf.fit(iris["data"], iris["target"])
for name, score in zip(iris["feature_names"],
rnd_clf.feature_importances_):
```

```
| print(name, score)

sepal length (cm) 0.107845222345
sepal width (cm) 0.0258824377512
petal length (cm) 0.435021720329
petal width (cm) 0.431250619575
```

We observe that the most important features are the petal length (43.5%) and width (43.1%), while sepal length and width are rather unimportant in comparison (10.8% and 2.6%, respectively).

Recall what we have plotted in Chapter 3 (shown below): we can also observe that the classification ability of `petal_length` and `petal_width` is better than `sepal_length` and `sepal_width`.



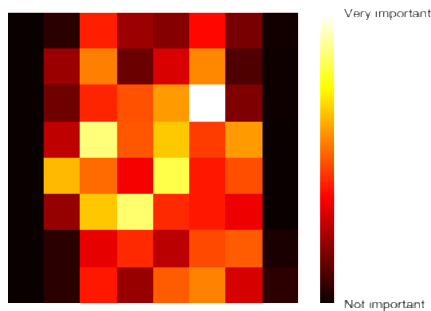
Similarly, if you plot each pixel's importance of a Random Forest classifier on the previous example of digit images with 8x8 pixels, you get the image represented in the figure below.

```
import matplotlib
def plot_digit(data):
    image = data.reshape(8, 8)
    plt.imshow(image, cmap = matplotlib.cm.hot,
                interpolation="nearest")
    plt.axis("off")

plot_digit(model.feature_importances_)

cbar = plt.colorbar(ticks=[model.feature_importances_.min(),
model.feature_importances_.max()])
```

