Introduction to Machine Learning

Coursera

Sergio Pablo Rodríguez Guzmán

January 2022

Contents

List of Figures

Chapter 1

Fundamentals of Machine Learning - Intro to SciKit Learn

This chapter introduces basic machine learning concepts, tasks, and workflow using an example classification problem based on the K-nearest neighbors method, and implemented using the scikit-learn library. Learning objectives:

- Understand basic machine learning concepts and workflow.
- Distinguish between different types of machine learning tasks, based on examples of how they are applied to real-world problems.
- Understand how a basic classification algorithm (k-nearest neighbors) learns and makes predictions.
- Build and evaluate a basic k-nearest neighbors classifier on an example dataset using Python and scikit-learn.

1.1 Why Machine Learning?

Machine Learning is about extracting knowledge from data. The basic problem of machine learning is to explore how computers can program themselves to perform a task, and to improve their performance automatically as they gain more experience. Now this experience can take the form of data in a lot of different formats or situations.

Machine Learning algorithms that learn from input/output pairs are called *supervised Learning* algorithms because a teacher provides supervision to the algorithms in the form of the desired outputs for each example that they learn from.

- Classification (target values are discrete)
- Regression (target values are continuous values)

In *Unsupervised Learning* algorithms only the input data is known, and no known output data is given to the algorithm.

- Find groups of similar instances in the data (clustering)
- Finding unusual patterns (outlier detection)

1.2 Basic Machine Learning Workflow

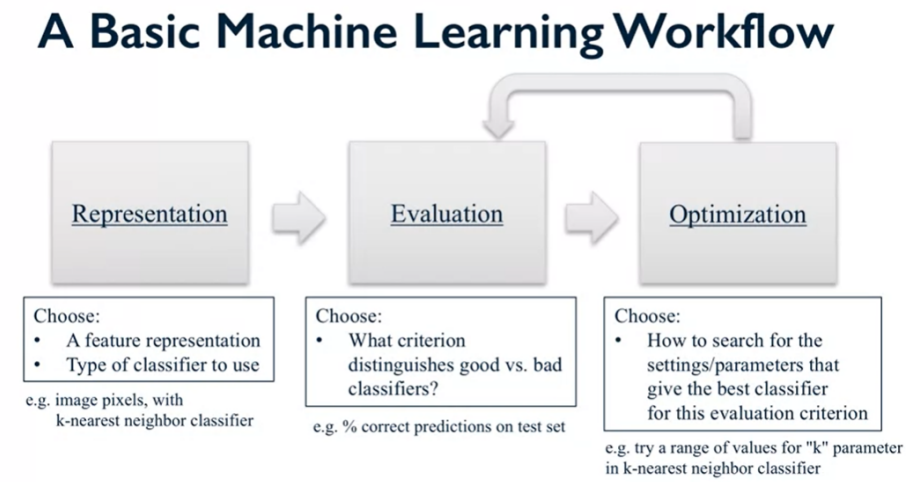


Figure 1.1: A Basic Machine Learning Workflow

So the first step in solving a problem with machine learning is you have to figure out how to represent the learning problem in terms of something the computer can understand. You need to be able to take your data or even formulate the description of your object that you're interested in recognizing, for example, in a way that you can use input to an algorithm. And you also need to decide what type of learning algorithm to apply to this data.

The second thing need to do is decide on an evaluation method that provides some type of quality or accuracy score. For the predictions or the output that is coming out of the machine learning algorithm typically I say classifier. So if you have any violation method this allows you to access and compare the effectiveness of different classifiers. So you can tell what classifiers are doing well, which are the good ones, and which are the bad ones for your particular problem.

The third thing you need to do in applying machine learning to solve a problem is. Once we've decided on how to represent the input data, the type of classifier and the evaluation method. We need to then search for the optimal classifier that gives the best evaluation outcome for that problem.

So often the process of addressing a machine learning task is a cycle. It involves an iterative process, as I show here, where we make an initial guess about what some good features are for the problem. And the classifier that might be appropriate. We then train the system using our training data. Produce an evaluation, see how well the classifier works and then based on what worked and what didn't work which examples get classified correctly or incorrectly we can do a failure analysis to see where the system is still making mistakes. And then with the results of that failure analysis,

we typically will always refine the set of feature. We may discover that important feature is missing that would help fix some of the mistakes for example.

1.3 A First Application

1.3.1 Python Tools

Import required modules:

```
%matplotlib notebook
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
import pandas as pd
```

1.3.2 Meet and Load Data

The most important part in Machine Learning process is understanding the data you are working with and how it relates to the task you want to solve.

```
from sklearn.datasets import load_iris
iris = load_iris()
iris.shape
print('Keys of iris_data: \n {}'.format(iris_keys()))

#other data base
fruits = pd.read_table('fruit_data_with_colors.txt')
fruits.head()
fruits.shape
lookup_fruit_name = dict(zip(fruits.fruit_label.unique(),
                    fruits.fruit_name.unique()))
```

For both supervised and unsupervised data is helpful to think of your data as a table, where each row or entity is known as a *sample* or data point while the columns are the properties that describes these entities are called *features*.

1.3.3 Measuring Success: Training and Testing Data

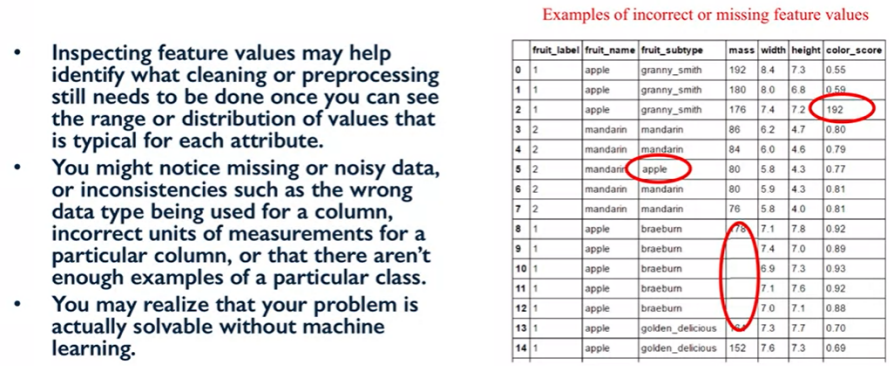
We cannot use the data we used to build the model to evaluate it. To assess the model's performance, we show it new data that it hasn't seen before for which we have labels. This is done by splitting the labeled data we have collected into two parts. Partitioning the data set into training and test sets using the Train/Test split function.

- training set: 75% → used to build our Machine Learning model.
- test set: 25% → used to assess how well the model works.
- random state parameter makes sure we will get the same output if we run the same function several times.

```
from sklearn.model_selection import train_test_split
X = fruits[['mass','width','height']]
y = fruits['fruit_label']
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state = 0)
```

1.3.4 Look at Your Data

Some reasons why looking at the data initially is important:



	fruit_label	fruit_name	fruit_subtype	mass	width	height	color_score
0	1	apple	granny_smith	192	8.4	7.3	0.55
1	1	apple	granny_smith	180	8.0	6.8	0.59
2	1	apple	granny_smith	176	7.4	7.2	192
3	2	mandarin	mandarin	86	6.2	4.7	0.80
4	2	mandarin	mandarin	84	6.0	4.6	0.79
5	2	mandarin	apple	80	5.8	4.3	0.77
6	2	mandarin	mandarin	80	5.9	4.3	0.81
7	2	mandarin	mandarin	76	5.8	4.0	0.81
8	1	apple	braeburn	178	7.1	7.8	0.92
9	1	apple	braeburn		7.4	7.0	0.89
10	1	apple	braeburn		6.9	7.3	0.93
11	1	apple	braeburn		7.1	7.6	0.92
12	1	apple	braeburn		7.0	7.1	0.88
13	1	apple	golden_delicious		7.3	7.7	0.70
14	1	apple	golden_delicious	152	7.6	7.3	0.69

Figure 1.2: Examples of incorrect or missing feature values

One of the best ways to inspect data is to visualize it. A *pair plot* looks at all possible pairs of features.

```
from matplotlib import cm
cmap = cm.get_cmap('gnuplot')
scatter = pd.scatter_matrix(X_train, c=y_train, figsize=(15,15),
marker='o', hist_kwds={'bins': 20}, s=60, alpha=0.8, cmap=cmap)
```

We can also look at features that use a subset of three features by creating a three dimensional plot to see a possible well-defined cluster appear in the visualization.

```
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(X_train['width'], X_train['height'], X_train['color_score'], c=y_train,
marker='o', s=100)
ax.set_xlabel('width')
ax.set_ylabel('height')
ax.set_zlabel('color_score')
plt.show()
```

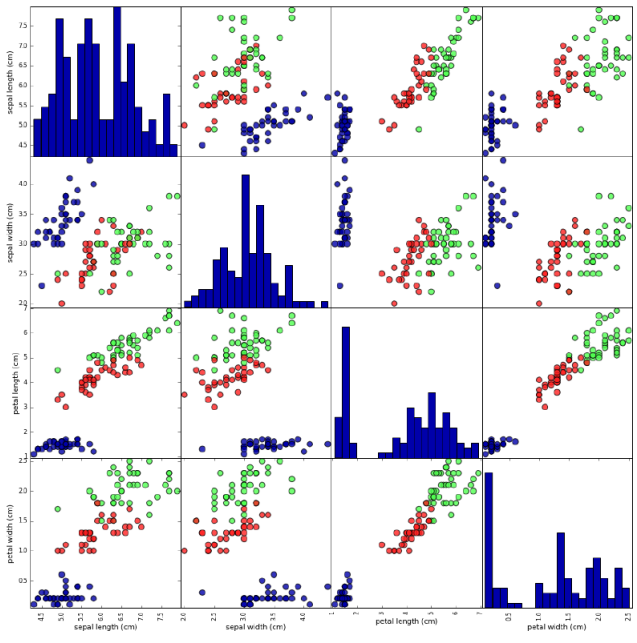


Figure 1.3: Pair plot of the Iris dataset, colored by class label

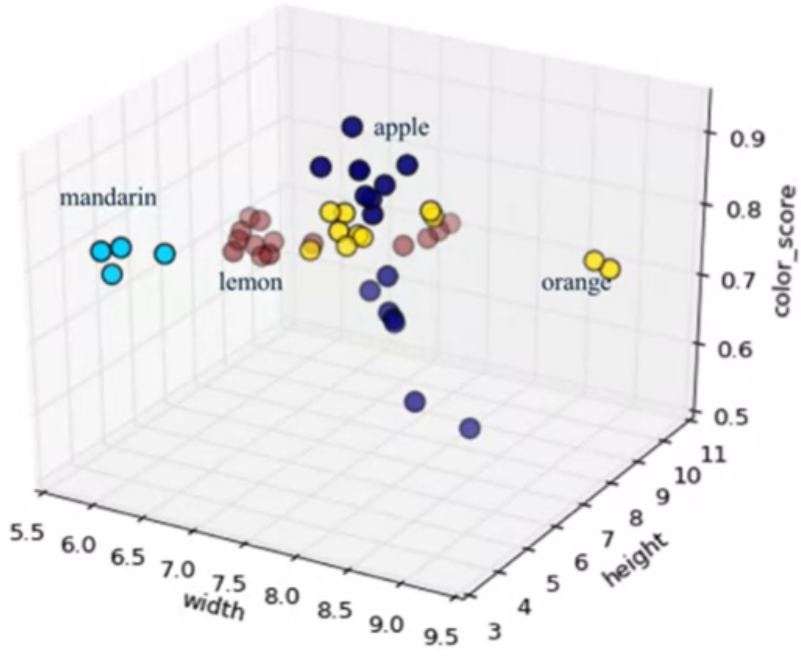


Figure 1.4: A three-dimensional feature scatterplot

1.3.5 Build Model

Now we can start building the actual Machine Learning Model. All models in *scikit-learn* are implemented in their own classes, which are called *Estimator* classes. The KNN classification algorithm is implemented in the *KNeighborsClassifier* class in the *neighbors* module. Before we can use the model, we need to instantiate the class into an object. This is when we will set any parameters of the model. In *KNeighborsClassifier* the most important parameter is the number of neighbors. The knn object encapsulates the algorithm that will be used to build the model from training data, as well the algorithm to make predictions on new data points. It will also hold the information that the algorithm has extracted from the training data. To build a model on the training set, we call the fit method of the knn object on the training set to estimate the model.

```
from sklearn.neighbors import KNeighborsClassifier
# create classifier object
knn = KNeighborsClassifier(n_neighbors=1)

# train the classifier (fit the estimator) using the training data
knn.fit(X_train, y_train)
```

1.3.6 Making Predictions and Evaluating the Model

We can now make predictions applying the model by using the predict method of the knn object to estimate a target value for the new data instances, or by using the score method to evaluate the trained model's performance on the test set.

```
example_fruit = [[5.8, 2.2, 10, 0.7]]
example_fruit_scaled = scaler.transform(example_fruit)
prediction = knn.predict(example_fruit_scaled)
```

Evaluate the model computing accuracy of the classifier on future data or use score method, using the test data.

```
y_pred = knn.predict(X_test)
print('Test set score:{:.2f}'.format(np.mean(y_pred==y_test)))
print('Test set score:{:.2f}'.format(knn.score(X_test,y_test)))
```

Plot the decision boundaries of the k-NN classifier

This will produce the colored plots that have the decision boundaries. You can then try out different values of k for yourself to see what the effect is on the decision boundaries.

```
from adspy_shared_utilities import plot_fruit_knn
# the last parameter is the waiting method to be used.
# So here I'm passing in the string uniform, which means
# to treat all neighbours equally when combining their labels
# try using distance wave method
plot_fruit_knn(X_train, y_train, 5, 'uniform')
```

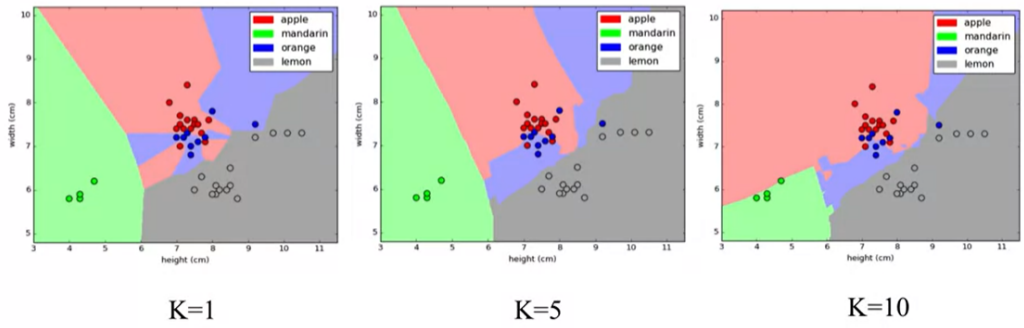


Figure 1.5: Different decision boundaries of k-NN classifer

We can see that when K has a small value like 1, the classifier is good at learning the classes for individual points in the training set. But with a decision boundary that's fragmented with considerable variation. This is because when K = 1, the prediction is sensitive to noise, outliers, mislabeled data, and other sources of variation in individual data points. For larger values of K, the areas assigned to different classes are smoother and not as fragmented and more robust to noise in the individual points. But possibly with some mistakes, more mistakes in individual points. This is an example of what's known as the bias variance tradeoff.

how the value of k, how the choice of k, affects the accuracy of the classifier?

We can plot the accuracy as a function of k very easily using this short snippet of code.

```
k_range = range(1,20)
scores = []

for k in k_range:
    knn = KNeighborsClassifier(n_neighbors = k)
    knn.fit(X_train, y_train)
    scores.append(knn.score(X_test, y_test))

plt.figure()
plt.xlabel('k')
plt.ylabel('accuracy')
plt.scatter(k_range, scores)
plt.xticks([0.5,10,15,20]);
```

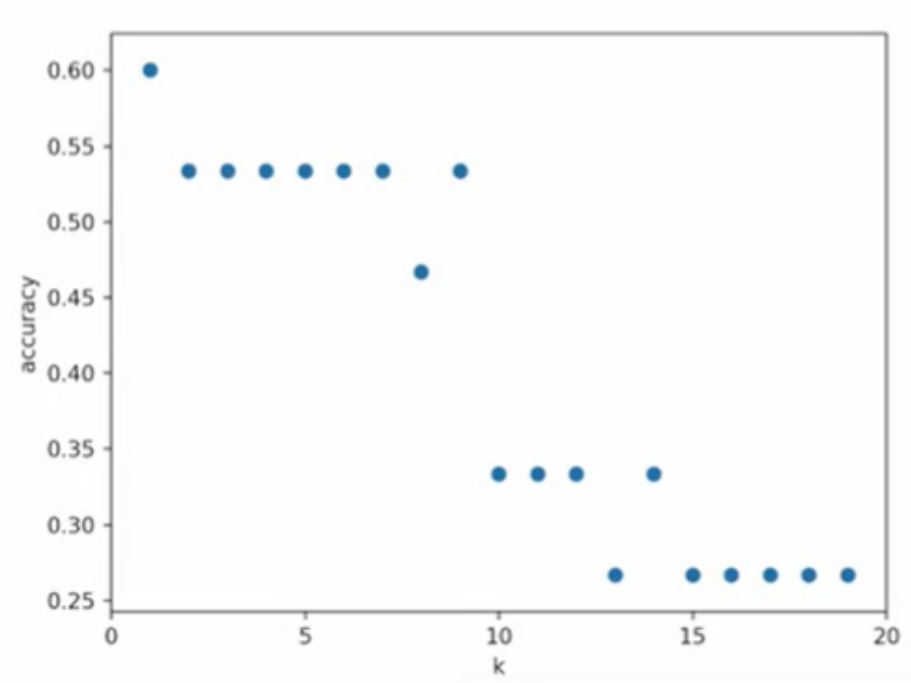


Figure 1.6: Plot accuracy vs k of k-NN classifier

We see that, indeed, larger values of k do lead to worse accuracy for this particular dataset and fixed single train test split. In general, the best choice of the value of k, that is the one that leads to the highest accuracy, can vary greatly depending on the data set.

Chapter 2

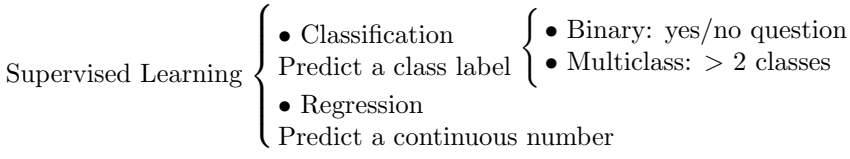
Supervised Machine Learning

This chapter delves into a wider variety of supervised learning methods for both classification and regression, learning about the connection between model complexity and generalization performance, the importance of proper feature scaling, and how to control model complexity by applying techniques like regularization to avoid overfitting. In addition to k-nearest neighbors, this chapter covers linear regression (least-squares, ridge, lasso, and polynomial regression), logistic regression, support vector machines, the use of cross-validation for model evaluation, and decision trees. Learning objectives:

- Understand how different supervised learning algorithms - in particular, those based on linear models - estimate their own parameters from data to make new predictions.
- Understand the strengths and weaknesses of particular supervised learning methods in order to apply the right algorithm for a given task.
- Apply specific supervised machine learning algorithms in Python with scikit-learn.
- Recognize general principles of supervised machine learning that are common across algorithms, such as the connection between model complexity and generalization performance.
- Apply techniques like regularization, feature scaling, and cross-validation to avoid common pitfalls like under- and overfitting.

Supervised ML is one of the most commonly used and successful types of machine learning. Machine Learning is used whenever we want to predict a certain outcome from given input, and we have examples of input/output pairs. We build a model from these input/output pairs, which comprise our training set. Our goal is to make accurate predictions for new, never-before-seen data.

There are two major types of supervised Machine Learning problems are:



- Generalization: If a model is able to make accurate predictions on unseen data.
- Overfitting: Build a model that is too complex for the amount of information that is not able to generalize to new data.
- Underfitting: Build a model too simple, that is not be able to capture all the aspects of and variability in the data.
- Trade-off between overffiting and underfitting:

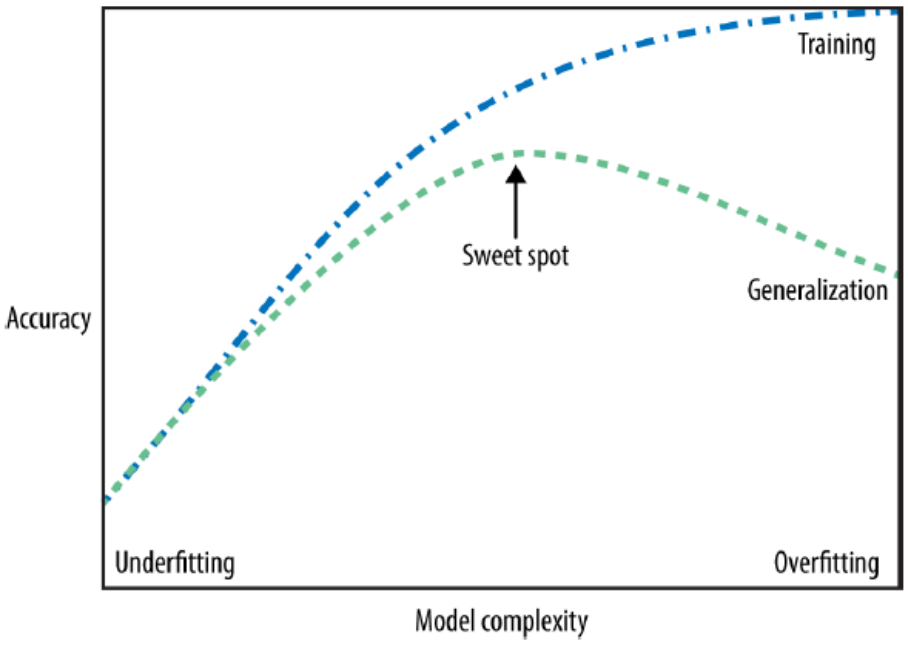


Figure 2.1: Trade-off of model complexity against training and test accuracy

It's important to note that model complexity is intimately tied to the variation of inputs contained in your training dataset: the larger variety of data points your data set contains, the more complex a model you can use with out overfitting. However, simply duplicating the same data points or collecting very similar data will not help.

2.1 Supervised Machine Learning Algorithms

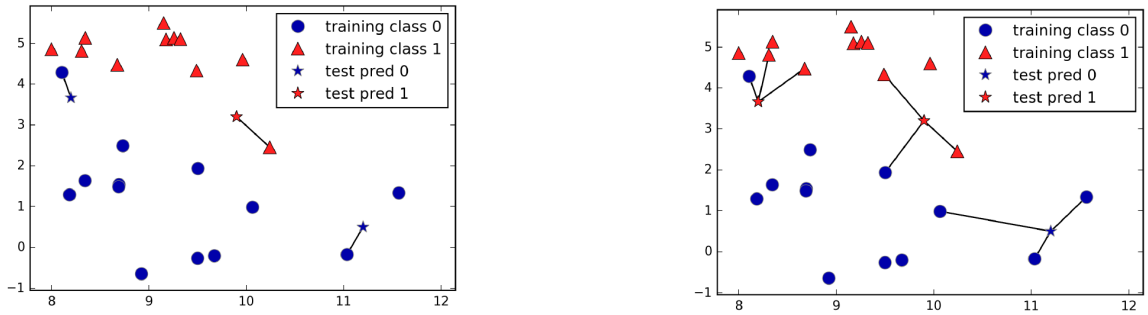
We will review the most popular machine learning algorithms and explain how they learn from data and how they make predictions.

2.1.1 k-Nearest Neighbors

The algorithm finds the closest data points in the training dataset - it's "nearest neighbors."

k-Neighbors classification

The k-NN algorithm considers one or more number of neighbors, which is the closest training data point to the point we want to make a prediction for. this means that for each test point, we count how many neighbors belong to class 0 and how many neighbors belong to class 1. We then assign the class that is more frequent.



(a) one-nearest-neighbor model (b) three-nearest-neighbors model

Figure 2.2: Prediction made by k-NN classification algorithm for different values of k

For two-dimensional datasets, we can also illustrate the prediction for all possible test points in the xy-plane. We color the plane according to the class that would be assigned to a point in this region. This lets us view the *decision boundary* which is the divide between where the algorithm assigns class 0 versus where it assigns class 1.

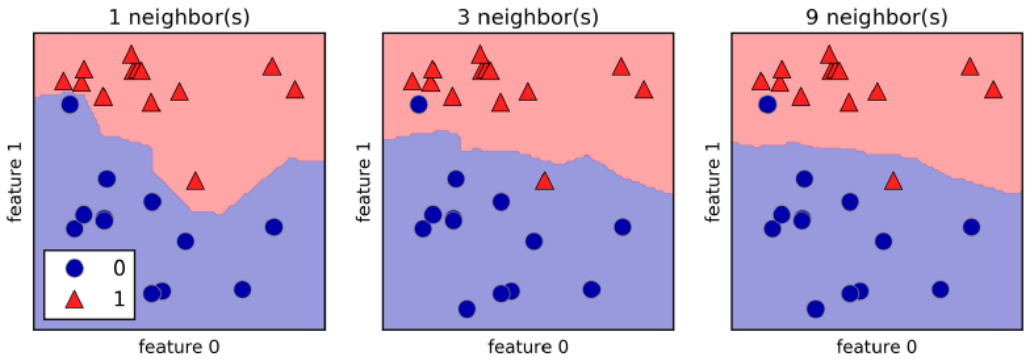


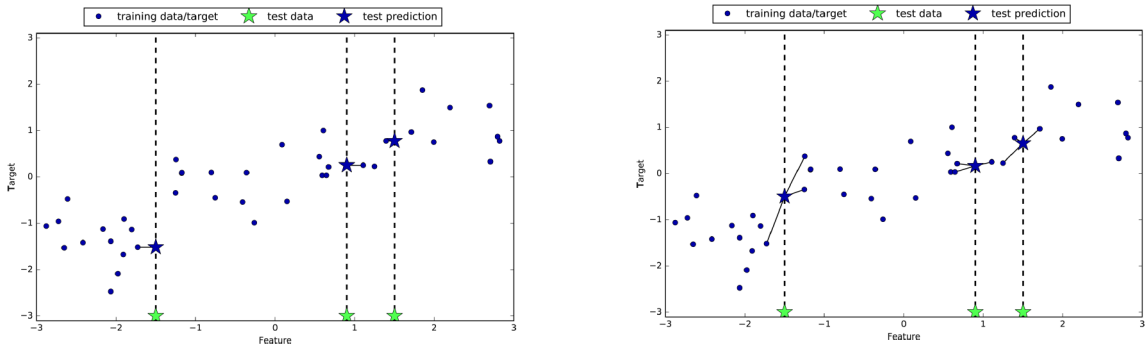
Figure 2.3: Decision boundaries created by the k-NN model for different values of k

As you can see using few neighbors corresponds to high model complexity, and using many neighbors corresponds to low model complexity.

k-Neighbors regression

There is also a regression variant of the k-nearest neighbors algorithm. The prediction using a single neighbor is just a target value of the nearest neighbor, when using multiple nearest neighbors, the

prediction is the average, or mean, of the relevant neighbors.



(a) one-nearest-neighbor regression

(b) three-nearest-neighbors regression

Figure 2.4: Prediction made by k-NN regression algorithm for different values of k

```
from sklearn.neighbors import KneighborsRegressor
reg = KneighborsRegressor(n_neighbors=3)
reg.fit(X_train, y_train)
```

Now we can make evaluations on the test set:

```
y_pred = reg.predict(X_test)
print('Test set prediction:\n{}'.format(np.mean(y_pred==y_test)))
```

We can also evaluate the model using score method, which for regressors return the R^2 score or coefficient of determination, is a measure of goodness of a prediction for a regression model. A value of 1 corresponds to a perfect prediction, and a value of 0 corresponds to a constant model that just predicts the mean of the training set responses, y-train.

```
print('Test set R^2:\n{}'.format(reg.score(X_test, y_test)))
```

Strengths, weaknesses and parameters

There are two important parameters to the Kneighbors classifier:

- The number of neighbors (default typically odd = 5)
- The distance between data points (default: Euclidean distance or Minkowski metric, p=2)

Pros	Cons
Easy to understand Good base line method	With large dataset is prediction can be slow badly with sparse datasets (0)

2.1.2 Linear models

Linear models make a prediction using a *linear function* of the input features.

Linear models for regression

The general prediction formula for regression is:

$$y = w[0] * x[0] + w[1] * x[1] + ... + w[p] * x[p] + b \tag{2.1}$$

- p = number of features
- w,b = parameters of the model
- y = prediction of the model

For a dataset with a single feature, this is:

$$y = w[0] * x[0] + b -> lineequation \tag{2.2}$$

where:

- w[0] = slope or coef attribute
- b = axis offset or y-intercept attribute

There are many different linear models for regression. The difference between these models lies in how the model parameters w and b are learned from the training data, and how model complexity can be controlled. we will take a look at the most popular linear models fro regression.

Linear Regression (aka ordinary least squares - OLS)

The simplest and most classic method fro regression. This finds the parameters w and b that minimize the *mean square error* between predictions and the true regression targets, y, on the training set. The mean square error is the sum of the differences between the predictions and the true values. Linear regression has no parameters, which is a benefit, but it also has no way to control model complexity.

$$RSS(w, b) = \sum_{i=1}^{N} (y_i - (w * x_i + b))^2 \tag{2.3}$$

where:

- y_i is the training set target value
- $(y_i - (w * x_i + b)$ is the predicted target value using model

```
from sklearn.linear_model import LinearRegression
Linreg = LinearRegression().fit(X_train,y_train)
```

The parameters w and b are stored in:

```
print('linear model coeff (w): {:.2f}'.format(Linreg.coef_))
print('linear model intercept (b): {:.2f}'.format(Linreg.intercept_))
```

Let's look at the training and test set performance:

```
print('R-squared score (training): {:.2f}'.format(Linreg.score(X_train, y_train)))
print('R-squared score (test): {:.2f}'.format(Linreg.score(X_test, y_test)))
```

With higher-dimensional datasets or datasets with a large number of features, linear models become more powerful, and there is a higher chance of overfitting, and therefore we should try to find a model that allows us to control complexity. One of the most commonly used alternatives to standard linear regression is *ridge regression*.

Ridge Regression

Ridge is a more restricted model, so we are less likely to overfit. The formula it uses to make predictions is the same one used for OLS. In ridge regression, though, the coefficient w are chosen not only so that they predict well on the training data, but also to fit an additional constraint. We also want the magnitude of coefficients to be as small as possible; in other words, all entries of w should be close to zero, this means each feature should have as little effect on the outcome as possible which translates to having a small slope, while still predicting well. This constraint is an example of what is called *regularization*. Regularization means explicitly restricting a model to avoid overfitting. The particular kind used by ridge regression is know as L2 regularization[1].

$$RSS_{Ridge}(w,b) = \sum_{i=1}^{N}(y_i - (w * x_i + b)^2 + \alpha \sum_{j=1}^{p} w_j^2 \tag{2.4}$$

```
from sklearn.linear_model import Ridge
Linridge = Ridge(alpha = 20).fit(X_train, y_train)
print('Ridge regression linear model coeff (w): {}'.format(Linreg.coef_))
print('Ridge regression linear model intercept (b): {}'.format(Linreg.intercept_))
print('Training set score: {:.2f}'.format(Linridge.score(X_train, y_train)))
print('Test set score: {:.2f}'.format(Linridge.score(X_test, y_test)))
print('Numer of non-zero features: {}'.format(np.sum(linridge.coef_ != 0)))
```

This model makes a trade-off between the simplicity of the model (near-zero coefficients) and its performance on the training set. How much importance the model places on simplicity versus training set performance can be specified by the use, using the *alpha* parameter. The optimum setting of *alpha* depends on the particular dataset we are using.

The need for feature normalization:

- important for some ML methods that all features are on the same scale for faster convergence in learning. (e.g. regularized regression, k-NN, SVM, neuronal networks, ...)
- Depend on the data. MinMax scaling:

$$x_i^{'} = (x_i - x_i^{MIN})/(x_i^{MAX} - x_i^{MIN}) \tag{2.5}$$

Example of Ridge regression with feature normalization:

[1] Mathematically, Ridge penalizes the L2 norm of the coefficients, or Euclidean length of w

```
from sklearn.preprocessing import MaxMinScaler
scaler = MinMaxScaler()
#it's more efficient to do fitting and transforming together on the training set
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
linridge = Ridge(alpha = 20).fit(X_train_scaled, y_train)
print('Training set score: {:.2f}'.format(Linridge.score(X_train_scaled, y_train)))
print('Test set score: {:.2f}'.format(Linridge.score(X_test_scaled, y_test)))
print('Numer of non-zero features: {}'.format(np.sum(linridge.coef_ != 0)))
```

$$\text{Increasing } \alpha \rightarrow \downarrow \text{ Training set } \uparrow \text{ Test set or generalization} \tag{2.6}$$

Lasso

As with ridge regression, using the lasso also restricts coefficients to be close to zero, but in a slightly different way, called L1 regularization.[2] The consequence of L1 regularization is that when using lasso, some coefficients are *exactly zero*. This means some features are entirely ignored by the model. This can be seen as a form of automatic feature selection. Having some coefficients be exactly zero often makes a model easier to interpret, and can reveal the most important features of your model.

$$RSS_{Lasso}(w,b) = \sum_{i=1}^{N}(y_i - (w * x_i + b)^2 + \alpha \sum_{j=1}^{p} |w_j| \tag{2.7}$$

```
from sklearn.Linear_model import Lasso
linlasso = Lasso().fit(X_train, y_train)
print('Training set score: {:.2f}'.format(linlasso.score(X_train, y_train)))
print('Test set score: {:.2f}'.format(linlasso.score(X_test, y_test)))
print('Numer of features used: {}'.format(np.sum(linlasso.coeff_ != 0)))
```

Similarly to Ridge, the Lasso has also a regularization parameter, alpha, that controls how strongly coefficients are pushed toward zero. When we use alpha, we also need to increase the default setting of the maximum number of iteration to run.

```
linlasso001 = Lasso(alpha=0.01, max_iter=10000).fit(X_train, y_train)
print('Training set score: {:.2f}'.format(linlasso001.score(X_train, y_train)))
print('Test set score: {:.2f}'.format(linlasso001.score(X_test, y_test)))
print('Numer of features used: {}'.format(np.sum(linlasso001.coeff_ != 0)))
```

A lower alpha like 0.01 allowed us to fit more complex model, which worked better on the training and test data, but if we set alpha too low like 0.0001, however, we again remove the effect of regularization and end up overfitting. Also we could use scaling on our data.

In practice, ridge regression is usually the first choice between these two models. However, if you have a large amount of features and expect only a few of them to be important, Lasso might be a better choice, Lasso also will provide a model that is easier to understand, as it will select only a subset of the input features.

[2]The lasso penalizes the L1 norm of the coefficient vector - or in other words, the sum of the absolute values of coefficients.

Polynomial models for Regression

When we add this polynomial features we capture interactions between the original features by adding them as features to the linear model. This make a classification problem easier.

For example if we had a set of two-dimensional data points with features x_0 and x_1. Then we could transform each data point by adding additional features that were the three unique multiplicative combinations of x_0 and x_1. So we've transformed our original two-dimensional points into a set of five-dimensional points that rely only on the information in the two-dimensional points.

$$\begin{aligned} x = (x_0, x_1) \rightarrow x^{'} &= (x_0, x_1, x_0^2, x_0 x_1, x_1^2) \\ y = w_0 x_0 + w_1 x_1 + w_{00} x_0^2 + w_{01} x_0 x_1 + w_{11} x_1^2 + b & \end{aligned} \tag{2.8}$$

Generate new features consisting of all polynomial combinations of the original two features (x_0, x_1). The degree of the polynomial specifies how many variables participate at the time in each new feature (e.g. degree = 2). This is still a weighted linear combination of features, so it's a linear model.

```
from sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures(degree = 2)
X_F1_poly = poly.fit_transform(X_F1)
X_train, X_test, y_train, y_test = train_test_split(X_F1_poly, y_F1, random_state = 0)
Linreg = LinearRegression().fit(X_train, y_train)
print('Training set score: {:.2f}'.format(Linreg.score(X_train, y_train)))
print('Test set score: {:.2f}'.format(Linreg.score(X_test, y_test)))
```

Addition of many polynomial features often leads to overfitting, so we use polynomial features in combination with regression that has a regularization penalty like ridge regression.