```
| cbar.ax.set_yticklabels(['Not important', 'Very important'])
```



Remark: Random Forests are very handy to get a quick understanding of what features actually matter, in particular if you need to perform feature selection.

You can download the source codes `Ch4_RFDigitsRecog.ipynb` and `Ch4_RFFeatures.ipynb` to reproduce these two examples from the course website.

2.2.4 Boosting

Boosting refers to the ensemble methods that train a weak classifier sequentially, each trying to correct its predecessor. In other words, boosting uses the misclassifications of prior iterations to influence the training of the next iterative classifier. Boosting advocates an approach in which the weak classifier is forced to focus its attention on the “hardest” examples, that is, the ones for which the previous classifiers were most apt to give incorrect predictions.

Remarks:

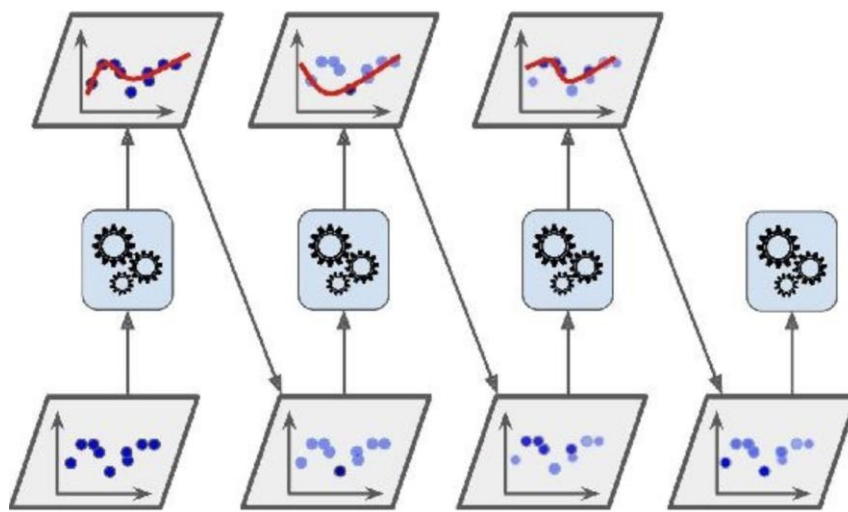
1. As we discussed, **bagging** is particularly effective for individual high-variance classifiers because it tends to smooth out the individual classifiers. On the other hand, boosting is particularly effective for high-bias classifiers that are slow to adjust to new data.
2. Boosting is serially iterative, whereas the individual base classifiers in bagging can be trained in parallel.
3. Boosting is similar to bagging in that it uses a majority-voting (or other similar schemes) process at the end; and it also combines base classifiers of the same type. On the other hand,

There are many boosting methods available, but by far the most popular are **AdaBoost** (short for **Adaptive Boosting**) and **Gradient Boosting**. Let's start with AdaBoost.

AdaBoost

One way for a new predictor to correct its predecessor is to **pay a bit more attention to the training instances that the predecessor underfitted**. This results in new predictors focusing more and more on the hard cases. This is the technique used by **AdaBoost**.

For example, to build an AdaBoost classifier, a first base classifier (such as a Decision Tree) is trained and used to make predictions on the training set. The relative weight of misclassified training instances is then increased. A second classifier is trained using the updated weights and again it makes predictions on the training set, weights are updated, and so on (see figure below).



AdaBoost sequential training with instance weight updates

The purpose of boosting is to sequentially apply the weak classification algorithm to repeatedly modified versions of the data, thereby producing a sequence of weak classifiers $G_m(\mathbf{x})$, $m = 1, 2, \dots, M$. The predictions from all of them are then combined through a weighted majority vote to produce the final prediction as illustrated below.

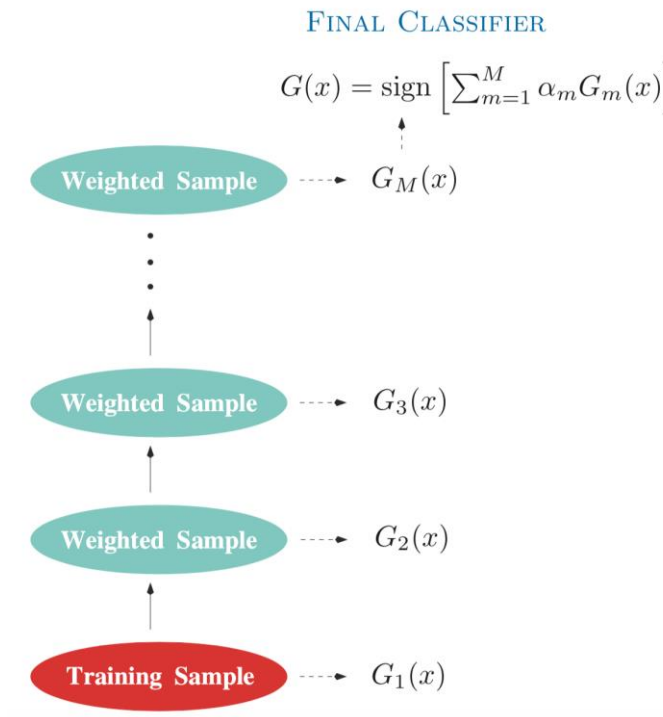
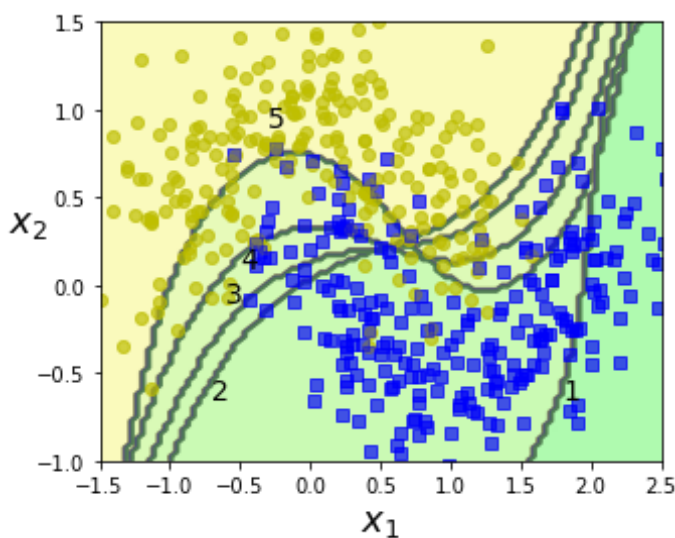


Figure below shows the decision boundaries of five consecutive predictors on the moons dataset (in this example, each predictor is a highly regularized SVM classifier with an RBF kernel). The first classifier gets many instances wrong, so their weights get boosted. The second classifier therefore does a better job on these instances, and so on.



You can download the complete source code `Ch4_AdaBoost.ipynb` from the course website to find the details.

Theoretical Minimum for AdaBoost

Let $\{(\mathbf{x}_j, y_j) | j = 1, 2, \dots, N\}$ denote a set of N training examples. In AdaBoost algorithm, the importance of a base classifier C_i depends on its error rate, which is defined as:

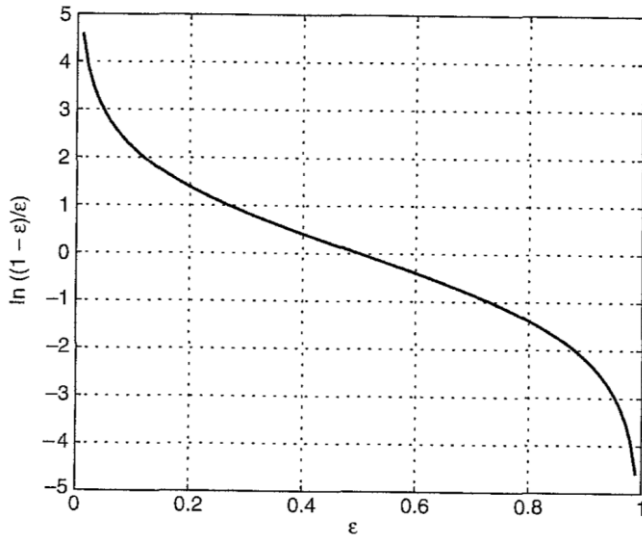
$$\varepsilon_i = \frac{1}{N} \left[\sum_{j=1}^N w_j I(C_i(\mathbf{x}_j) \neq y_j) \right]$$

where $I(p) = 1$ if the predicate p is true, and 0 otherwise. The importance of a classifier C_i is given by the following parameter:

$$\alpha_i = \frac{1}{2} \ln \left(\frac{1 - \varepsilon_i}{\varepsilon_i} \right)$$

Figure below plots the relationship between α_i and ε_i . What have you observed?

A:



Remark: As the base classifier should do better than a classifier that performs random guessing $\varepsilon = 0.5$, α parameter in general should be greater than 0.

The α_i parameter is also used to update the weight of the training examples. Let $w_i^{(j)}$ denote the weight assigned to example (\mathbf{x}_i, y_i) during the j^{th} boosting round. The weight update mechanism for AdaBoost is given by:

$$w_i^{(j+1)} = \frac{w_i^{(j)}}{Z_j} \times \begin{cases} e^{-\alpha_j} & \text{if } C_j(\mathbf{x}_i) = y_i \\ e^{\alpha_j} & \text{if } C_j(\mathbf{x}_i) \neq y_i \end{cases}$$

where Z_j is the normalization factor used to ensure $\sum_i w_i^{(j+1)} = 1$.

Q: What does the weight update formula imply?

A:

Gradient Boosting

Another very popular Boosting algorithm is **Gradient Boosting**. Just like AdaBoost, Gradient Boosting works by **sequentially adding predictors** to an ensemble, each one correcting its predecessor. However, instead of tweaking the instance weights at every iteration like AdaBoost does, this method tries to fit the new predictor to the residual errors made by the previous predictor.

XGBoost

XGBoost stands for eXtreme Gradient Boosting and is a popular and efficient open-source implementation of the gradient boosted trees algorithm. As we mentioned in Chapter 1, in 2016 and 2017, Kaggle was dominated by two approaches: **gradient boosting machines** and **deep learning**. Specifically, gradient boosting is used for problems where structured data is available, whereas deep learning is used for perceptual problems such as image classification. Practitioners of the former almost always use the excellent XGBoost library, which offers support for the two most popular languages of data science: Python and R. Meanwhile, most of the Kaggle entrants using deep learning use the Keras library, due to its ease of use, flexibility, and support of Python.

To install XGBoost, simply type the following command:

```
| conda install -c conda-forge xgboost
```

Or consult XGBoost Installation Guide, available on line.

Python Example

In this toy example, we are going to use the Pima Indians onset of diabetes dataset. This dataset is comprised of 8 input features that describe medical details of patients and one output variable to indicate whether the patient will have an onset of diabetes within 5 years. You can learn more about this dataset on the UCI Machine Learning Repository website.

This is a good dataset for a first XGBoost model because all of the input variables are numeric and the problem is a simple binary classification problem. It is not necessarily a good problem for the XGBoost algorithm because it is a relatively small dataset and an easy problem to model. For now, download this dataset from the course website and place it into your current working directory with the file name `pima-indians-diabetes.csv`.

Let us load and prepare the data. We can load the CSV file as a NumPy array using the NumPy function `loadtxt()`.