Linear models for classification

Linear models are also extensively used for classification. Logistic Regression is made using the logistic function that transform re-valued input to an output number between 0 and 1.

$$y = \frac{1}{1 + \exp[-(b + w_0 x_0 + w_1 x_1 + ... + w_n x_n)]} \tag{2.9}$$

The formula looks very similar to the one for linear regression, but instead of just returning the weighted sum of the features, we threshold the predicted value to zero.

- $y < 0 \rightarrow class - 1$
- $y > 0 \rightarrow class + 1$

For linear models for regression, the output, y, is a linear function of the features: a line, plane, or hyper plane (in higher dimensions). For linear models for classification, the *decision boundary* is a linear function of the input. For example, a linear classifier is a classifier that separates two classes using a line.

There are many algorithms for learning linear models. These algorithms all differ in:

- the way in which they measure how well a particular combination of coefficients and intercept fits the training data.

- if and what kind of regularization they use.

The two most common algorithms are:

$$\text{Linear Classification algorithms} \begin{cases} \bullet \text{ Logistic Regression} \\ \bullet \text{ Linear Support Vector Machine (Linear SVM's)} \end{cases}$$

Both models apply an L2 regularization, in the same way that Ridge does for regression. The trade-off parameter that determines the strength of the regularization is called C.

- higher C $\rightarrow$ less regularization, overfitting
- low C $\rightarrow$ more regularization, try to adjust to the majority of data points.

Code for Logistic regression:

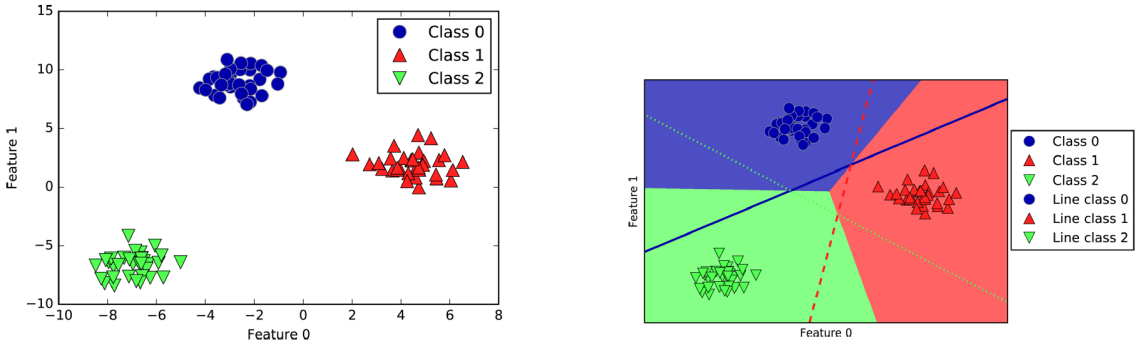
```
from sklearn.linear_models import LogisticRegression
clf = LogisticRegression(C=100).fit(X_train, y_train)
print('Training set score: {:.2f}'.format(clf.score(X_train, y_train)))
print('Test set score: {:.2f}'.format(clf.score(X_test, y_test)))
```

Code for Linear SVM:

```
from sklearn.svm import LinearSVC
clf = LinearSVC(C=100).fit(X_train, y_train)
print('Training set score: {:.2f}'.format(clf.score(X_train, y_train)))
print('Test set score: {:.2f}'.format(clf.score(X_test, y_test)))
```

Linear models for Multiclass classification

Many linear classification models are for binary classification only, and don't extend naturally to the multiclass case (with exception of logistic regression). A common technique to extend a binary classification algorithm to multiclass classification algorithm is the *one-vs-rest* approach. In one-vs-rest approach, a binary model is learned for each class that tries to separate that class from all of the other classes, resulting in as many binary models as there are classes. To make prediction, all binary classifiers are run on a test point. The classifier that has the highest score on its single class "wins", and this class label is returned as the prediction.



(a) Dataset containing three classes

(b) Multiclass decision boundaries

Figure 2.5: three one-vs-rest classifiers

Strengths, weaknesses and parameters

The main parameters of liner models is the regularization parameter, called alpha in the regression models and C in Logistic regression and Linear SVM's. Large values for alpha or small values for C mean simple models. The other decision you have to make is whether you want to use L1 regularization or L2 regularization. If you assume that only a few of your features are actually important, you should use L1. Otherwise, you should default to L2.

Pros	Cons
Simple and easy to train, and predict	Coefficients might be hard to interpret
Scale well to very large datasets	For lower dimensional data use other models
Work well with sparse data	

2.1.3 Decision Trees

Decision trees are widely used models for classification and regression tasks. Essentially, they learn a hierarchy of if/else questions, leading to a decision.

In a decision tree, each node in the tree either represents a question or a terminal node (also called a *leaf*) that contains the answer. The edges connect the answers to a question with the next question you would ask. The top node, also called the *root*, represents the whole dataset.

A prediction on a new data point is made by checking which region of the partition of the feature space the point lies in, and then predicting the majority target in that region.The region can be found by traversing the tree from the root and going left or right, depending on whether the test is full filled or not.

There are two common strategies to prevent overfitting:

- stopping the creation of the tree early (*pre-pruning*)
- building a tree but then removing or collapsing nodes that contain little information (*post-pruning*)

Possible criteria for pre-prunning include limiting the maximum depth of the tee, limiting the maximum number of leaves, or requiring minimum number of points in a node to keep splitting it.

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
from adspy_shared_utilities import plot_decision_tree
from sklearn.model_selection import train_test_split

iris_load_iris()
X_train, X_test,
y_train, y_test = train_test_split(iris.data, iris.target, random_state=3)

tree = DecisionTreeClassifier().fit(X_train, y_train)

#Setting max decision tree depth to help avoid overfitting
#other max_leaf_nodes or min_samples_leaf
tree2 = DecisionTreeClassifier(max_depth=3).fit(X_train, y_train)

print('Training set score: {:.2f}'.format(tree.score(X_train, y_train)))
print('Test set score: {:.2f}'.format(tree.score(X_test, y_test)))
print('Training set score with max_depth: {:.2f}'
.format(tree2.score(X_train, y_train)))
print('Test set score with max_depth: {:.2f}'.format(tree2.score(X_test, y_test)))

#Visualizing decision trees
plot_decision_tree(tree, iris.features_names, iris.target_names)
```

Feature importance: How important is a feature to overall prediction accuracy? A number between 0 and 1 assigned to each feature and indicates how important that feature is to overall prediction accuracy.

- feature importance of 0 means that the feature is not use at all in the prediction
- feature importance of 1 means that feature perfectly predicts the target

```
from adspy_shared_utilities import plot_feature_importances
plt.figure(figsize=(10,4))
plot_feature_importances(tree, iris.feature_names)
plt.show()
print('Feature importances: {}'.format(tree.feature_importances_))
```

Strengths, weaknesses and parameters

Pros	Cons
Easily visualized an interpreted	they tend to overfit
No feature normalization or scaling	Usually need an ensemble of trees
Work well with mixed datasets	

2.1.4 Kernelized Support Vector Machines

This are an extension of Linear SVM that allows for more complex models that are no defined simply by lines or hyperplanes. We saw how linear SVM classifiers could effectively find a decision

boundary with maximum margin. But with real data, many classification problems aren't this easy. With the different classes located in future space in a way that a line or hyperplane can't act as an effectively classifier. In essence, Kernelized SVM's do, is they take the original input data space and transform it to a new higher dimensional feature space, where it becomes much easier to classify the transform to data using linear classifier.

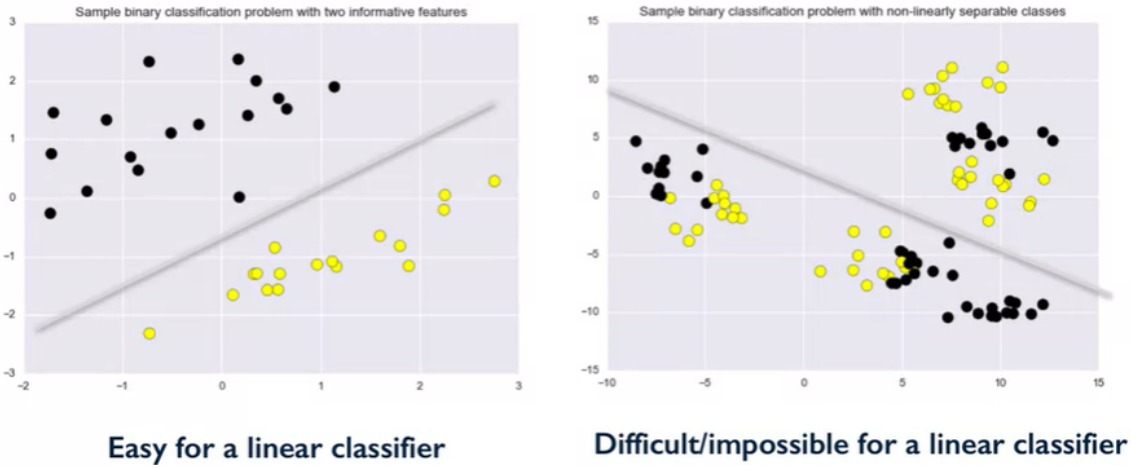


Figure 2.6: Two-class classification dataset in which classes are and aren't linearly separable

There are different kernels available for kernelized SVM correspond to different possible transformations we could apply to data. Here we're going to focus mainly on what's called the *radial basis function kernel - RBF*, there is also something called *polynomial kernel*.

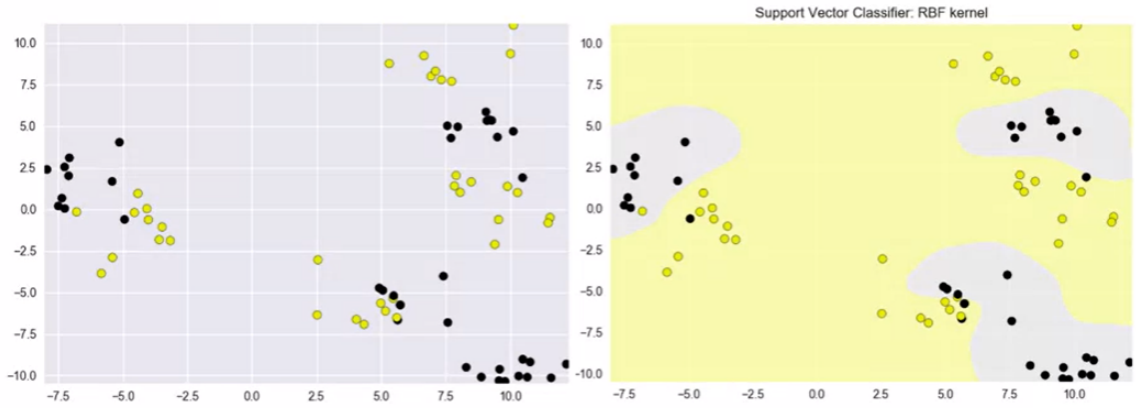


Figure 2.7: Applying the SVM with RBF Kernel

```
from sklearn.svm import SVC
from adspy_shared_utilities import plot_class_regions_for_classifier
```

```
X_train, X_test, y_train, y_test = train_test_split(X_D2, y_D2, random_state=0)
#There is a SVC parameter called kernel by default, the SVM will use the RBF
plot_class_regions_for_classifier(SVC().fit(X_train, y_train), X_train, y_train,
None, None, 'Support Vector Classifier: RBF kernel'))
#Here we use polynomial kernel instead of RBF kernel
plot_class_regions_for_classifier(SVC(kernel='poly', degree=3)
.fit(X_train, y_train), X_train, y_train, None, None,
'Support Vector Classifier: \ Polynomial kernel, degree = 3'))
```

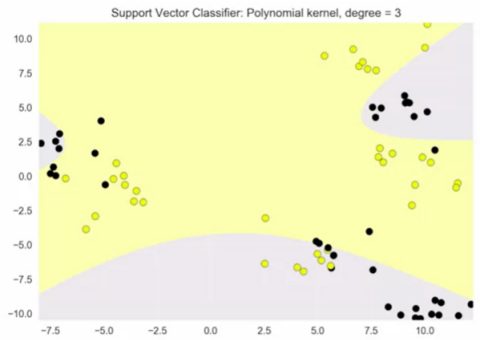


Figure 2.8: Applying the SVM with polynomial kernel

Example of SVM with RBF kernel, gamma parameter:

```
from adspy_shared_utilities import plot_class_regions_for_classifier

X_train, X_test, y_train, y_test = train_test_split(X_D2, y_D2, random_state = 0)
fig, subaxes = plt.subplots(3, 1, figsize=(4, 11))

for this_gamma, subplot in zip([0.01, 1.0, 10.0], subaxes):
    clf = SVC(kernel = 'rbf', gamma=this_gamma).fit(X_train, y_train)
    title = 'Support Vector Classifier: \nRBF kernel, gamma = {:.2f}'
    .format(this_gamma)
    plot_class_regions_for_classifier_subplot(clf, X_train, y_train,
                                              None, None, title, subplot)
    plt.tight_layout()
```

Example of SVM with RBF kernel, using both C and gamma parameter:

```
from sklearn.svm import SVC
from adspy_shared_utilities import plot_class_regions_for_classifier_subplot

from sklearn.model_selection import train_test_split


X_train, X_test, y_train, y_test = train_test_split(X_D2, y_D2, random_state = 0)
```

```
fig, subaxes = plt.subplots(3, 4, figsize=(15, 10), dpi=50)

for this_gamma, this_axis in zip([0.01, 1, 5], subaxes):

    for this_C, subplot in zip([0.1, 1, 15, 250], this_axis):
        title = 'gamma = {:.2f}, C = {:.2f}'.format(this_gamma, this_C)
        clf = SVC(kernel = 'rbf', gamma = this_gamma,
                  C = this_C).fit(X_train, y_train)
        plot_class_regions_for_classifier_subplot(clf, X_train, y_train,
                                                  X_test, y_test, title,
                                                  subplot)
        plt.tight_layout(pad=0.4, w_pad=0.5, h_pad=1.0)
```

Example of SVM normalized data with feature preprocessing using minmax scaling:

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

clf = SVC(C=10).fit(X_train_scaled, y_train)
print('Breast cancer dataset (normalized with MinMax scaling)')
print('RBF-kernel SVC (with MinMax scaling) training set accuracy: {:.2f}'
     .format(clf.score(X_train_scaled, y_train)))
print('RBF-kernel SVC (with MinMax scaling) test set accuracy: {:.2f}'
     .format(clf.score(X_test_scaled, y_test)))
```

Strengths, weaknesses and parameters

There are three important parameters that controls model complexity for Kernelized SVM's:

- The type of kernel function (e.g. RBF, polynomial)
- Parameters of each kernel (e.g. gamma (γ): in RBF).
- C: regularization parameter. C and gamma both control the complexity of the model, with large values in either resulting in a more complex model. C and gamma should be adjusted together.

Pros	Cons
Perform well on a variety of datasets	Runtime speed and memory usage decreases
Versatile: different kernel functions	Needs careful normalization of input data
Works well on low and high-dimensional data	Difficult to interpret

Cross-Validation

Cross-validation is a statistical method of evaluating generalization performance that is more stable and through than using a split into a training and test set. In cross-validation, the data is instead split repeatedly and multiple models are trained. The most commonly used version of

cross-validation is *k-fold cross-validation* where k is a user-specified number, usually 5 or 10. When performing five-fold cross validation, the data is first partitioned into five parts of (approximately) equal size, called *folds*. Next, a sequence of models is trained. The first model is trained using the first fold as the test set, and the remaining folds(2-5) are used as the training set. Then another model is built, this time using fold 2 as the test set and the data in folds(1,3,4 and 5) as the training set. This process is repeated using folds 3, 4 and 5 as test sets. In the end, we have collected five accuracy values, one per fold.

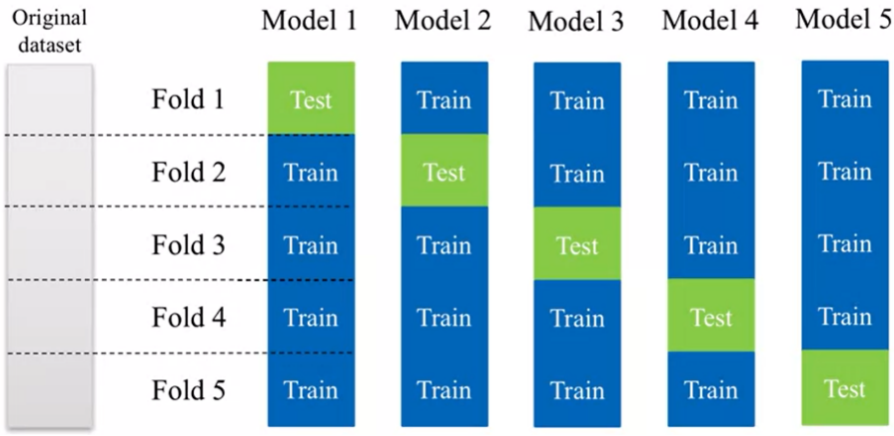


Figure 2.9: Cross-validation (5-fold)

```
from sklearn.model_selection import cross_val_score

clf = KNeighborsClassfier(n_neighbors = 5)
X = X_fruits_2d.as_matrix()
y = y_fruits_2d.as_matrix()
cv_scores = cross_val_score(clf, X, y)

#By default cross-validation does threefold cross-validation.
#We can change the number of folds used by changing the cv parameter
print('Cross-validation scores (3-fold):', cv_scores)
#It's typical to then compute the mean of all the accuracy scores across
#the folds, and report as a measure of how accuracy we can expect the model
#to be on average
print('Mean cross-validation score (3-fold): {:.3f}'.format(np.mean(cv_scores)))
```

When you ask scikit-learn to do cross-validation for a classification task, it actually does what is called "Stratified k-fold Cross-validation" meaning that when splitting the data, the proportions of classes in each fold are made as close as possible to the actual proportions of the classes in the overall data. For regression scikit-learn uses regular k-fold validation since the concept of preserving class proportions isn't something that's really relevant for regression problems.

Sometimes we want to evaluate the effect that an important parameter of a model has on the cross-validation scores. The useful function validation curve makes it easy to run this type of

experiment.

```
from sklearn.svm import SVC
from sklearn.model_selection import validation_curve

param_range = np.logspace(-3, 3, 4)
train_scores, test_scores = validation_curve(SVC(), X, y,
                                             param_name='gamma',
                                             param_range=param_range, cv=3)
# print(train_scores)
# print(test_scores)

# This code based on scikit-learn validation_plot example
# See:  http://scikit-learn.org/stable/auto_examples/
# model_selection/plot_validation_curve.html
plt.figure()

train_scores_mean = np.mean(train_scores, axis=1)
train_scores_std = np.std(train_scores, axis=1)
test_scores_mean = np.mean(test_scores, axis=1)
test_scores_std = np.std(test_scores, axis=1)

plt.title('Validation Curve with SVM')
plt.xlabel('$\gamma$ (gamma)')
plt.ylabel('Score')
plt.ylim(0.0, 1.1)
lw = 2

plt.semilogx(param_range, train_scores_mean, label='Training score',
             color='darkorange', lw=lw)

plt.fill_between(param_range, train_scores_mean - train_scores_std,
                 train_scores_mean + train_scores_std, alpha=0.2,
                 color='darkorange', lw=lw)

plt.semilogx(param_range, test_scores_mean, label='Cross-validation score',
             color='navy', lw=lw)

plt.fill_between(param_range, test_scores_mean - test_scores_std,
                 test_scores_mean + test_scores_std, alpha=0.2,
                 color='navy', lw=lw)

plt.legend(loc='best')
plt.show()
```

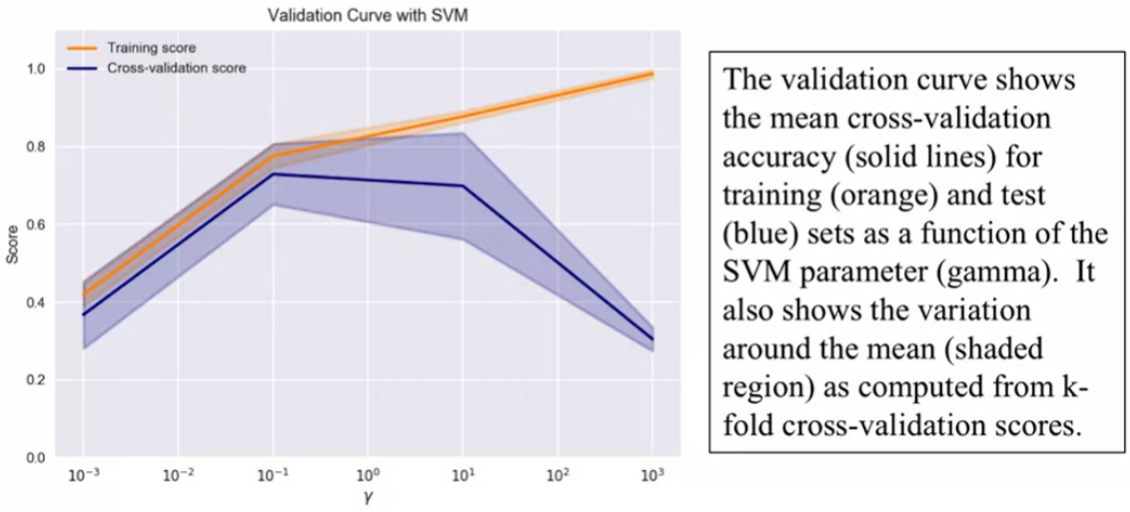


Figure 2.10: Validation Curve

Finally as a reminder, cross-validation is used to evaluate the model and not learn or tune a new model. To do model tuning, we'll look at how to tune the models parameters using something called "Grid Search"

Chapter 3

Model Evaluation and Selection

This chapter covers evaluation and model selection methods that you can use to help understand and optimize the performance of your machine learning models

- Understand why accuracy alone can be an inadequate metric for getting a more complete picture of a classifier's performance.
- Understand the motivation and definition of a variety of important evaluation metrics in machine learning and how to interpret the results of using a given evaluation metric.
- Optimize a machine learning algorithm using a specific evaluation metric appropriate for a given task.

3.1 Introduction

When we began working with supervised machine learning methods, we evaluated a classifier's performance using its accuracy. Accuracy, is the fraction of samples that were classified correctly. That is, where the classifier's predicted label matched the correct or true label. We also evaluated a regression model's performance using the default r squared metric. In this chapter, you'll learn why measures like accuracy, which are simple and easy to understand, also have drawbacks. In that they don't give a complete enough picture of a supervised learning model's performance. And may not be the right metric for measuring success in your application. So we're going to cover several additional evaluation metrics beyond accuracy to get a better picture of how well A supervised model is doing on a given data set. You'll also learn how about to choose the right evaluation matrix for your application that can help you select the best model or find the optimal parameters.

So we're going to cover several additional evaluation metrics beyond accuracy. We'll see how they're defined, what the motivation is for using them, and how to use them scikit-learn to get a better picture of how well A supervised model is doing on a given data set. You'll also learn how about to choose the right evaluation matrix for your application that can help you select the best model or find the optimal parameters.

So let's return for a moment to this workflow diagram that we introduced earlier in the course (Fig.1.1). Once a model is trained, the evaluation step provides critical feedback on the trained

model's performance characteristics. Particularly those that might be important for your application. The results of the evaluation step, for example, might help you understand which data instances are being classified or predicted incorrectly. Which might in turn suggest better features or different kernel function or other refinements to your learning model in the feature and model refinement phase. The objective function that's optimized during the training phase may be a different, what's called a surrogate metric. That's easier to use in practice for optimization purposes than what's used for the evaluation metric. So the evaluation measures are the ones that in the end are used to select between different trained models or settings. So it's very important to:

- Choose evaluation methods that match the goal of your application.
- Compute your selected evaluation metric for multiple different models.
- Then select the model with the best value of evaluation metric.

Before we start defining and using some evaluation metrics for binary classification, multi-class classification or regression. Lets start by looking at example of why just looking at accuracy may not be enough to gain a good picture of what a classifier's doing.

Accuracy with Imbalanced Class Scenario, let's consider the case where we have a binary classification task

- Suppose you have two classes:
  - Relevant (R): the positive class
  - Not Relevant (N): the negative class
- Out of 1000 randomly selected items, on average
  - One item is relevant and has a R label
  - The rest of the items (999 of them) are not relevant and labelled N
- Recall that:

$$Accuracy = \frac{\#\text{correct predictions}}{\#\text{total instances}}$$

Let's suppose you develop a classifier for predicting relevant e-commerce items. And after you've finished the development, you measure its accuracy on the test set to be 99.9%. At first, that might seem to be amazingly good, right? That's incredibly close to perfect. But let's compare that with a "Dummy" classifier that didn't look at the features at all, and always just blindly predicted the most frequent class (i.e. the negative N class or not relevant class). And so the accuracy of the dummy classifier is also going to be 99.9%. So in reality our own classifier's performance isn't impressive at all. It's no better than just always predicting The majority class without even looking at the data.

scikit-learn makes it easy to create a dummy classifier just by using the DummyClassifier class as shown here. Dummy classifiers, again, are called that because they don't even look at the input data to make a prediction. They simply use the strategy or rule of thumb that you instruct them to use, when creating them. In fact, when you create the classifier, you set the strategy argument to tell it what rule of thumb to use to make its predictions. So here, we set this to the most frequent strategy to predict the most frequent class. The dummy classifier provides what is called a null

accuracy baseline. That is the accuracy that can be achieved by always picking the most frequent class. You should not use a dummy classifier for real classification problems, but it does provide a useful sanity check in point of comparison.

Some commonly-used settings for the `strategy` parameter for DummyClassifier in sciKit-learn:

- most_frequent: predicts the most frequent label in the training set.
- stratified: random predictions based on training set class distribution.
- uniform: generates predictions uniformly at random.
- constant: always predicts a constant label provided by the user.
  - a major motivation of this method is F1-scoring, when the positive class is in the minority.

So what does it mean if we discover that our classifier has close to the DummyClassifier's performance? While typically it means:

- That the features in our model may be ineffective, or erroneously computed or missing for some reason
- It could also be caused by a poor choice of kernel or hyperparameter in the model.
- Large class imbalance

In general, for imbalanced classification problems, you should use metrics other than accuracy.

Dummy Regressors, they serve a similar role as a null outcome baseline and sanity check for regression models. `strategy` parameter options:

- mean: predicts the mean of the training targets.
- median: predicts the median of the training targets.
- quantile: predicts a user-provided quantile of the training targets.
- constant: predicts a constant user-provided value.

3.2 Metrics for Binary Classification

Binary predictions outcomes with a positive and negative class, there are four possible outcomes that we can break into two cases corresponding to the first and second row of this matrix.

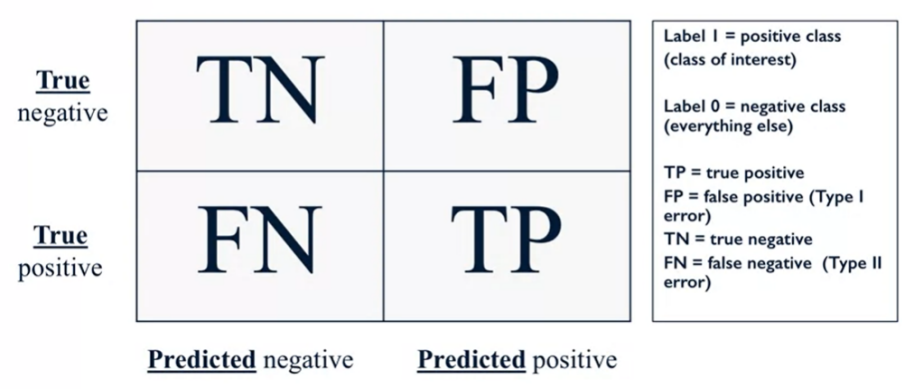


Figure 3.1: Binary Outcomes matrix

With a positive and negative class, there are four possible outcomes that we can break into two cases corresponding to the first and second row of this matrix. If the true label for an instance is negative, the classifier can predict either negative, which is correct, and call the true negative. Or it can erroneously predict positive, which is an error and called a false positive. If the true label for an instance is positive, the classifier can predict either negative, which is an error and called a false negative. Or it can predict positive, which is correct and that's called a true positive.

So maybe a quick way to remember this is that the first word in these matrix cells is false, if it's a a classifier error, or true if it's a classifier success. The second word is negative if the true label is negative and positive if the true label is positive.

This matrix of all combinations of predicted label and true label is called a confusion matrix.

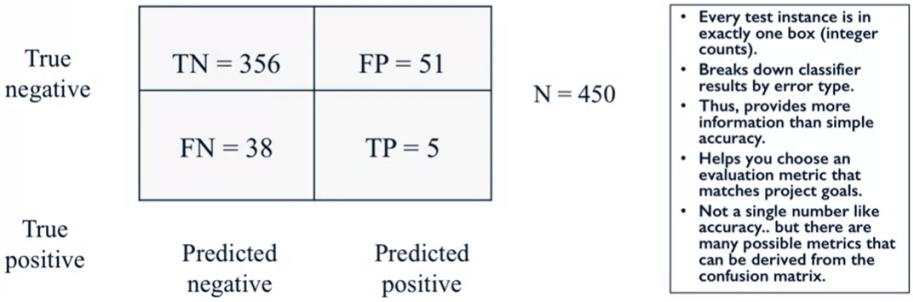


Figure 3.2: Confusion matrix

In particular, the successful predictions of the classifier are on the diagonal where the true class matches the predicted class. The cells off the diagonal represent errors of different types.

```
# Binary confusion matrix
from sklearn.metrics import confusion_matrix

# Negative class (0) is most frequent
```

```
dummy_majority = DummyClassifier(strategy = 'most_frequent').fit(X_train, y_train)
y_majority_predicted = dummy_majority.predict(X_test)
confusion = confusion_matrix(y_test, y_majority_predicted)

print('Most frequent class (dummy classifier)\n', confusion)

# produces random predictions w/ same class proportion as training set
dummy_classprop = DummyClassifier(strategy='stratified').fit(X_train, y_train)
y_classprop_predicted = dummy_classprop.predict(X_test)
confusion = confusion_matrix(y_test, y_classprop_predicted)

print('Random class-proportional prediction (dummy classifier)\n', confusion)

svm = SVC(kernel='linear', C=1).fit(X_train, y_train)
svm_predicted = svm.predict(X_test)
confusion = confusion_matrix(y_test, svm_predicted)

print('Support vector machine classifier (linear kernel, C=1)\n', confusion)

from sklearn.linear_model import LogisticRegression

lr = LogisticRegression().fit(X_train, y_train)
lr_predicted = lr.predict(X_test)
confusion = confusion_matrix(y_test, lr_predicted)

print('Logistic regression classifier (default settings)\n', confusion)

from sklearn.tree import DecisionTreeClassifier

dt = DecisionTreeClassifier(max_depth=2).fit(X_train, y_train)
tree_predicted = dt.predict(X_test)
confusion = confusion_matrix(y_test, tree_predicted)

print('Decision tree classifier (max_depth = 2)\n', confusion)
```

So first, we'll apply the most frequent class DummyClassifier we saw earlier. What we can see here is that the right column, that represent cases where the classifier predicted the positive class, is all zero. Which makes sense for this dummy classifier because it's always predicting the negative class, the most frequent one. We see that 407 instances are true negatives, and there are 43 errors that are false negatives.

Here we apply the stratified DummyClassifier that gives random output in proportion to the ratio labels in the training set. Now the right column is no longer all zero because this DummyClassifier does predict occasionally predict the positive class. If we add the numbers in the right column, we see that 32 plus 6 equals 38 times the number of times the classifier predicted the positive class. Of those times, in six cases, the lower right diagonal, this was a true positive.

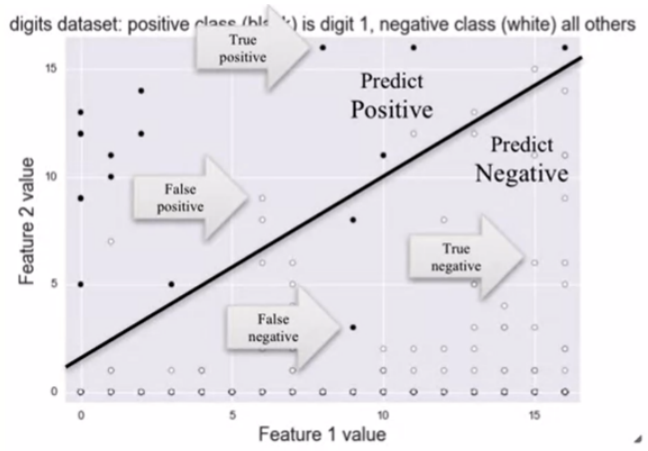
We'll apply a support vector classifier with linear kernel and seed parameter equal to one. We note that looking along the diagonal compared to the stratified dummy classifier above, which had

a total of 375 plus 6, or 381 correct predictions. The support vector classifier has a total of 402 plus 38, which is 440 correct predictions on the same data set.

Likewise, we can apply a logistic regression classifier, and that obtains similar results to the support vector classifier. And finally, we can apply a decision tree classifier, and look at the confusion matrix that results from that. One thing we notice is, that unlike the support vector or logistic regression classifier, which had balanced numbers of false negatives and false positives. The decision tree makes more than twice as many false negative errors, 17 of them actually, as false positive errors, of which there are 7. Now we known how a confusion matrix can give us a little more information about the types of errors a classifier makes.

3.2.1 Confusion Matrices & Basic Evaluation Metrics

Our original motivation for creating this matrix was to go beyond a single number accuracy, to get more insight into the different types of predictions successes and failures of a given classifier. Let's look at this classification result visually to help us connect these four numbers to a classifier's performance.



TN = 429	FP = 6
FN = 2	TP = 13

Figure 3.3: Visualization of different error types

What I've done here is plot the data instances by using two specific feature values out of the total 64 feature values that make up each instance in the digits dataset. The black points here are the instances with true class positive namely the digit one and the white points have true class negative, that is, there are all the other digits except for one. The black line shows a hypothetical linear classifier's decision boundary for which any instance to the left of the decision boundary is predicted to be in the positive class and everything to the right of the decision boundary is predicted to be in the negative class. The true positive points are those black points in the positive prediction region and false positives are those white points in the positive prediction region. Likewise, true negatives are the white points in the negative prediction region and false negatives are black points in the negative prediction region. There are many ways to summarize the information in the confusion matrix.

Accuracy

We've already seen one metric that can be derived from the confusion matrix counts namely accuracy. Accuracy: for what fraction of all instances is the classifier's prediction correct (for either positive or negative class)?

$$Accuracy = \frac{TN + TP}{TN + TP + FN + FP} \tag{3.1}$$

Classification Error

Let's look at some other evaluation metrics we can compute from these four numbers. Well, a very simple related number that's sometimes used is classification error. Classification error: for what fraction of all instances is the classifier's prediction incorrect?

$$ClassificationError = \frac{FP + FN}{TN + TP + FN + FP} \rightarrow 1 - Accuracy \tag{3.2}$$

Recall or True Positive Rate (TPR)

When we wanted an evaluation metric that would give higher scores to classifiers that not only achieved the high number of true positives but also avoided false negatives. That is, that rarely failed to detect a true. Recall, also known as the true positive rate, sensitivity or probability of detection is such an evaluation metric. TPR: What fraction of all positive instances does the classifier correctly identify as positive?

$$Recall = \frac{TP}{TP + FN} \tag{3.3}$$

You can see from this formula that there are two ways to get a larger recall number. First, by either increasing the number of true positives or by reducing the number of false negatives. Since this will make the denominator smaller.

Precision or Positive Predictive Value (PPV)

Now suppose that we have a machine learning task, where it's really important to avoid or limit the number of false positives. In other words, we're fine with cases where not all true positive instances are detected but when the classifier does predict the positive class, we want to be very confident that it's correct. How many of the samples predicted as positive are actually positive or What fraction of positive predictions are correct?

$$Precision = \frac{TP}{TP + FP} \tag{3.4}$$

So to increase precision, we must either increase the number of true positives the classifier predicts or reduce the number of errors where the classifier incorrectly predicts that a negative instance is in the positive class.