```
# First XGBoost model for Pima Indians dataset
from numpy import loadtxt
from xgboost import XGBClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
# load data
dataset = loadtxt('pima-indians-diabetes.csv', delimiter=",")
# split data into X and y
X = dataset[:,0:8]
y = dataset[:,8]
# split data into train and test sets
test_size = 0.33
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=test_size, random_state=7)
```

XGBoost provides a wrapper class to allow models to be treated like classifiers or regressors in the scikit-learn framework. This means we can use the full scikit-learn library with XGBoost models. The XGBoost model for classification is called `XGBClassifier`. We can create and fit it to our training dataset. Models are fit using the scikit-learn API and the `model.fit()` function. Parameters for training the model can be passed to the model in the constructor. Here, we use the sensible defaults.

```
# fit model to training data
model = XGBClassifier()
model.fit(X_train, y_train)
# make predictions for test data
y_pred = model.predict(X_test)
predictions = [round(value) for value in y_pred]
# evaluate predictions
accuracy = accuracy_score(y_test, predictions)
print("Accuracy: %.2f%%" % (accuracy * 100.0))
```

Accuracy: 77.95%

Remark: we can tune many parameters in XGBoost, for example the number of decision trees, learning rate, stochastic gradient boosting etc. Consult XGBoost website and Kaggle for details.

You can download the above source code `Ch4_XGBoostExample.ipynb` from the course website.

2.5 Class Imbalance and Classification Performance

In many data sets there are a disproportionate number of instances that belong to different classes, a property known as **skew** or **class imbalance**. For example, consider a health-care application where diagnostic reports are used to decide whether a person has a rare disease. Because of the infrequent nature of the disease, we can expect to observe a smaller number of subjects who are positively diagnosed.

Q: Can you think of other scenarios related to class imbalance problems?

A:

Despite their infrequent occurrences, **a correct classification of the rare class** often has greater value than a correct classification of the majority class. For example, it may be more dangerous to ignore a patient suffering from a disease than to misdiagnose a healthy person.

More generally, class imbalance poses **two challenges for classification**:

1. It can be difficult to find sufficiently many labeled samples of a rare class. In general, a classifier trained over an imbalanced data set shows a bias toward improving its performance over the majority class, which is often not the desired behavior. As a result, many existing classification models, when trained on an imbalanced data set, may not effectively detect instances of the rare class.
2. Accuracy, which is the traditional measure for evaluating classification performance, is not well-suited for evaluating models in the presence of class imbalance. For example, if 1% of the credit card transactions are fraudulent, then a trivial model that predicts every transaction as legitimate will have an accuracy of 99% even though it fails to detect any of the fraudulent activities. Thus, there is a need to use alternative evaluation metrics that are sensitive to the skew and can capture different criteria of performance than accuracy.

In this section, we will first present some of the generic methods for building classifiers when there is class imbalance in the training set. We then discuss methods for evaluating classification performance and adapting classification decisions in the presence of a skewed data set. In the remainder of this section, we will consider binary classification problems for simplicity where the minority class is referred as the positive (+) class while the majority class is referred as the negative (−) class.

2.5.1 Building Classifiers with Class Imbalance

There are **two primary considerations** for building classifiers in the presence of class imbalance in the training set:

1. We need to ensure that the learning algorithm is trained over a data set that has adequate representation of both the majority as well as the minority classes. Some common approaches for ensuring this includes the methodologies of **oversampling** and **undersampling** the training set.
2. Having learned a classification model, we need a way to adapt its classification decisions to best match the requirements of the imbalanced test set. This is typically done by **converting the outputs of the classification model to real-valued scores**, and then selecting a suitable threshold on the classification score to match the needs of a test set.

Undersampling and Oversampling

The first step in learning with imbalanced data is to transform the training set to a **balanced training set**, where both classes have nearly equal representation. The balanced training set can then be used with any of the existing classification techniques.

Q: How to transform an **imbalanced** training set to a **balanced** training set?

A:

Undersampling

Consider a training set that contains 100 positive examples and 1,000 negative examples. To overcome the skew among the classes, we can select a random sample of 100 examples from the negative class and use them with the 100 positive examples to create a balanced training set.

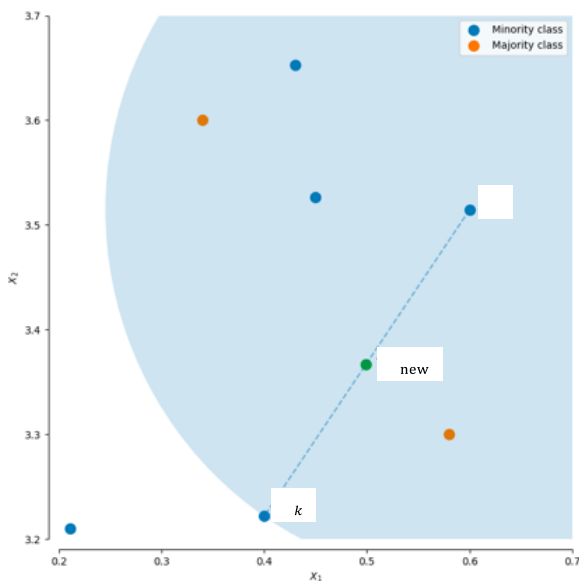
Q: what is the drawback of this approach?

A:

over fitting

Oversampling

Oversampling attempts to create a balanced training set by artificially generating new positive examples. A popular approach for oversampling is to generate synthetic positive instances for the minority class in the neighborhood of existing positive instances. In this approach, called the Synthetic Minority Oversampling Technique (SMOTE)², we first determine the k -nearest positive neighbors of every positive instance \mathbf{x} , and then generate a synthetic positive instance of some intermediate point along the line segment joining \mathbf{x} to one of its randomly chosen k -nearest neighbor, \mathbf{x}_k as illustrated below. This process is repeated until the desired number of positive instances is reached.



Python Example: SMOTE and additional under-sampling and over-sampling methods have been implemented in the `imbalanced-learn` package. The package is fully compatible with `scikit-learn`. To install the package, type

```
conda install -c conda-forge imbalanced-learn
```

Or see <https://imbalanced-learn.org/en/stable/install.html> for more installation guide. Below is a simple example using SMOTE in the package

```
from collections import Counter
from sklearn.datasets import make_classification
from imblearn.over_sampling import SMOTE
```


² N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, pages 321–357, 2002.