False Positive Rate (FPR)

Another related evaluation metric that will be useful is called the false positive rate, also known as specificity. This gives the fraction of all negative instances that the classifier incorrectly identifies as positive. What fraction of all negative instances does the classifier incorrectly identify as positive?

$$FPR = \frac{FP}{TN + FP} \tag{3.5}$$

The Precision-Recall Tradeoff

Going back to our classifier visualization (Fig.3.3), let's look at how precision and recall can be interpreted. We can see that a precision of 0.68 means that about 68 percent of the points in the positive prediction region to the left of the decision boundary or 13 out of the 19 instances are correctly labeled as positive. A recall of 0.87 means, that of all true positive instances, so all black points in the figure, the positive prediction region has found about 87 percent of them' or 13 out of 15.

$$Precision = \frac{TP}{TP + FP} = \frac{13}{13 + 6} = \frac{13}{19} = 0.68 \tag{3.6}$$

$$Recall = \frac{TP}{TP + FN} = \frac{13}{13 + 2} = \frac{13}{15} = 0.87 \tag{3.7}$$

High Precision, Lower Recall

If we wanted a classifier that was oriented towards higher levels of precision, we might change the decision boundary, this comes at a cost, some points are incorrectly predicted as being negative. On the other hand, if we want to minimize false negatives and no true negatives either and obtain High recall, also we need to change the decision boundary, this also comes with a cost since result in many number of false positives. This illustrate a classic trade-off that often appears in machine learning applications. Namely, that you can often increase the precision of a classifier but the downside is that you may reduce recall, or you could increase the recall of a classifier at the cost of reducing precision.

- Recall-oriented machine learning tasks:
  - Search and information extraction in legal discovery
  - Tumor detection
  - Often paired with a human expert to filter out false positive
- Precision-oriented machine learning tasks:
  - Search engine ranking, query suggestion
  - Document classification
  - Many customer-facing tasks (user remember failures!)

F1-score

When evaluating classifiers, it's often convenient to compute a quantity known as an F1 score, that combines precision and recall into a single number. Mathematically, this is based on the harmonic mean of precision and recall using this formula.

$$F_1 = 2 * \frac{Precision \cdot Recall}{Precision + Recall} = \frac{2 \cdot TP}{2 \cdot TP + FN + FP} \tag{3.8}$$

This F1 score is a special case of a more general evaluation metric known as an F score that introduces a parameter beta. By adjusting beta we can control how much emphasis an evaluation is given to precision versus recall.

$$f_\beta = (1 + \beta^2) \cdot \frac{Precision \cdot Recall}{(\beta^2 \cdot Precision) + Recall} = \frac{(1 + \beta^2) \cdot TP}{(1 + \beta^2) \cdot TP + \beta \cdot FN + FP} \tag{3.9}$$

- Precision-oriented users: $\beta = 0.5$ (false positives hurt performance more than false negatives)
- Recall-oriented users: $\beta = 2$ (false negatives hurt performance more than false positives)
- Setting $\beta = 1$ corresponds to the F_1 score special case that weights precision and recall equally

```
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score

print('Accuracy: {:.2f}'.format(accuracy_score(y_test, tree_predicted)))
print('Precision: {:.2f}'.format(precision_score(y_test, tree_predicted)))
print('Recall: {:.2f}'.format(recall_score(y_test, tree_predicted)))
print('F1: {:.2f}'.format(f1_score(y_test, tree_predicted)))
```

If we want a more comprehensive summary of precision, recall and F1-score, sklearn metrics provides a handy classification report function. Like the previous core functions, classification report takes the true or positive class and predicted labels as the first two required arguments. It also takes some optional arguments that control the format of the output. Here, we use the target names option to label the classes in the output table. The last column support, shows the number of instances in the test set that have that true label.

```
# Combined report with all above metrics
from sklearn.metrics import classification_report

print(classification_report(y_test, tree_predicted, target_names=['not 1', '1']))
```

Here we show classification reports for four different classifiers on the binary digit classification problem.

```
print('Random class-proportional (dummy)\n',
      classification_report(y_test, y_classprop_predicted,
      target_names=['not 1', '1']))
print('SVM\n',
      classification_report(y_test, svm_predicted,
```

```
      target_names = ['not 1', '1']))
print('Logistic regression\n',
      classification_report(y_test, lr_predicted,
      target_names = ['not 1', '1']))
print('Decision tree\n',
      classification_report(y_test, tree_predicted,
      target_names = ['not 1', '1']))
```

3.2.2 Classifier Decision Functions & Predicting Probabilities

The confusion matrix and the classification report provide a very detailed analysis of a particular set of predictions. However, the predictions themselves already threw away a lot of information that is contained in the model. Many classifiers in scikit learn can provide information about the uncertainty associated with a particular prediction either by using the `decision_function` method or the `predict_proba` method. When given a set of test points, the `decision_function` method provides for each classifier score value per test point that indicates how confidently the classifier predicts the positive class so there will be (large-magnitude positive values) for those points or it predicts a negative class so there will be (large-magnitude negative values) for negative points. Choosing a fixed decision threshold gives a classification rule. By sweeping the decision threshold through the entire range of possible score values, we get a series of classification outcomes that form a curve.

Here's an example in the notebook showing the first few instances from our classification problem using a logistic regression classifier. We can see the instances in the negative class often have large magnitude negative scores. And indeed the instances in the positive class has positive scores from the logistic regression classifier.

```
X_train, X_test, y_train, y_test = train_test_split(X, y_binary_imbalanced,
random_state=0)
y_scores_lr = lr.fit(X_train, y_train).decision_function(X_test)
y_score_list = list(zip(y_test[0:20], y_scores_lr[0:20]))

# show the decision_function scores for first 20 instances
y_score_list
```

Likewise, the `predict_proba` function provides predicted probabilities of class membership. (predict_proba)

- Typical rule: a classifier choose the most likely class
  - e.g. class I if threshold > 0.50.
- Adjusting this decision threshold affects predictions of the classifier.
- Higher threshold results in a more conservative classifier or more confident in predicting the class.
  - e.g. only predict Class I if estimated probability of Class I is above 70%
  - This increases precision. Doesn't predict Class I as often, but when it does, it gets high proportion of Class I instances correct.

- Not all models provide realistic probability estimates.

```
X_train, X_test, y_train, y_test = train_test_split(X, y_binary_imbalanced,
random_state=0)
y_proba_lr = lr.fit(X_train, y_train).predict_proba(X_test)
y_proba_list = list(zip(y_test[0:20], y_proba_lr[0:20,1]))

# show the probability of positive class for first 20 instances
y_proba_list
```

Note that not all models provide useful probability estimates of this type. For example, a model that was over-fit to a trending set. Might provide overly optimistic high probabilities that were in fact not accurate.

Now, we can use these decision scores or prediction probabilities for getting more complete evaluation picture of a classifiers performance. For a particular application, we might pick a specific decision threshold depending on whether we want the classifier to be more or less conservative about making false-positive or false-negative errors. It might not be entirely clear when developing a new model, what the right decision threshold would be, and how that choice will affect evaluation metrics like precision and recall. So instead, what we'll do is, look at how classifier performs for all possible decision thresholds.

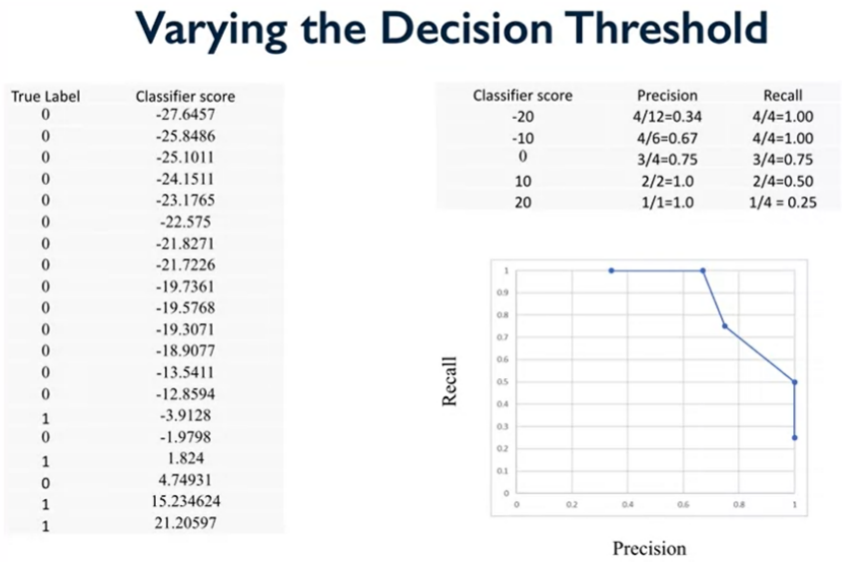


Figure 3.4: Varying the Decision Threshold

We plot a series of points through the space which we can connected at as a curve. This way we can get a more complete picture by varying the threshold of how the precision and recall of the result and classifier output changes as a function of the decision threshold. And this resulting chart here is called a precision recall curve.

3.2.3 Precision-Recall and ROC Curves

Precision-Recall Curves

Precision-Recall Curves are very widely used evaluation method from machine learning.

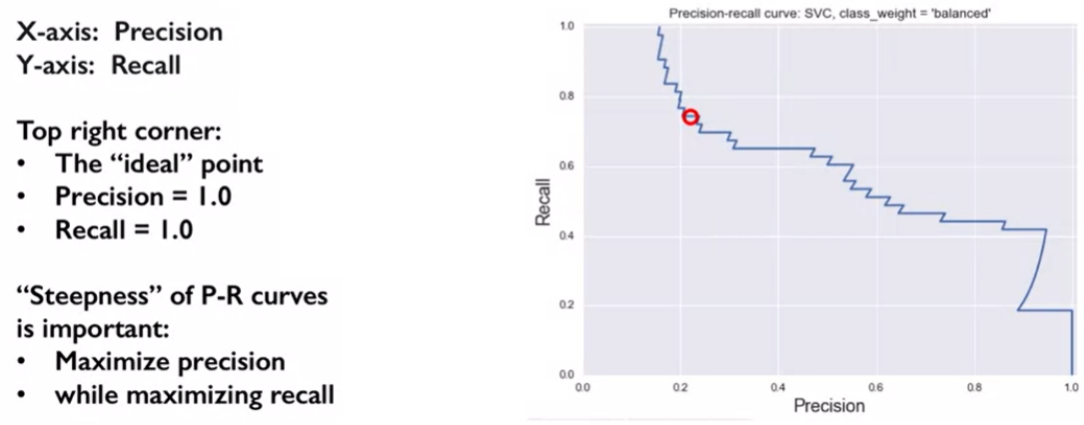


Figure 3.5: Precision-Recall Curve

Now an ideal classifier would be able to achieve perfect precision of 1.0 and perfect recall of 1.0. And in general, with precision recall curves, the closer in some sense, the curve is to the top right corner, the more preferable it is, the more beneficial the tradeoff it gives between precision and recall. The red circle indicates the precision and recall that's achieved when the decision threshold is zero. So you can see that in this particular application there is a general downward trend. So as the precision of the classifier goes up, the recall tends to go down.

ROC Curves and AUC

ROC curves or receiver operating characteristic curves are a very widely used visualization method that illustrate the performance of a binary classifier.

X-axis: False Positive Rate
Y-axis: True Positive Rate

Top left corner:
- The "ideal" point
- False positive rate of zero
- True positive rate of one

"Steepness" of ROC curves is important:
- Maximize the true positive rate
- while minimizing the false positive rate

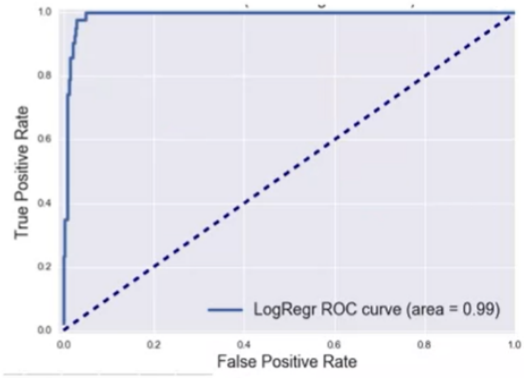


Figure 3.6: ROC Curve

The ideal point in ROC space is one where the classifier achieves zero, a false positive rate of zero, and a true positive rate of one. So that would be the upper left corner. So curves in ROC space represent different tradeoffs as the decision boundary, the decision threshold is varied for the classifier. So just as in the precision recall case, as we vary decision threshold, we'll get different numbers of false positives and true positives that we can plot on a chart. The dotted line here that I'm showing is the classifier curve that secretly results from a classifier that randomly guesses the label for a binary class. So the dotted line here is used as a base line. Reasonably good classifier will give an ROC curve that is consistently better than random across all decision threshold choices. And then an excellent classifier would be one like I've shown here, which is way up into the left. We can qualify the goodness of a classifier in some sense by looking at how much area there is underneath the curve. So the area underneath the random classifier is going to be 0.5. We use something called area under the curve, AUC. That's the single number that measures this total area underneath the ROC curve as a way to summarize a classifier's performance. So, an AUC of zero represents a very bad classifier, and an AUC of one will represent an optimal classifier.

3.3 Metrics for Multiclass Classification

- Multi-class evaluation is an extension of the binary case
  - The result for multi-class evaluation amount to a collection of true vs predicted binary outcomes, one per class
  - You can generate confusion matrices that are specially useful
  - You can generate classification report
- Overall evaluation metrics are averages across classes
  - But there are different ways to average multi-class results
  - The support (number of instances) for each class is important to consider, e.g. in case of imbalanced classes
- Multi-label classification: each instance can have multiple labels

3.3.1 Multi-class Confusion Matrix

The correct prediction is by the classifier where the true class matches the predicted class are all along the diagonal and misclassifications are off the diagonal. As an aside, it's sometimes useful to display a confusion matrix as a heat map in order to highlight the relative frequencies of different types of errors. So as a general rule of thumb as part of model evaluation, I suggest always looking at the confusion matrix for your classifier. To get some insight into what kind of errors it is making for each class including whether some classes are much more prone to certain kinds of errors than others.

```
dataset = load_digits()
X, y = dataset.data, dataset.target
X_train_mc, X_test_mc, y_train_mc, y_test_mc = train_test_split(X, y,
                                                                random_state=0)


svm = SVC(kernel = 'linear').fit(X_train_mc, y_train_mc)
svm_predicted_mc = svm.predict(X_test_mc)
confusion_mc = confusion_matrix(y_test_mc, svm_predicted_mc)
df_cm = pd.DataFrame(confusion_mc,
           index = [i for i in range(0,10)], columns = [i for i in range(0,10)])

plt.figure(figsize=(5.5,4))
sns.heatmap(df_cm, annot=True)
plt.title('SVM Linear Kernel \nAccuracy:{0:.3f}'.format(accuracy_score(y_test_mc,
                                                        svm_predicted_mc)))
plt.ylabel('True label')
plt.xlabel('Predicted label')


svm = SVC(kernel = 'rbf').fit(X_train_mc, y_train_mc)
svm_predicted_mc = svm.predict(X_test_mc)
confusion_mc = confusion_matrix(y_test_mc, svm_predicted_mc)
df_cm = pd.DataFrame(confusion_mc, index = [i for i in range(0,10)],
                                           columns = [i for i in range(0,10)])

plt.figure(figsize = (5.5,4))
sns.heatmap(df_cm, annot=True)
plt.title('SVM RBF Kernel \nAccuracy:{0:.3f}'.format(accuracy_score(y_test_mc,
                                                      svm_predicted_mc)))
plt.ylabel('True label')
plt.xlabel('Predicted label');
```

3.3.2 Multi-class classification report

You can get a classification report that summarizes multiple evaluation metrics for a multi-class classifier with an average metric computed for each class.

```
print(classification_report(y_test_mc, svm_predicted_mc))
```

The most commonly used metric for imbalanced datasets in the multiclass setting is the multiclass version of the $f-score$. The idea behind the multiclass $f-score$ is to compute one binary $f-score$ per class,with that class being the positive class and the other classes making up the negative classes. Then, these per-class $f-scores$ are averaged using one of the following strategies:

- `macro`: averaging computes the unweighted per-class $f-scores$. This gives equal weight to all classes, no matter what size is.
- `micro`: averaging computes the total number of false positives, false negatives, and true positives over all classes, and then computes precision, recall, and $f-score$ using these counts.
- `weighted`: averaging computes the mean of the per-class $f-scores$, weighted by their support. This i what is reported in the classification report.

If you care about each sample equally much, it is recommended to use the `micro` average $f-score$; if you care about each class equally much, it is recommended to use the `macro` average $f-score$. If the micro-average is much lower than the macro-average, then examine the larger classes for poor metric performance. If the macro-average is much lower than the micro-average, then you should examine the smaller classes to see why they have poor metric performance.

```
# we used the precision metric and specify whether we want micro-average
# precision which is the first case or macro-average precision in the
# second case.

print('Micro-averaged precision = {:.2f} (treat instances equally)'
      .format(precision_score(y_test_mc, svm_predicted_mc, average = 'micro')))
print('Macro-averaged precision = {:.2f} (treat classes equally)'
      .format(precision_score(y_test_mc, svm_predicted_mc, average = 'macro')))

# In the second example, we use the f1 metric and compute micro and
# macro-averaged f1

print('Micro-averaged f1 = {:.2f} (treat instances equally)'
      .format(f1_score(y_test_mc, svm_predicted_mc, average = 'micro')))
print('Macro-averaged f1 = {:.2f} (treat classes equally)'
      .format(f1_score(y_test_mc, svm_predicted_mc, average = 'macro')))
```

3.4 Regression Metrics

In theory, we could apply the same type of error analysis and more detailed evaluation to regression that we applied for classification.

- Typically R^2 score is enough. R^2 computes how well future instances will be predicted
  - Best possible is 1.0

  - Constant prediction score is 0.0
  - R^2 does have the potential to go negative for bad model fits, such as when fitting non-linear functions to data.

- Alternative metrics include:
  - mean_absolute_error (absolute difference of target & predicted values) (L1 norm laws)
  - mean_squared_error (squared difference of target & predicted values) (L2 norm laws)
  - median_absolute_error (robust to ignoring outliers)

There's a dummy regressor class that provides predictions using simple strategies that do not look at the input data. Although, you should not use the dummy regressor for actual problems. Its only use is to provide a baseline for comparison. The dummy regressor implements four simple baseline rules for regression, using the `strategy` parameter:

- mean, predicts the mean of the training target values
- median, predicts the median of the training target values
- quantile, predicts a user-provided quantile of the training target values (e.g. value at the 75^{th} percentile)
- constant, predicts a custom constant value provided by the user

Make sure the evaluation metric you choose for a regression problem does penalize errors in a way that reflects the consequences of those errors for the business, organizational, or user needs of your application.

```
%matplotlib notebook
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn import datasets
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.dummy import DummyRegressor

diabetes = datasets.load_diabetes()

X = diabetes.data[:, None, 6]
y = diabetes.target

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

lm = LinearRegression().fit(X_train, y_train)
lm_dummy_mean = DummyRegressor(strategy = 'mean').fit(X_train, y_train)

y_predict = lm.predict(X_test)
```

```
y_predict_dummy_mean = lm_dummy_mean.predict(X_test)

print('Linear model, coefficients: ', lm.coef_)
print("Mean squared error (dummy): {:.2f}".format(mean_squared_error(y_test,
                                                    y_predict_dummy_mean)))
print("Mean squared error (linear model): {:.2f}".format(mean_squared_error
                                                      (y_test, y_predict)))
print("r2_score (dummy): {:.2f}".format(r2_score(y_test, y_predict_dummy_mean)))
print("r2_score (linear model): {:.2f}".format(r2_score(y_test, y_predict)))

# Plot outputs
plt.scatter(X_test, y_test,  color='black')
plt.plot(X_test, y_predict, color='green', linewidth=2)
plt.plot(X_test, y_predict_dummy_mean, color='red', linestyle = 'dashed',
                                                 linewidth=2, label = 'dummy')

plt.show()
```

3.5 Model Selection: Optimizing Classifiers for Different Evaluation Metrics

How you can apply metrics as criteria for selecting the best classifier for your application, otherwise known as model selection. In previous chapters we've seen a number of different evaluation frameworks for potential model selection. First, we simply did training and testing on the same dataset, which as we well know, typically overfits badly and doesn't generalize well to new data. As a side note however, it can serve as a useful sanity check to make sure your software engineering and feature generation pipeline is working correctly. Second, we frequently use the single train-test split to produce a single evaluation metric. While fast and easy, this doesn't give as realistic a set of estimates for how well the model may work on future, new data. And we don't get a good picture for the variance in the evaluation metrics that may result as we do prediction on different test sets. Third, we used k-fold cross-validation to create K random train-test splits, where the evaluation metric was averaged across splits. This leads to models that are more reliable on unseen data. In particular, we can also use grid search using for example the GridSearchCV method within each cross-validation fold, to find optimal parameters for a model with respect to the evaluation metric. The default evaluation metric used for a cross-val score or GridSearchCV is accuracy. So how do you apply the new metrics you've learned about here like AUC in model selection? Scikit-learn makes this very easy. You simply add a scoring parameter that's set to the string with the name of the evaluation metric you want to use.

Let's first look at an example using the scoring parameter for cross-validation

```
from sklearn.model_selection import cross_val_score
from sklearn.svm import SVC

dataset = load_digits()
# again, making this a binary problem with 'digit 1' as positive class
# and 'not 1' as negative class
```

```
X, y = dataset.data, dataset.target == 1

# cross-validation example where we're running five folds using a
# support vector classifier with  a linear kernel and C parameter
# set to one.
clf = SVC(kernel='linear', C=1)

# First call to cross-val score use accuracy as the default scoring metric
print('Cross-validation (accuracy)', cross_val_score(clf, X, y, cv=5))
# Second call use AUC as scoring metric
print('Cross-validation (AUC)', cross_val_score(clf, X, y, cv=5,
                                                scoring = 'roc_auc'))
# Third call use recall as scoring metric
print('Cross-validation (recall)', cross_val_score(clf, X, y, cv=5,
                                                 scoring = 'recall'))
```

You can see the resulting list of five evaluation values, one per fold for each metric. Now, here we're not doing any parameter tuning we're simply evaluating our model's average performance across multiple folds.

Now, in this grid search example we use a support vector classifier that uses a radio basis function kernel. And the critical parameter here is the gamma parameter that intuitively sets the radius or width of influence of the kernel. We use GridSearchCV to find the value of gamma that optimizes a given evaluation metric in two cases. In the first case, we just optimize for average accuracy; in the second case we optimize for AUC.

```
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import roc_auc_score

dataset = load_digits()
X, y = dataset.data, dataset.target == 1
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

clf = SVC(kernel='rbf')
grid_values = {'gamma': [0.001, 0.01, 0.05, 0.1, 1, 10, 100]}

# default metric to optimize over grid parameters: accuracy
grid_clf_acc = GridSearchCV(clf, param_grid = grid_values)
grid_clf_acc.fit(X_train, y_train)
y_decision_fn_scores_acc = grid_clf_acc.decision_function(X_test)

print('Grid best parameter (max. accuracy): ', grid_clf_acc.best_params_)
print('Grid best score (accuracy): ', grid_clf_acc.best_score_)

# alternative metric to optimize over grid parameters: AUC
grid_clf_auc = GridSearchCV(clf, param_grid = grid_values, scoring = 'roc_auc')
grid_clf_auc.fit(X_train, y_train)
```

```
y_decision_fn_scores_auc = grid_clf_auc.decision_function(X_test)

print('Test set AUC: ', roc_auc_score(y_test, y_decision_fn_scores_auc))
print('Grid best parameter (max. AUC): ', grid_clf_auc.best_params_)
print('Grid best score (AUC): ', grid_clf_auc.best_score_)
```

The optimal parameter value can be quite different depending on the evaluation metric used to optimize. You can see the complete list of names for the evaluation metric supported by the scoring parameter by running the following code that uses the score's variable imported from sklearn metrics.

```
from sklearn.metrics.scorer import SCORERS

print(sorted(list(SCORERS.keys())))
```

Let's take a look at a specific example that shows how a classifier's decision boundary changes when it's optimized for different evaluation metrics. This classification problem is based on the same binary digit classifier training and test sets we've been using as an example throughout the notebook. In these classification visualization examples, the positive examples, the digit one are shown as black points and the region of positive class prediction is shown in the light-colored or yellow region to the right of this decision boundary. The negative examples, all other digits, are shown as white points. And the region of negative class prediction here in these figures is to the left of the decision boundary. The data points have been plotted using two out of the 64 future values in the digits' dataset and have been jittered a little. That is, I've added a little bit of random noise so we can see more easily the density of examples in the feature space. We apply grid search here to explore different values of the optional class weight parameter that controls how much weight is given to each of the two classes during training. As it turns out, optimizing for different evaluation metrics results in different optimal values of the class weight parameter. As the class weight parameter increases, more emphasis will be given to correctly classifying the positive class instances.

```
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from adspy_shared_utilities import plot_class_regions_for_classifier_subplot
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV


dataset = load_digits()
X, y = dataset.data, dataset.target == 1
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

# Create a two-feature input vector matching the example plot above
# We jitter the points (add a small amount of random noise) in case there
are areas in feature space where many instances have the same features.
jitter_delta = 0.25
X_twovar_train = X_train[:,[20,59]]+ np.random.rand(X_train.shape[0], 2) -
```

```
                                                                    jitter_delta
X_twovar_test  = X_test[:,[20,59]] + np.random.rand(X_test.shape[0], 2) -
                                                                    jitter_delta

clf = SVC(kernel = 'linear').fit(X_twovar_train, y_train)
grid_values = {'class_weight':['balanced', {1:2},{1:3},{1:4},{1:5},{1:10},{1:20},
                                                                        {1:50}]}
plt.figure(figsize=(9,6))
for i, eval_metric in enumerate(('precision','recall', 'f1','roc_auc')):
    grid_clf_custom = GridSearchCV(clf, param_grid=grid_values,
                                                            scoring=eval_metric)
    grid_clf_custom.fit(X_twovar_train, y_train)
    print('Grid best parameter (max. {0}): {1}'
          .format(eval_metric, grid_clf_custom.best_params_))
    print('Grid best score ({0}): {1}'
          .format(eval_metric, grid_clf_custom.best_score_))
    plt.subplots_adjust(wspace=0.3, hspace=0.3)
    plot_class_regions_for_classifier_subplot(grid_clf_custom, X_twovar_test,
                                y_test, None, None, None, plt.subplot(2, 2, i+1))

    plt.title(eval_metric+'-oriented SVC')
plt.tight_layout()
plt.show()
```

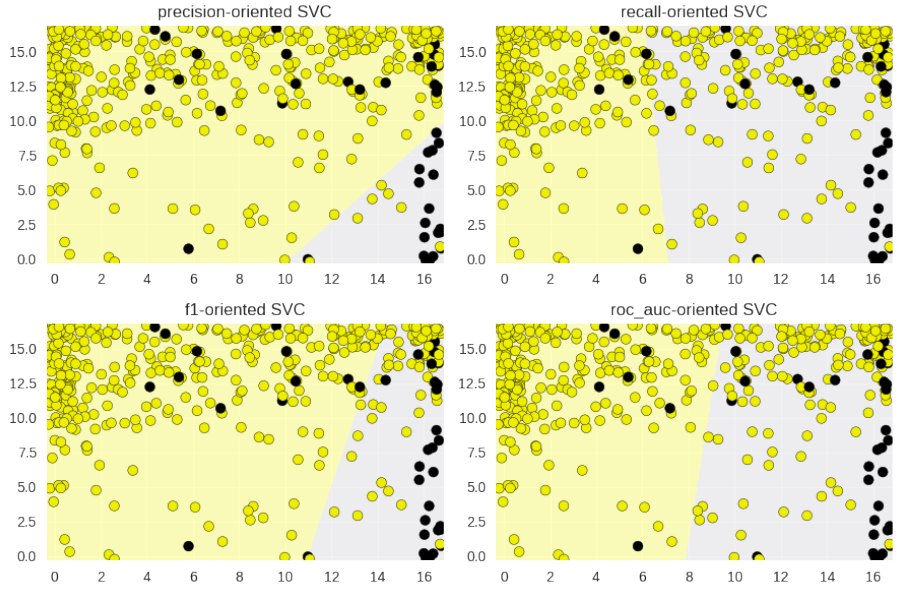


Figure 3.7: Optimizing a classifier using different evaluation metrics

The precision-oriented classifier we see here with class weight of two, tries hard to reduce false negatives while increasing true positives. So it focuses on the cluster of positive class points in the lower right corner where there are relatively few negative class points. Here, precision is over 50 percent. In contrast, the recall-oriented classifier with class weight of 50, tries hard to reduce the number of false negatives while increasing true positives. That is, it tries to find most of the positive class points as part of its positive class predictions. We can also see that the decision boundary for the F1-oriented classifier has an optimal class weight of two, which is between the optimal class weight values for the precision and recall-oriented classifiers. Visually we can see that the F1-oriented classifier also has a kind of intermediate positioning between the precision and recall-oriented, decision boundaries. This makes sense given that F1 is the harmonic mean of precision and recall. The AUC-oriented classifier with optimal class weight to 5 has a similar decision boundary to the F1-oriented classifier, but shifted slightly in favor of higher recall.

Training, Validation, ad Test Framework for Model Selection and Evaluation

- Using only cross-validation or a test set to do model selection may lead to more subtle overfitting/optimistic generalization estimates
- Instead, use three data splits:
    1. Training set (model building)
    2. Validation set (model selection)
    3. Test set (final evaluation)
- In practice
    - Create an initial training/test split.
    - Do cross-validation on the training data for model/parameter selection
    - Save the held-out test set for final model evaluation

Concluding notes

- Accuracy is often not the right evaluation metric for many real-word machine learning tasks
    - false positives and false negatives might have very different real-world effects for users or for organization outcomes. So it's important to select an evaluation metric that reflects those user application or business needs
- Examples of additional evaluation methods include:
    - Learning curve, may be useful as part of a cost-benefit analysis: How much does accuracy (or other metric) change as a function of the amount of the training data?
    - Sensitivity analysis: How much does accuracy (or other metric) change as a function of key learning parameter values?

Chapter 4

Supervised Machine Learning - Part II

This chapter covers more advanced supervised learning methods that include ensembles of trees (random forests, gradient boosted trees), and neural networks (with an optional summary on deep learning). You will also learn about the critical problem of data leakage in machine learning and how to detect and avoid it.

- Understand how specific supervised learning algorithms - in particular, those based on decision trees and neural networks - estimate their own parameters from data to make new predictions.
- Apply the right algorithm for a given task by understanding the strengths and weaknesses of additional supervised learning methods.
- Apply additional types of supervised machine learning algorithms in Python with scikit-learn.
- Recognize and avoid instances of data leakage

4.1 Naive Bayes Classifiers

Another family of supervised learning models that's related to linear classification models is the Naive Bayes family of classifiers, which are based on simple probabilistic models of how the data in each class might have been generated. Naive Bayes classifiers are called naive because informally, they make the simplifying assumption that each feature of an instance is independent of all the others, given the class. In practice, of course, this is not often the case, features often are somewhat correlated. This naive simplifying assumption means on the one hand, that learning a Naive Bayes classifier is very fast. Because only simple per class statistics need to be estimated for each feature and applied for each feature independently. On the other hand, the penalty for this efficiency is that the generalization performance of Naive Bayes Classifiers can often be a bit worse than other more sophisticated methods, or even linear models for classification. Even so, especially for high dimensional data sets, Naive Bayes Classifiers can achieve performance that's often competitive to other more sophisticated methods, like support vector machines, for some tasks.

4.1.1 Naive Bayes Classifier Types

- Bernoulli: binary features (e.g. word presence/absence). The Bernoulli Naive Bayes model is quit handy because we could represent the presence or the absence of a given word in the text with the binary feature. Of course this doesn't take into account how often the word occurs in the text.
- Multinomial: discrete features (e.g. word counts). This model uses a set of count base features each of which does account for how many times a particular feature such as a word is observed in training example like a document.
- Gaussian: continuous/real-values features
  - Statistics computed for each class:
    - For each feature: mean, standard deviation

This chapter will focus on Gaussian Naive Bayes classifiers. The Gaussian Naive Bayes Classifier assumes that the data for each class was generated by a simple class specific Gaussian distribution. Predicting the class of a new data point corresponds mathematically to estimating the probability that each classes Gaussian distribution was most likely to have generated the data point. Classifier then picks the class that has the highest probability. Without going into the mathematics involved, it can be shown that the decision boundary between classes in the two class Gaussian Naive Bayes Classifier. In general is a parabolic curve between the classes. And in the special case where the variance of these feature is the same for both classes. The decision boundary will be linear.

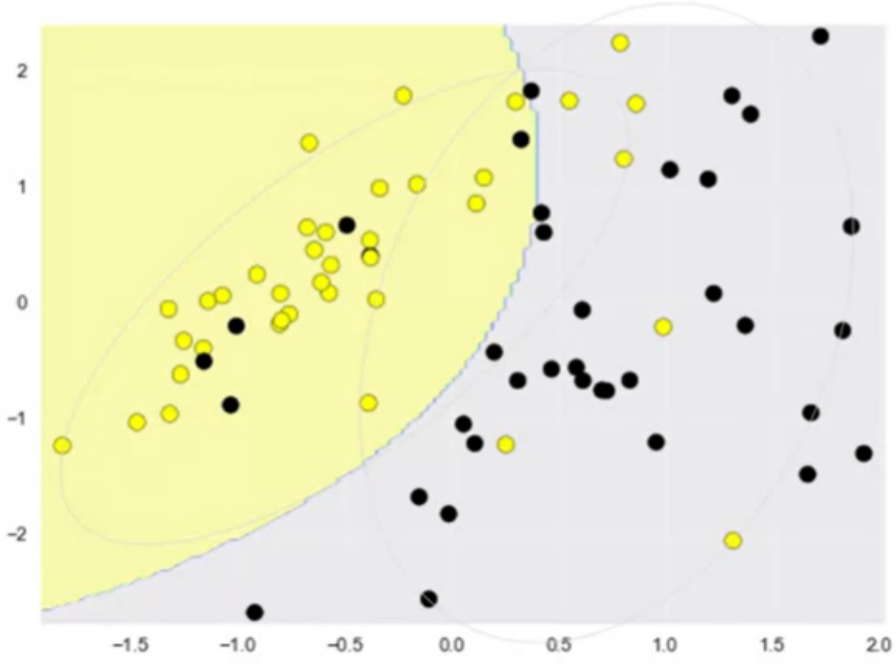


Figure 4.1: Gaussian Naive Bayes Classifier

You can see the centers of the Gaussian's correspond to the mean value of each feature for each class. More specifically, the gray ellipses show the contour line of the Gaussian distribution for each class, that corresponds to about two standard deviations from the mean. The line between the yellow and gray background areas represents the decision boundary. And we can see that this is indeed parabolic.

```
from sklearn.naive_bayes import GaussianNB
from adspy_shared_utilities import plot_class_regions_for_classifier

X_train, X_test, y_train, y_test = train_test_split(X_C2, y_C2, random_state=0)

# GaussianNB supports partial fit to train the classifier incrementally in case
# you're working with a huge data set that doesn't fit into memory
nbclf = GaussianNB().fit(X_train, y_train)
plot_class_regions_for_classifier(nbclf, X_train, y_train, X_test, y_test,
                                  'Gaussian Naive Bayes classifier: Dataset 1')
```

Other, on a real world example

```
# breast cancer data set
X_train, X_test, y_train, y_test = train_test_split(X_cancer, y_cancer, random_state = 0)

nbclf = GaussianNB().fit(X_train, y_train)
print('Breast cancer dataset')
print('Accuracy of GaussianNB classifier on training set: {:.2f}'
     .format(nbclf.score(X_train, y_train)))
print('Accuracy of GaussianNB classifier on test set: {:.2f}'
     .format(nbclf.score(X_test, y_test)))
```

Typically, Gaussian Naive Bayes is used for high-dimensional data. When each data instance has hundreds, thousands or maybe even more features. And likewise the Bernoulli and Multinomial flavors of Naive Bayes are used for text classification where there are very large number of distinct words is features and where the future vectors are sparse because any given document uses only a small fraction of the overall vocabulary.