```
X, y = make_classification(n_classes=2, class_sep=2,
                           weights=[0.1, 0.9], n_informative=3,
                           n_redundant=1, flip_y=0, n_features=20,
                           n_clusters_per_class=1, n_samples=1000,
                           random_state=10)

print('Original dataset shape %s' % Counter(y))
```

Original dataset shape Counter({1: 900, 0: 100})

```
sm = SMOTE(random_state=42)
X_res, y_res = sm.fit_resample(X, y)
print('Resampled dataset shape %s' % Counter(y_res))
```



Resampled dataset shape Counter({0: 900, 1: 900})

One limitation of the SMOTE approach is that it can only generate new positive instance in the convex hull of the existing positive class. Hence, it does not help improve the representation of the positive class outside the boundary of existing positive instances. Despite their complementary strengths and weaknesses, undersampling and oversampling provide useful directions for generating balanced training sets in the presence of class imbalance.

2.5.2 Evaluating Performance with Class Imbalance

Again let us consider binary classification problems for simplicity where the minority class is referred as the positive (+) class while the majority class is referred as the negative (−) class.

The most basic approach for representing a classifier's performance on a test set is to use a **confusion matrix**, as shown below:

		Predicted Class	
		+	−
Actual Class	+	f_{++} (TP)	f_{+-} (FN)
	−	f_{-+} (FP)	f_{--} (TN)

A confusion matrix summarizes the number of instances predicted correctly or incorrectly by a classifier using the following four counts:

- True positive (TP) or f_{++} , which corresponds to the number of positive examples correctly

predicted by the classifier.

- False positive (FP) or f_{-+} (also known as Type I error), which corresponds to the number of negative examples wrongly predicted as positive by the classifier.
- False negative (FN) or f_{+-} (also known as Type II error), which corresponds to the number of positive examples wrongly predicted as negative by the classifier.
- True negative (TN) or f_{--} , which corresponds to the number of negative examples correctly predicted by the classifier.

Q: Recall the `accuracy_score` function computes the accuracy. If \hat{y}_i is the predicted value of the i^{th} sample and y_i is the corresponding true value, then the fraction of correct predictions over N samples is defined as:

$$\text{accuracy}(y_i, \hat{y}_i) = \frac{1}{N} \sum_{n=1}^N 1(\hat{y}_i = y_i)$$

The $1(\hat{y}_i = y_i)$ is an indicator function; it is 1.0 if $\hat{y}_i = y_i$ and 0.0 if $\hat{y}_i \neq y_i$. How to define accuracy in terms of TP, FP, FN, TN?

A:

Remark: Let us consider a dataset describing 1,000 patients who have been diagnostic for a rare disease. Because of the infrequent nature of the disease, we can expect to observe a smaller number of subjects who are positively diagnosed (say 5%). Suppose we have a classifier that always predicts negative for the dataset:

Ground Truth

	Predicted (+)	Predicted (−)
Actual (+)	$f_{++} = 50$	$f_{+-} = 0$
Actual (−)	$f_{-+} = 0$	$f_{--} = 950$

Prediction

	Predicted (+)	Predicted (−)
Actual (+)	0	50
Actual (−)	0	950

The accuracy score is 95%, impressive but misleading.

The confusion matrix provides a concise representation of classification performance on a given test

data set. However, it is often difficult to interpret and compare the performance of classifiers using the four-dimensional representations (corresponding to the four counts) provided by their confusion matrices.

Hence, the four counts in the confusion matrix are often summarized using a number of **evaluation measures**. Accuracy is an example of one such measure that combines **these four counts into a single value**, which is used extensively when classes are balanced. However, the accuracy measure is not suitable for handling data sets with imbalanced class distributions as it tends to favor classifiers that correctly classify the majority class.

There are a few **evaluation measurements** other than accuracy to help us determine the performance of our classifier. Let us define a few of them:

		Predicted Class	
		+	-
Actual Class	+	f_{++} (TP)	f_{+-} (FN)
	-	f_{-+} (FP)	f_{--} (TN)

Recall or True Positive Rate TPR: It is also known as sensitivity or hit rate. It corresponds to the proportion of positive data points that are correctly classified as positive versus the total number of positive points:

$$\text{recall (TPR)} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Example: let us consider a binary classification system implemented for example by a World War II air reconnaissance troop. Their task consists of distinguishing enemy aircraft from flocks of birds. Let us imagine that they have taken 100 measurements and created a confusion matrix as shown in Table 2.1 below. The troop is of course interested in detecting correctly when an enemy plane is flying above them.

Table 2.1

		Predicted Class	
		Enemy Aircraft	Flock of Birds
Actual class	Enemy Aircraft	20	4
	Flock of Birds	6	70

Q: what is the recall for the case?

A:

Specificity or True Negative Rate TNR: It is the counterpart of the True Positive Rate as it measures the proportion of negatives that have been correctly identified. It is given by:

$$\text{specificity (TNR)} = \frac{\text{TN}}{\text{FP} + \text{TN}}$$

Q: what is the specificity for the case shown in Table 2.1?

A:

Fallout or False Positive Rate FPR: It corresponds to the proportion of negative data points that are mistakenly considered as positive, with respect to all negative data points:

$$\text{fallout (FPR)} = \frac{\text{FP}}{\text{FP} + \text{TN}} = 1 - \text{TNR}$$

In other words, the higher the FPR, the more negative data points we will have misclassified. In our example in Table 2.1, $\text{FPR} = \frac{6}{6+70} = 0.079$.

		Predicted Class	
		+	−
Actual Class	+	f_{++} (TP)	f_{+-} (FN)
	−	f_{-+} (FP)	f_{--} (TN)

Precision or Positive Predictive Value PPV: It is the proportion of positive results that are true positive results. Positive predictive value

$$\text{precision (PPV)} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Q: what is the precision for the case shown in Table 2.1?

A:

We now present a discussion on a number of alternate metrics used for evaluating the performance of classifiers on imbalanced datasets.

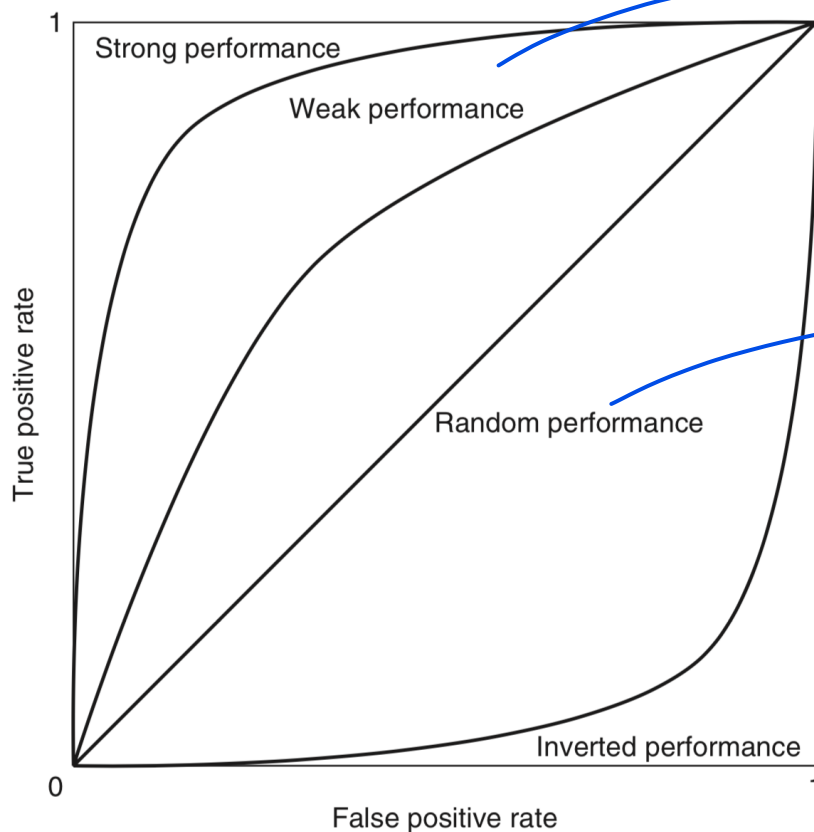
2.5.3 ROC and AUC

The receiver operating characteristic (ROC) curve is a standard technique for evaluating classifiers on datasets that exhibit class imbalance. ROC curves achieve this skew insensitivity by summarizing the performance of classifiers over a range of true positive rates (TPRs) and false positive rates (FPRs). By evaluating the models at various error rates, ROC curves are able to determine what proportion of instances will be correctly classified for a given FPR.

In the figure below, we see an example of a ROC curve. In the figure, the x -axis represents the FPR ($\text{FPR} = \text{FP}/(\text{TN} + \text{FP})$), and the y -axis represents the TPR ($\text{TPR} = \text{TP}/(\text{TP} + \text{FN})$).

Q: what would a perfect classifier behave in the ROC curve?

A:



$AUC > 0.5$

$AUC = 0.5$

Alternatively, the classifier that misclassifies all instances would have a single point at (1, 0). While (0, 1) represents the ideal classifier and (1, 0) represents its complement, in ROC space, the line $y = x$ represents a random classifier, that is, a classifier that applies a random prediction to each instance. This gives a trivial lower bound in ROC space for any classifier. An ROC curve is said to “dominate” another ROC curve if, for each FPR, it offers a higher TPR. By analyzing ROC curves, one can determine the best classifier for a specific FPR by selecting the classifier with the best corresponding TPR.

While ROC curves provide a visual method for determining the effectiveness of a classifier, the area under the ROC curve (**AUC** or **AUROC**) has become the *de facto* standard metric for evaluating classifiers under imbalance. This is due to the fact that it is both independent of the selected threshold and prior probabilities, as well as **offering a single number to compare classifiers**.

Q: what is the AUC for a random guess classifier?

A:

2.5.4 Precision and Recall

Alternatives to AUC are precision and recall.

$$\text{recall (TPR)} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

$$\text{precision (PPV)} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

In imbalanced datasets, the goal is to improve recall without hurting precision. These goals, however, are often conflicting, since in order to increase the TP for the minority class, the number of FP is also often increased, resulting in reduced precision.

In order to obtain a more accurate understanding of the trade-offs between precision and recall, one can use **precision–recall (PR) curves**. PR curves are similar to ROC curves in that they provide a graphical representation of the performance of classifiers. While the x -axis of ROC curves is FPR and the y -axis is TPR, in PR curves, the x -axis is recall (TPR) and the y -axis is precision (PPV). While TPR measures the fraction of positive examples that are correctly classified, precision measures the fraction of examples that are classified as positive that are actually positive.

Similar to ROC curves are being compared on the basis of AUC, PR curves are also compared on the basis of area under the PR curve (**AUPR** or sometimes called **PR AUC**). **Similar to AUC, the higher the AUPR is, the better the classification model is.** This practice has become more common, as recent research suggests that PR curves (and **AUPR**) are a better discriminator of performance than their ROC (and AUC) counterparts³.

Example: For illustration, let's take an example of an information retrieval problem where we want to find a set of, say, 100 relevant documents out of a list of 1 million possibilities based on some query. Let's say we've got two algorithms we want to compare with the following performance:

Method 1: 100 retrieved documents, 90 relevant.

Method 2: 2000 retrieved documents, 90 relevant.

³ J. Davis and M. Goadrich, "The relationship between precision-recall and roc curves," in *Proceedings of the Twenty third International Conference on Machine Learning*, pp. 233–240. ACM, 2006.

Q: which one is better?

A:

model 1

The ROC measures of TPR and FPR will reflect that, but since the number of irrelevant documents dwarfs the number of relevant ones, the difference is mostly lost:

Method 1: 0.9 TPR, 0.00001 FPR

Method 2: 0.9 TPR, 0.00191 FPR (difference of 0.0019)

Precision and recall, however, don't consider true negatives and thus won't be affected by the relative imbalance (which is precisely why they're used for these types of problems):

Method 1: 0.9 precision, 0.9 recall

Method 2: 0.045 precision (difference of 0.855), 0.9 recall

Obviously, those are just single points in ROC and PR space, but if these differences persist across various scoring thresholds, using ROC AUC, we'd see a very small difference between the two algorithms, whereas PR AUC would show quite a large difference.

2.5.5 F_β -Measure

A final common metric is the F_β -measure. F_β -measure is a family of metrics that attempts to measure the trade-offs between precision and recall by outputting a single value that reflects the goodness of a classifier in the presence of rare classes. While ROC curves represent the trade-off between different TPRs and FPRs, F_β -measure represents the trade-off among different values of TP, FP, and FN.

The general equation for F_β -measure is:

$$F_\beta = (1 + \beta^2) \frac{\text{precision} \cdot \text{recall}}{(\beta^2 \cdot \text{precision}) + \text{recall}}$$

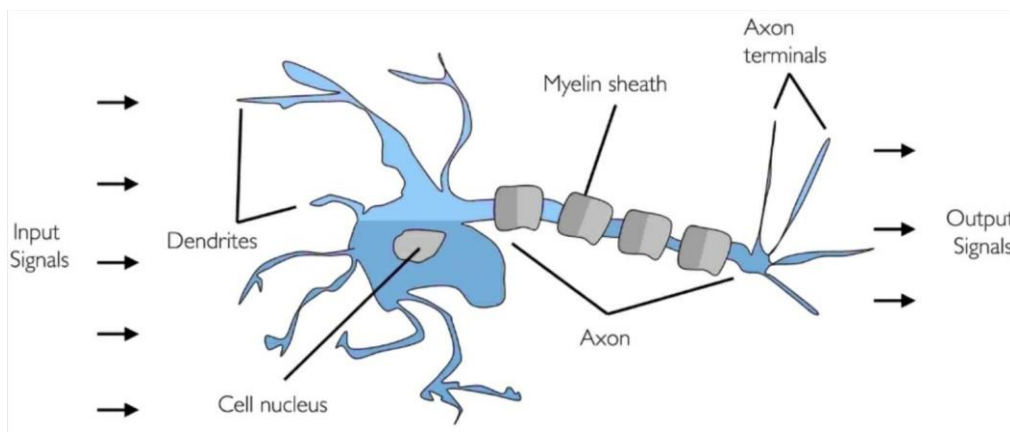
where β represents the relative importance of precision and recall. Traditionally, when β is not specified, the F_1 -measure is assumed.

In spite of its (relatively) useful properties for imbalance, F_β -measure is not commonly used when comparing classifiers, as **AUROC** and **AUPR** provide more robust and better performance estimates.

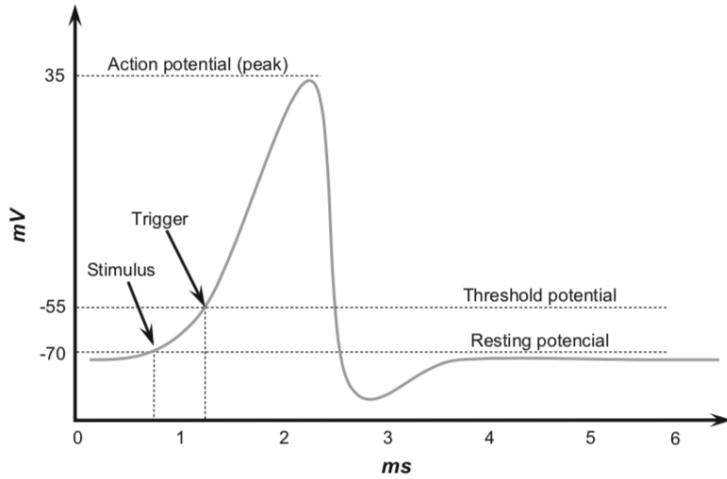