Strengths, weaknesses and parameters

Pros	Cons
Easy to understand	Assumption that features are conditionally independent
Simple, efficient parameter estimation	As a result, other classifier types often have better generalization performance
Works well with high dimensional data	Their confidence estimates for predictions are not very accurate
baseline comparison against more sophisticated methods	

Table 4.1: Naive Bayes classifiers: Pros and Cons

4.2 Ensembles of Decision Trees

A widely used and effective method in machine learning involves creating learning models known as ensembles. An ensemble takes multiple individual learning models and combines them to produce an aggregate model that is more powerful than any of its individual learning models alone. Why are ensembles effective? Well, one reason is that if we have different learning models, although each of them might perform well individually, they'll tend to make different kinds of mistakes on the data set. And typically, this happens because each individual model might overfit to a different part of the data. By combining different individual models into an ensemble, we can average out their individual mistakes to reduce the risk of overfitting while maintaining strong prediction performance.There are many models in machine learning that have been proven to be effective on a wide range of datasets for classification and regression, both of which use decision trees as their building blocks:

- Random forests
- Gradient boosted decision trees

4.2.1 Random Forests

Random forests are an example of the ensemble idea applied to decision trees. Random forests are widely used in practice and achieve very good results on a wide variety of problems.

sklearn.ensemble module

- Classification: `RandomForestClassifer`
- Regression: `RandomForestRegressor`

As we saw earlier, one disadvantage of using a single decision tree was that decision trees tend to be prone to overfitting the training data. As its name would suggest, a random forest creates lots of individual decision trees on a training set, often on the order of tens or hundreds of trees. The idea is that each of the individual trees in a random forest should do reasonably well at predicting the target values in the training set but should also be constructed to be different in some way from the other trees in the forest. Again, as the name would suggest this difference is accomplished by introducing random variation into the process of building each decision tree. This random variation during tree building happens in two ways. First, the data used to build each tree is selected randomly and second, the features chosen in each split tests are also randomly selected.

Building Random Forests

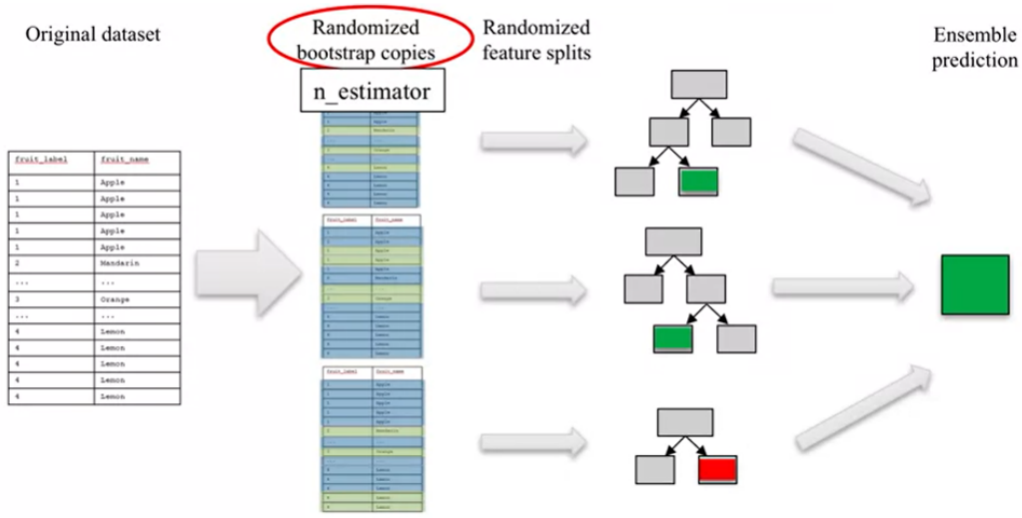


Figure 4.2: Random Forest Process

To create a random forest model you first decide on how many trees to build. This is set using the `n_estimators` parameter for both `RandomForestClassifier` and `RandomForestRegressor`. Each tree were built from a different random sample of the data called the *bootstrap sample*. Bootstrap samples are commonly used in statistics and machine learning. If your training set has N instances or `n_samples` in total, a bootstrap sample of size N is created by just repeatedly picking one of the N dataset rows at random with replacement, that is, allowing for the possibility of picking the same row again at each selection. You repeat this random selection process N times. The resulting bootstrap sample has N rows just like the original training set but with possibly some rows from the original dataset missing and others occurring multiple times just due to the nature of the random selection with replacement. When building a decision tree for a random forest, the process is almost the same as for a standard decision tree but with one important difference. When picking the best split for a node, instead of finding the best split across all possible features, a random subset of features is chosen and the best split is found within that smaller subset of features. The number of features in the subset that are randomly considered at each stage is controlled by the `max_features` parameter. This randomness in selecting the bootstrap sample to train an individual tree in a forest ensemble, combined with the fact that splitting a node in the tree is restricted to random subsets of the features of the split, virtually guarantees that all of the decision trees and the random forest will be different. The random forest model is quite sensitive to the `max_features` parameter.

- Setting $max\_features = 1$ leads to forest with diverse from each other and possibly with many levels in order to produce a good fit to the data, more complex trees.
- Setting $max\_features = high < closetonumberof features >$ will lead to similar forests with simpler trees. They will be able to fit the data easily.

- $max\_features = low$ means that the trees will be quite different, and that each tree might need to be very deep in order to fit the data well.

Once a random forest model is trained, it predicts the target value for new instances by:

1. Make a prediction for every tree in the random forest.
2. Combine individual predictions
    - Regression: the overall prediction is then typically mean of individual tree predictions
    - Classification: the overall prediction is based on a weighted vote.
        - Each tree gives probability for each class
        - Probabilities averaged across all the trees
        - Predict the class with highest probability

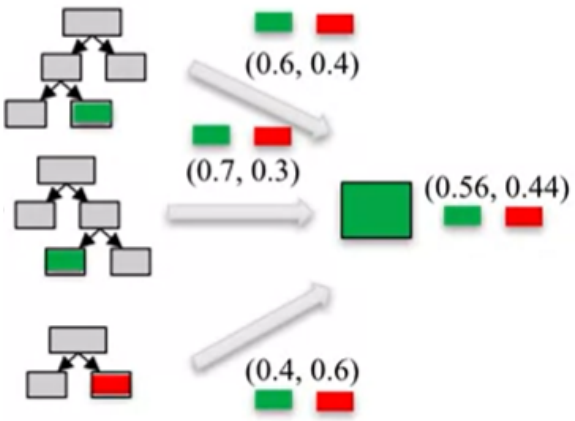


Figure 4.3: Random Forest Prediction

Let's take a look at the code that created and visualized this random forest on the fruit dataset. This code also plots the decision boundaries for the other five possible feature pairs

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

X_train, X_test, y_train, y_test = train_test_split(X_fruits.as_matrix(),
                                                    y_fruits.as_matrix(),
                                                    random_state = 0)
fig, subaxes = plt.subplots(6, 1, figsize=(6, 32))

title = 'Random Forest, fruits dataset, default settings'
pair_list = [[0,1], [0,2], [0,3], [1,2], [1,3], [2,3]]
```

```
for pair, axis in zip(pair_list, subaxes):
    X = X_train[:, pair]
    y = y_train

    clf = RandomForestClassifier().fit(X, y)
    plot_class_regions_for_classifier_subplot(clf, X, y, None,
                                              None, title, axis,
                                              target_names_fruits)

    axis.set_xlabel(feature_names_fruits[pair[0]])
    axis.set_ylabel(feature_names_fruits[pair[1]])

plt.tight_layout()
plt.show()

clf = RandomForestClassifier(n_estimators = 10,
                             random_state=0).fit(X_train, y_train)

print('Random Forest, Fruit dataset, default settings')
print('Accuracy of RF classifier on training set: {:.2f}'
     .format(clf.score(X_train, y_train)))
print('Accuracy of RF classifier on test set: {:.2f}'
     .format(clf.score(X_test, y_test)))
```

For comparison with other supervised learning methods, we use the breast cancer dataset again. Since there are about 30 features, we'll set max_features to eight to give a diverse set of trees that also fit the data reasonably well.

```
from sklearn.ensemble import RandomForestClassifier

X_train, X_test, y_train, y_test = train_test_split(X_cancer, y_cancer,
random_state = 0)

clf = RandomForestClassifier(max_features = 8, random_state = 0)
clf.fit(X_train, y_train)

print('Breast cancer dataset')
print('Accuracy of RF classifier on training set: {:.2f}'
     .format(clf.score(X_train, y_train)))
print('Accuracy of RF classifier on test set: {:.2f}'
     .format(clf.score(X_test, y_test)))
```

Notice that we did not have to perform scaling or other pre-processing as we did with a number of other supervised learning methods. Also note that we passed in a fixed value for the random state parameter in order to make the results reproducible. If we didn't set the random state parameter, the model would likely be different each time due to the randomized nature of the random forest algorithm.

Strengths, weaknesses and parameters

Pros	Cons
Widely used, excellent prediction performance on many problems	The resulting models are often difficult for humans to interpret
Doesn't require careful normalization of features or extensive parameter tuning	Like decision trees, random forests may not be a good choice for very high-dimensional tasks (e.g. text classifiers) compared to fast, accurate linear models
Like decision trees, handles a mixture of features types	
Easily parallelized across multiple CPU's	

Table 4.2: Random Forests: Pros and Cons

Random Forest Classifier: Key Parameters

- `n_estimators`: sets the number of trees to use in ensemble (default: 10).
  - Should be larger for larger datasets to reduce overfitting (but uses more computation).

  `max_features`: has a strong effect on performance. Influences the diversity of trees in the forest. Smaller values of max features tending to reduce overfitting.
  - Default (for classification: square root of the total number of features, for regression: log base two of the total number of features) works well in practice, but adjusting may lead to some further gains.
- `max_depth`: controls the depth of each tree in the ensemble (default: None. Splits until all leaves are pure).
- `n_jobs`: How many cores to use in parallel during training. Generally, you can expect something close to a linear speed up.
  - Set -1, it will use all the cores on your system.
  - Set *number* > number of cores on your system won't have any additional effect.
- Choose a fixed setting for the `random_state` parameter if you need reproducible results.

4.2.2 Gradient Boosted Decision Trees (GBDT)

Like random forest, gradient boosted trees used an ensemble of multiple tress to create more powerful prediction models for classification and regression. These models can be used for regression and classification. Unlike the random forest method that builds and combines a forest of randomly different trees in parallel, the key idea of gradient boosted decision trees is that they build a series of trees. Where each tree is trained, so that it attempts to correct the mistakes of the previous tree in the series.

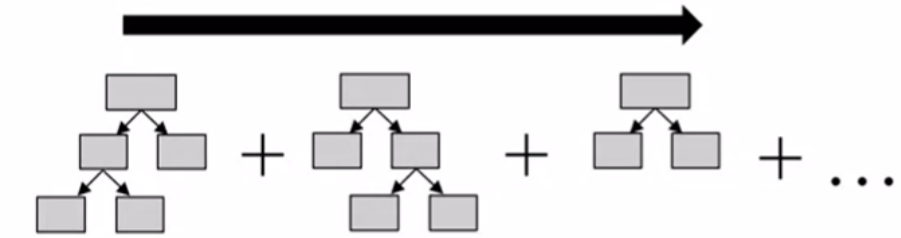


Figure 4.4: GBDT build a series of trees

Typically, gradient boosted tree ensembles use lots of shallow trees known in machine learning as weak learners. Built in a nonrandom way, to create a model that makes fewer and fewer mistakes as more trees are added. Once the model is built, making predictions with a gradient boosted tree models is fast and doesn't use a lot of memory. Like random forests, the number of estimators in the gradient boosted tree ensemble is an important parameter in controlling model complexity. A new parameter that does not occur with random forest is something called the learning rate. The learning rate controls how the gradient boost the tree algorithms, builds a series of collective trees. The learning rate controls how hard each new tree tries to correct remaining mistakes from previous round.

- High learning rate: each successive tree put strong emphases on correcting the mistakes of its predecessor. More complex individual trees, and those overall are more complex model.
- Low learning rate: there's less emphasis on thoroughly correcting the errors of the previous step, which tends to lead to simpler trees at each step.

Here's an example showing how to use gradient boosted trees in scikit-learn on our sample fruit classification test, plotting the decision regions that result.

```
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import train_test_split
from adspy_shared_utilities import plot_class_regions_for_classifier_subplot

X_train, X_test, y_train, y_test = train_test_split(X_D2, y_D2, random_state = 0)
fig, subaxes = plt.subplots(1, 1, figsize=(6, 6))

clf = GradientBoostingClassifier().fit(X_train, y_train)
title = 'GBDT, complex binary dataset, default settings'
plot_class_regions_for_classifier_subplot(clf, X_train, y_train, X_test,
                                          y_test, title, subaxes)

plt.show()
```

We then create the GradientBoostingClassifier object, and fit it to the training data in the usual way. By default, the learning rate parameter is set to 0.1, the n_estimators parameter giving the number of trees to use is set to 100, and the max depth is set to 3. As with random forests, you can see the decision boundaries have that box-like shape that's characteristic of decision trees or ensembles of trees.

Now let's apply gradient boosted decision trees to the breast cancer dataset.

```
from sklearn.ensemble import GradientBoostingClassifier

X_train, X_test, y_train, y_test = train_test_split(X_cancer, y_cancer,
                                                    random_state = 0)

clf = GradientBoostingClassifier(random_state = 0)
clf.fit(X_train, y_train)

print('Breast cancer dataset (learning_rate=0.1, max_depth=3)')
print('Accuracy of GBDT classifier on training set: {:.2f}'
     .format(clf.score(X_train, y_train)))
print('Accuracy of GBDT classifier on test set: {:.2f}\n'
     .format(clf.score(X_test, y_test)))

clf = GradientBoostingClassifier(learning_rate = 0.01, max_depth = 2,
                                                    random_state = 0)
clf.fit(X_train, y_train)

print('Breast cancer dataset (learning_rate=0.01, max_depth=2)')
print('Accuracy of GBDT classifier on training set: {:.2f}'
     .format(clf.score(X_train, y_train)))
print('Accuracy of GBDT classifier on test set: {:.2f}'
     .format(clf.score(X_test, y_test)))
```

This code trains two different gradient boosted classifiers. The first one uses the default settings. We can see that the first result has perfect accuracy on the training set, which indicates the model is likely overfitting. Two ways to learn a less complex gradient boosted tree model are, to reduce the learning rate, so that each tree doesn't try as hard to learn a more complex model, that fixes the mistakes of its predecessor. And to reduce the max_depth parameter for the individual trees in the ensemble. The second classifier example makes these changes in the parameters. And you can see, that the training set accuracy does decrease, while the test set accuracy increases slightly.

Strengths, weaknesses and parameters

Pros	Cons
Often best off-the-shelf accuracy on many problems.	Like random forests, the models are often difficult for humans to interpret.
Using model for prediction requires only modest memory and is fast. Doesn't require careful normalization of features to perform well.	Requires careful tuning of the learning rate and other parameters.
Like decision trees, handles a mixture of features types.	Training can require significant computation.
	Like decision trees not recommended for test classification and other problems with very high dimensional sparse features, for accuracy and computational cost reasons.

Table 4.3: GBDT: Pros and Cons

GBDT: Key Parameters

- `n_estimators`: sets the number of small decision trees to use (weak learners:combine many simple models, like shallow trees) in the ensemble.
- `learning_rate`: controls emphasis on fixing errors from previous iteration.
- The above two are typically tuned together. Since making the learning rates smaller, will require more trees to maintain model complexity.
- `n_estimators` is adjusted first, to best exploit memory and CPUs during training, then other parameters. Unlike random forest, increasing an n_estimators can lead to overfitting.
- `max_depth` is typically set to a small value (e.g. 3-5) for most applications.

4.3 Neural Networks

4.3.1 Introduction

This is an introduction to the basics of neural networks[1]. Which are a broad family of algorithms that have formed the basis for the recent resurgence in the computational field called *deep learning*. And just recently, has experienced a resurgence of interest, as deep learning has achieved impressive state-of-the-art results. On specific tasks that range from object classification in images, to fast accurate machine translation, to gameplay. Here, we will only discuss some relatively simple methods, namely *Multilayer perceptrons (MLPs)* for classification and regression, that can serve as a starting point for more involved deep learning methods. MLPs are also know as (vanilla) feed-forward neural networks.

[1] Check out the excellent course called Neural Networks for Machine Learning by Professor Hinton.

4.3.2 The neural network model

Review: Regression and Classification

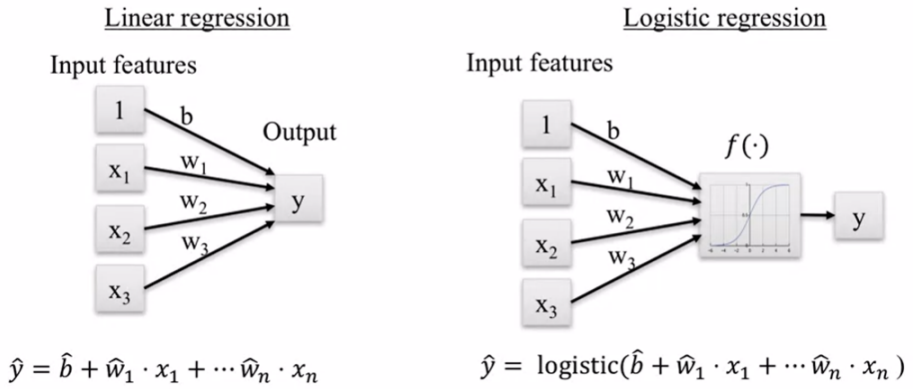


Figure 4.5: Review: Linear and Logistic Regression

- Linear regression: predicts a continuous output, $\hat{y}$, shown as the box on the right. As a function as the sum of the input variables x_i, shown in the boxes on the left. Each weighted by a corresponding coefficient, $\hat{w}_i$, plus an intercept or bias term, $\hat{b}$. We saw how various methods like ordinary least squares, ridge regression or lasso regression. Could be used to estimate these model coefficients, $\hat{w}_i$ and $\hat{b}$, shown above the arrows in the diagram, from training data.
- Logistic regression: takes this one step further, by running the output of the linear function of the input variables, x_i. Through an additional nonlinear function, the logistic function. Represented by the new box in the middle of the diagram, to produce the output, $\hat{y}$. Which, because of the logistic function, is now constrained to lie between zero and one. We use logistical regression for binary classification. Since we can interpret y as the probability that a given input data instance belongs to the positive class, in a two-class binary classification scenario.

MLPs can be viewed as generalization of linear models that perform multiple stages of processing to come to a decision.

Example of a simple neural network for regression, called a multi-layer perceptron (MLP). These are also known as feed-forward neural networks.

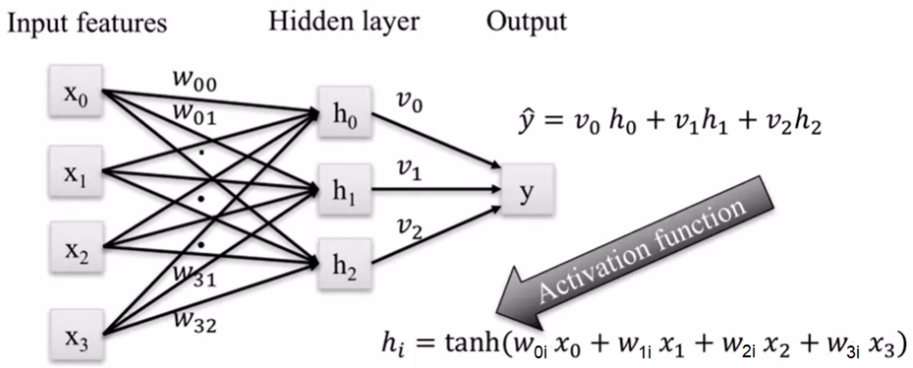


Figure 4.6: Multi-layer Perceptron with One Hidden Layer (and tanh activation function)

MLPs take this idea of computing weighted sums of the input features, like we saw in logistic regression. But it takes it a step beyond logistic regression, by adding an additional processing step called a hidden layer. Represented by this additional set of boxes, h_0, h_1, and h_2 in the diagram. These boxes, within the hidden layer, are called hidden units. And each hidden unit in the hidden layer computes a nonlinear function of the weighted sums of the input features. Resulting in intermediate output values, v_0, v_1, v_2. Then the MLP computes a weighted sum of these hidden unit outputs, to form the final output value, $\hat{y}$. This nonlinear function that the hidden unit applies. is called the activation function. In this example, your activation function is the hyperbolic tangent function, which is related to the logistic function. You can see that the result of adding this additional hidden layer processing step to the prediction model, is a formula for $\hat{y}$. That is already more involved than the one for logistic regression. Now predicting y involves computing a different initial weighted sum of the input feature values for each hidden unit. Which applies a nonlinear activation function. And then all of these nonlinear outputs are combined, using another weighted sum, to produce y.

In particular, there's one weight between each input and each hidden unit. And one weight between each hidden unit and the output variable. In fact, this addition and combination of non-linear activation functions. Allows multi-layer perceptrons to learn more complex functions. Than is possible with a simple linear or logistic function. This additional expressive power enables neural networks to perform more accurate prediction. When the relationship between the input and output is itself complex. Of course, this complexity also means that there are a lot more weights, model coefficients, to estimate in the training phase. Which means that both more training data and more computation are typically needed to learn in a neural network, compared to a linear model. As an aside, there are a number of choices for the activation function in a neural network, that gets applied in hidden units. Here, the plot shows the input value coming into the activation function, from the previous layer's inputs on the x-axis. And the y-axis shows the resulting output value for the function. The three main activation functions we'll compare later in this lecture are:

- The hyperbolic tangent. That's the S-shaped function in green.
- The rectified linear unit function, which I'll abbreviate to relu, shown as the piecewise linear function in blue.
- And the familiar logistic function, which is shown in red.