2.6 Artificial Neural Networks: The Perceptron (感知器) Network

Deep learning is a deep neural network and understanding deep learning requires familiarity with some concepts and mathematical building blocks from neural networks or sometimes called artificial neural networks (ANNs). In this section, we will introduce ANNs, starting with a quick tour of the very first ANN architectures, the Perceptron (感知器). We will then present Multi-Layer Perceptrons (MLPs).

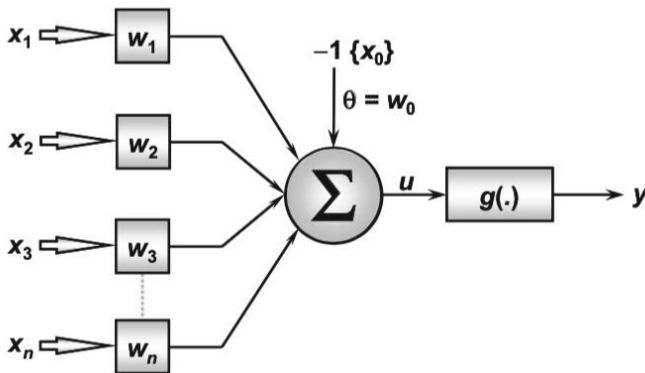
The Perceptron network is one of the simplest ANN architectures, invented in 1957 by Frank Rosenblatt. The Perceptron network is inspired by the biological neuron shown below. Neurons are interconnected nerve cells in the brain that are involved in the processing and transmitting of chemical and electrical signals, which is illustrated in the following figure:



When the nervous cell is stimulated with an impulse higher than its activation threshold (-55 mV), caused by the variation of internal concentrations of sodium (Na^+) and potassium (K^+) ions, it triggers an electrical impulse which will propagate throughout its axon with a maximum amplitude of 35 mV. The stages related to variations of the action voltage within a neuron during its excitation are shown in the Figure below.



The Perceptron network consists of one artificial neuron. Figure below illustrates a Perceptron network composed of n input signals, representing the problem being analyzed, and just one output.



In mathematical notation, the inner processing performed by the Perceptron can be described by the following expressions:

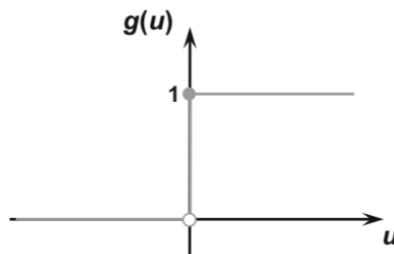
$$\begin{cases} u = \sum_{i=1}^n w_i x_i - \theta \\ y = g(u) \end{cases}$$

where x_i is a network input, w_i is the weight associated with the i^{th} input, θ is the activation threshold (or bias), $g(\cdot)$ is the activation function and u is the activation potential or sometimes called input signal.

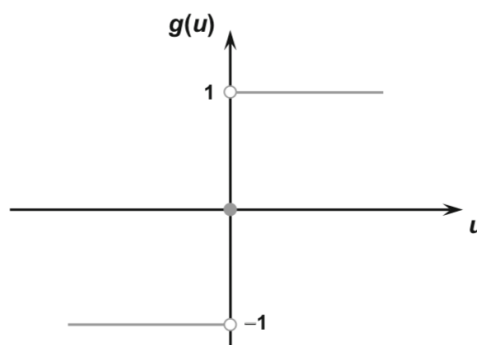
The most common activation functions used in Perceptrons are the step and bipolar step functions:

Step Function

$$g(u) = \begin{cases} 0 & \text{if } u < 0 \\ 1 & \text{if } u \geq 0 \end{cases}$$

Bipolar Step Function

$$g(u) = \begin{cases} -1 & \text{if } u < 0 \\ 1 & \text{if } u \geq 0 \end{cases}$$



Q: What are the possible output values possibly produced by the Perceptron?

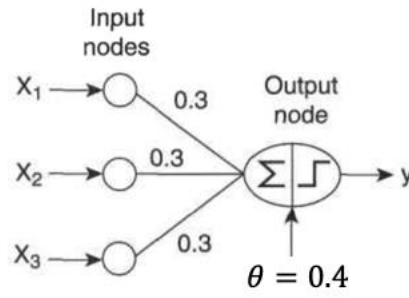
A:

The Perceptron network can be used for simple linear binary classification. It computes a linear combination of the inputs and if the result exceeds a threshold, it outputs the positive class or else outputs the negative class (just like a linear SVM). Training a Perceptron network means finding the right values for the weights w_i and threshold θ .

Example: Consider a Perceptron network shown below. A perceptron computes its output value, y , by performing a weighted sum on its inputs, subtracting a threshold from the sum, and then examining the sign of the result. The model has three input nodes, each of which has an identical weight of 0.3 to the output node and a threshold of 0.4. These values can be found by training covered in the sequel.

X_1	X_2	X_3	y
1	0	0	-1
1	0	1	1
1	1	0	1
1	1	1	1
0	0	1	-1
0	1	0	-1
0	1	1	1
0	0	0	-1

(a) Data set.



(b) Perceptron.

Q: Construct the model.

A:

2.6.1 Training Process of the Perceptron

The adjustment of Perceptron's weights and thresholds, to classify patterns that belong to one of the two possible classes, is performed by the use of Hebb's learning rule (Hebb 1949).

In short, if the output produced by the Perceptron coincides with the desired output, its weights and threshold remain unchanged (inhibitory condition); otherwise, in the case the produced output is different from the desired value, then its weights and threshold are adjusted proportionally to its input signals. This process is repeated sequentially for all training samples until the output produced by the Perceptron is like the desired output of all samples.

In mathematical notation, the rules for adjusting the weights w_i and threshold θ can be expressed, respectively, by the following equations:

$$w_i^{\text{current}} = w_i^{\text{previous}} - \eta \cdot (y - d^{(k)}) \cdot x_i^{(k)}$$

$$\theta^{\text{current}} = \theta^{\text{previous}} - \eta \cdot (y - d^{(k)}) \cdot (-1)$$

These two equations can be represented by a single vector expression given by:

$$\mathbf{w}^{\text{current}} = \mathbf{w}^{\text{previous}} - \eta \cdot (y - d^{(k)}) \cdot \mathbf{x}^{(k)}$$

In algorithmic notation, the inner processing performed by the Perceptron can be described by the following expression:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \cdot (y - d^{(k)}) \cdot \mathbf{x}^{(k)}$$

where

$\mathbf{w} = [\theta \ w_1 \ w_2 \ \cdots \ w_n]$ is the vector containing the threshold and weights;

$\mathbf{x}^{(k)} = [-1 \ x_1^{(k)} \ x_2^{(k)} \ \cdots \ x_n^{(k)}]$ is the k^{th} training sample;

$d^{(k)}$ is the desired value for the k^{th} training sample;

η is a constant that defines the learning rate of the Perceptron.

The learning rate η determines how fast the training process will take to its convergence (stabilization). The choice of η should be done carefully to avoid instabilities in the training process, and it is usually defined within the range $0 < \eta \leq 1$.

2.6.2 Implementing a perceptron learning algorithm in Python

In the previous section, we learned how Rosenblatt's perceptron rule works; let's now implement it in Python and apply it to the Iris dataset that we introduced in Chapter 1, Giving Computers the Ability to Learn from Data.

We will take an object-oriented approach to defining the perceptron interface as a Python class, which will allow us to initialize new Perceptron objects that can learn from data via a fit method and make predictions via a separate predict method. As a convention, we append an underscore (`_`) to attributes that are not created upon the initialization of the object, but we do this by calling the object's other methods, for example, `self.w_`.

The following is the implementation of a perceptron in Python:

```
import numpy as np
class Perceptron(object):
    """Perceptron classifier.

    Parameters
    -----
    eta : float
        Learning rate (between 0.0 and 1.0)
    n_iter : int
        Passes over the training dataset.
    random_state : int
        Random number generator seed for random weight
        initialization.

    Attributes
    -----
    w_ : 1d-array
        Weights after fitting.
```

```

errors_ : list
    Number of misclassifications (updates) in each epoch.

"""
def __init__(self, eta=0.01, n_iter=50, random_state=1):
    self.eta = eta
    self.n_iter = n_iter
    self.random_state = random_state

def fit(self, X, y):
    """Fit training data.

    Parameters
    -----
    X : {array-like}, shape = [n_examples, n_features]
        Training vectors, where n_examples is the number of
        examples and n_features is the number of features.
    y : array-like, shape = [n_examples]
        Target values.

    Returns
    -----
    self : object

    """
    rgen = np.random.RandomState(self.random_state)
    self.w_ = rgen.normal(loc=0.0, scale=0.01,
                          size=1 + X.shape[1])
    self.errors_ = []

    for _ in range(self.n_iter):
        errors = 0
        for xi, target in zip(X, y):
            update = self.eta * (target - self.predict(xi))
            self.w_[1:] += update * xi
            self.w_[0] += update
            errors += int(update != 0.0)
        self.errors_.append(errors)
    return self

def net_input(self, X):
    """Calculate net input"""
    return np.dot(X, self.w_[1:]) + self.w_[0]

def predict(self, X):
    """Return class label after unit step"""
    return np.where(self.net_input(X) >= 0.0, 1, -1)

```

Using this perceptron implementation, we can now initialize new Perceptron objects with a given learning rate, eta, and the number of epochs, n_iter (passes over the training dataset).

Via the fit method, we initialize the weights in self.w_ to a vector, , where m stands for the number of dimensions (features) in the dataset, and we add 1 for the first element in this vector that represents

the bias unit. Remember that the first element in this vector, `self.w_[0]`, represents the so-called bias unit that we discussed earlier.

Also notice that this vector contains small random numbers drawn from a normal distribution with standard deviation 0.01 via `rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])`, where `rgen` is a NumPy random number generator that we seeded with a user-specified random seed so that we can reproduce previous results if desired.

It is important to keep in mind that we don't initialize the weights to zero because the learning rate, η (eta), only influences the classification outcome if the weights are initialized to non-zero values. If all the weights are initialized to zero, the learning rate parameter, eta, affects only the scale of the weight vector, not the direction. After the weights have been initialized, the `fit` method loops over all individual examples in the training dataset and updates the weights according to the perceptron learning rule that we discussed in the previous section.

The class labels are predicted by the `predict` method, which is called in the `fit` method during training to get the class label for the weight update; but `predict` can also be used to predict the class labels of new data after we have fitted our model. Furthermore, we also collect the number of misclassifications during each epoch in the `self.errors_list` so that we can later analyze how well our perceptron performed during the training. The `np.dot` function that is used in the `net_input` method simply calculates the vector dot product, $w^T x$.

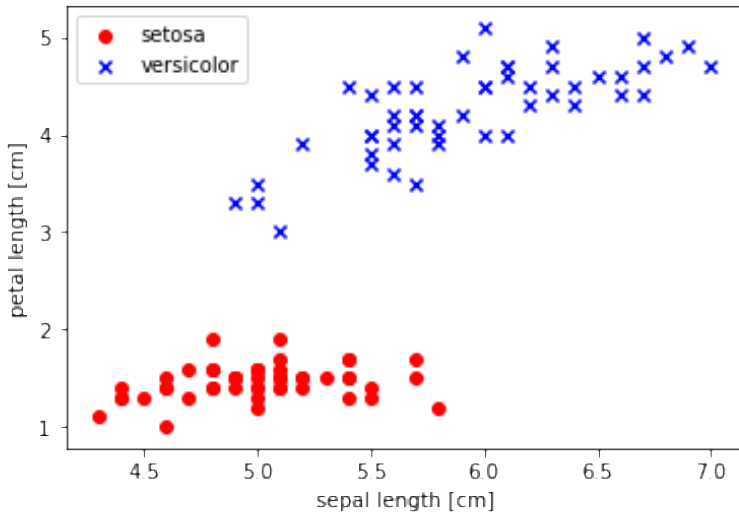
Python Example: The scikit-learn has Perceptron class implemented in the `linear_model` module. Let us use parts of Iris dataset to demonstrate its simple usage for binary classification. Our goal is to use sepal and petal lengths to classify the flowers for Setosa or Versicolor:

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron
from sklearn.model_selection import train_test_split

iris = load_iris()
X = iris.data[0:100, (0, 2)] # sepal length, petal length
y = iris.target[0:100] # Setosa or Versicolor

# plot data
plt.scatter(X[:50, 0], X[:50, 1],
            color='red', marker='o', label='setosa')
plt.scatter(X[50:100, 0], X[50:100, 1],
            color='blue', marker='x', label='versicolor')
```

```
plt.xlabel('sepal length [cm]')
plt.ylabel('petal length [cm]')
plt.legend(loc='upper left')
```



```
per_clf = Perceptron(random_state=42) #default learning rate = 1.0
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=1)
per_clf.fit(X_train,y_train)

y_pred = per_clf.predict(X_test)
print('Misclassified samples: %d' % (y_test != y_pred).sum())
```

Misclassified samples: 0

The threshold and weights can be found via:

```
print ("threshold: ", per_clf.intercept_)
print ("w1: ", per_clf.coef_[0,0])
print ("w2: ", per_clf.coef_[0,1])
```

```
threshold:  [-0.01]
w1:  -0.027
w2:  0.052
```

You can download the above Python codes `perceptron.ipynb` from the course website.

2.7 Summary

In this Chapter, we covered a few classification models, from simple techniques such as decision tree classifier, naïve Bayes classifier to more sophisticated techniques such as support vector machine, ensemble methods (bagging, random forests, boosting). To choose the best classification model for a particular task, we recall the techniques we have learned from Chapter 1: cross validation for model

validation and **the bias–variance trade-off**, validation curve, learning curve and grid search for model selection. It will in general take a few fine-tuning of hyperparameters and judgements to come up with the best model for a particular task.

In addition, practical issue such as the class imbalance problems are also discussed. When attempting to evaluate the performance of the classifiers, we realized that accuracy is not a valuable evaluation metric when learning in imbalanced environments. In order to overcome this issue, we presented multiple alternative evaluation metrics. The most commonly used alternatives discussed were AUROC and AUPR.

We finally note that most of the techniques presented for classification problems are also applicable to regression problems as we will encounter in the next Chapter.