```
xrange = np.linspace(-2, 2, 200)

plt.figure(figsize=(7,6))

plt.plot(xrange, np.maximum(xrange, 0), label = 'relu')
plt.plot(xrange, np.tanh(xrange), label = 'tanh')
plt.plot(xrange, 1 / (1 + np.exp(-xrange)), label = 'logistic')
plt.legend()
plt.title('Neural network activation functions')
plt.xlabel('Input value (x)')
plt.ylabel('Activation function output')

plt.show()
```

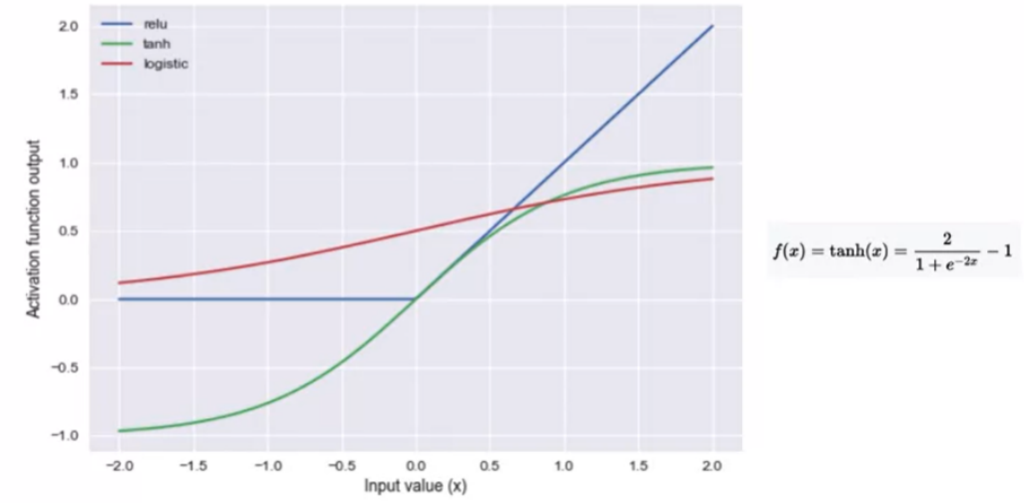


Figure 4.7: Activation Functions

The relu activation function is the default activation function for neural networks in scikit-learn. It maps any negative input values to zero. The hyperbolic tangent function, or tanh function. Maps large positive input values to outputs very close to one. And large negative input values, to outputs very close to negative one. These differences in the activation function can have some effect on the shape of regression prediction plots. Or classification decision boundaries that neural networks learn. In general, we'll be using either the hyperbolic tangent or the relu function as our default activation function. Since these perform well for most applications.

This graphic plots the results of running this code. To show how the number of hidden units in a single layer in the neural network affects the model complexity for classification.

- With a single hidden unit, the model is mathematically equivalent to logistic regression. We see the classifier returns the familiar simple linear decision boundary between the two classes.

The training set score's low, and the test score is not much better, so this network model is under-fitting.

- With ten hidden units, we can see that the MLPClassifier is able to learn a more complete decision boundary. That captures more of the nonlinear, cluster-oriented structure in the data, though the test set accuracy is still low.
- With 100 hidden units, the decision boundary is even more detailed. And achieves much better accuracy, on both the training and the test sets.

```
# Example: Synthetic binary classification dataset1: single hidden layer
from sklearn.neural_network import MLPClassifier
from adspy_shared_utilities import plot_class_regions_for_classifier_subplot

X_train, X_test, y_train, y_test = train_test_split(X_D2, y_D2, random_state=0)

fig, subaxes = plt.subplots(3, 1, figsize=(6,18))

for units, axis in zip([1, 10, 100], subaxes):
    # This code example shows the classifier being fit to the training data,
    # using a single hidden layer. With three different numbers of hidden units
    # in the layer, 1 unit, 10 units and 100 units.
    nnclf = MLPClassifier(hidden_layer_sizes = [units], solver='lbfgs',
                          random_state = 0).fit(X_train, y_train)

    title = 'Dataset 1: Neural net classifier, 1 layer, {} units'.format(units)

    plot_class_regions_for_classifier_subplot(nnclf, X_train, y_train,
                                              X_test, y_test, title, axis)
    plt.tight_layout()
```

Example of a multi-layer perceptron with two hidden layers

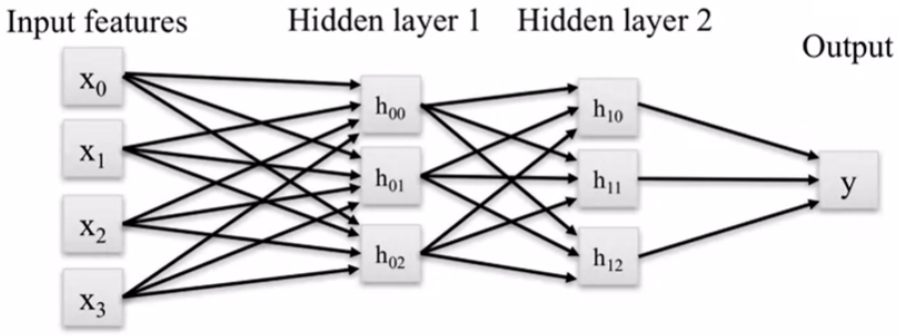


Figure 4.8: A Multi-layer Perceptron with Two Hidden Layers

Adding the second hidden layer further increases the complexity of functions that the neural network can learn, from more complex data sets. Taking this complexity further, large architectures of neural networks, with many stages of computation, are why deep learning methods are called deep.

```
# Synthetic dataset 1: two hidden layers
from adspy_shared_utilities import plot_class_regions_for_classifier

X_train, X_test, y_train, y_test = train_test_split(X_D2, y_D2, random_state=0)

# create a two-layer MLP, with 10 hidden units in each layer.
nnclf = MLPClassifier(hidden_layer_sizes = [10, 10], solver='lbfgs',
                      random_state = 0).fit(X_train, y_train)

plot_class_regions_for_classifier(nnclf, X_train, y_train, X_test, y_test,
                      'Dataset 1: Neural net classifier, 2 layers, 10/10 units')
```

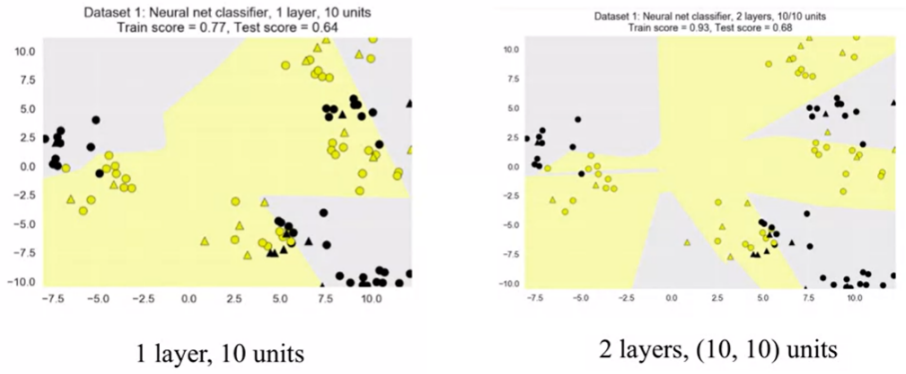


Figure 4.9: One vs Two Hidden Layers

You can see the result of of adding the second hidden layer, on the classification problem we saw earlier. On the left is the original MLP, with one hidden layer of ten units. And on the right is the same data set, using a new MLP with two hidden layers of ten units each. You can see the MLP with two hidden layers learned a more complex decision boundary. And achieved, in this case, a much better fit on the training data, and slightly better accuracy on the test data. Once we start adding more hidden layers, with lots of hidden units. You can see that the number of weights, or model coefficients, to estimate for a neural network can increase rapidly. So that more complex neural networks could have many thousands of weights to estimate.

Regularization parameter: alpha

We can control this model complexity, just as we did with ridge and lasso regression. By adding an L2 regularization penalty on the weights. Remember that L2 regularization penalizes models that have a large sum of squares of all the weight values. With the effect being, that the neural network

prefers models with more weights shrunk close to zero. The regularization parameter for MLPs is called alpha, like with the linear regression models. And in scikit-learn, it's set to a small value by default, like 0.0001, that gives a little bit of regularization.

```
X_train, X_test, y_train, y_test = train_test_split(X_D2, y_D2, random_state=0)

fig, subaxes = plt.subplots(4, 1, figsize=(6, 23))

for this_alpha, axis in zip([0.01, 0.1, 1.0, 5.0], subaxes):
    nnclf = MLPClassifier(solver='lbfgs', activation = 'tanh',
                          alpha = this_alpha,
                          hidden_layer_sizes = [100, 100],
                          random_state = 0).fit(X_train, y_train)

    title = 'Dataset 2: NN classifier, alpha = {:.3f} '.format(this_alpha)

    plot_class_regions_for_classifier_subplot(nnclf, X_train, y_train,
                                              X_test, y_test, title, axis)
    plt.tight_layout()
```

This code example shows the effects of changing alpha for a larger MLP, with 2 hidden layers of 100 nodes each. From a small value of 0.01, to a larger value of 5.0. For variety here, we're also setting the activation function to use the hyperbolic tangent function. You can see (Fig.4.10) the effect of increasing regularization with increasing alpha.

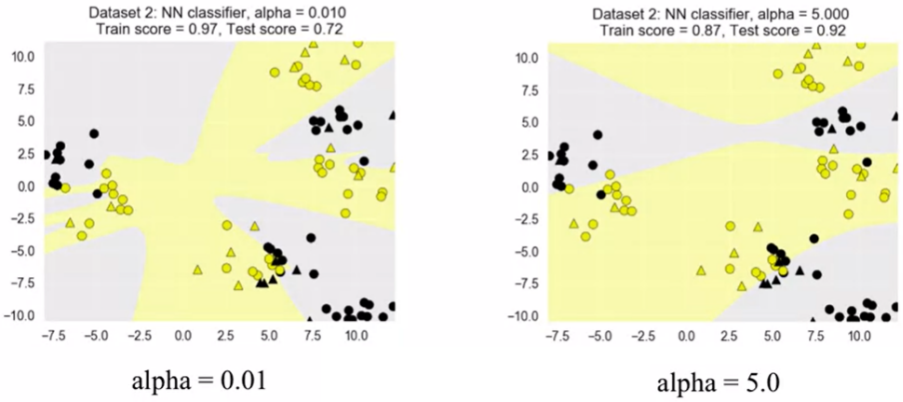


Figure 4.10: L2 Regularization with the Alpha Parameter

In the left plot, when alpha is small, the decision boundaries are much more complex and variable. And the classifier's over-fitting, as we can see from the very high training set score, and low test score. On the other hand, the right plot uses the largest value of alpha here, alpha 5.0. And that setting results in much smoother decision boundaries, while still capturing the global structure of the data. And this increased simplicity allows it to generalize much better, and not over-fit to the training set. And this is evident from the much higher test score, in this case.

Normalize

As with other supervised learning models, like regularized regression and support vector machines. It can be critical, when using neural networks, to properly normalize the input features. Let's apply the multi-layer perceptron to the breast cancer data set. And notice that we first apply the `MinMaxScaler`, to pre-process the input features.

```
# Application to real-world dataset for classification
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import MinMaxScaler


scaler = MinMaxScaler()

X_train, X_test, y_train, y_test = train_test_split(X_cancer, y_cancer,
                                                    random_state = 0)
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Use two hidden layers with 100 units each  With a higher regularization
# setting of alpha at 5.0, and using the lgbfs solver
clf = MLPClassifier(hidden_layer_sizes = [100, 100], alpha = 5.0,
                    random_state = 0, solver='lbfgs').fit(X_train_scaled, y_train)

print('Breast cancer dataset')
print('Accuracy of NN classifier on training set: {:.2f}'
     .format(clf.score(X_train_scaled, y_train)))
print('Accuracy of NN classifier on test set: {:.2f}'
     .format(clf.score(X_test_scaled, y_test)))
```

Like many of the other supervised learning methods we've seen, you can also use multi-layer perceptrons for regression, as well as classification.

4.3.3 Neuronal Networks: Regression

We're including MLP regression here, as an example, for two reasons. First, because MLP regression may be useful for some regression problems on its own. But more generally, because some deep learning problems are regression problems. And so, as with classification, using multi-layer perceptrons is a good starting point to learn about the more complex architectures used for regression in deep learning.

```
from sklearn.neural_network import MLPRegressor

fig, subaxes = plt.subplots(2, 3, figsize=(11,8), dpi=70)

X_predict_input = np.linspace(-3, 3, 50).reshape(-1,1)

X_train, X_test, y_train, y_test = train_test_split(X_R1[0::5], y_R1[0::5],
```

```
                                                    random_state = 0)

# This code has a loop that cycles through different settings of the
# activation function parameter, and the alpha parameter for L2 regularization.
for thisaxisrow, thisactivation in zip(subaxes, ['tanh', 'relu']):
    for thisalpha, thisaxis in zip([0.0001, 1.0, 100], thisaxisrow):
        mlpreg = MLPRegressor(hidden_layer_sizes = [100,100],
                              activation = thisactivation,
                              alpha = thisalpha,
                              solver = 'lbfgs').fit(X_train, y_train)
        y_predict_output = mlpreg.predict(X_predict_input)
        thisaxis.set_xlim([-2.5, 0.75])
        thisaxis.plot(X_predict_input, y_predict_output,
                      '^', markersize = 10)
        thisaxis.plot(X_train, y_train, 'o')
        thisaxis.set_xlabel('Input feature')
        thisaxis.set_ylabel('Target value')
        thisaxis.set_title('MLP regression\nalpha={}, activation={})'
                           .format(thisalpha, thisactivation))
        plt.tight_layout()
```

Again, as with classification, the effect of increasing the amount of L2 regularization, by increasing alpha. Is to constrain the regression to use simpler and simpler models, with fewer and fewer large weights.

Strengths, weaknesses and parameters

Pros	Cons
They form the basis of state-of-the-art models and can be formed into advanced architectures that effectively capture complex features given enough data and computation.	Larger, more complex models require significant training time, data, and customization. Careful preprocesssing of the data is needed. A good choice when the features are of similar types, but less so when features of very different types.

Table 4.4: Neuronal Networks: Pros and Cons

Neural Networks: Key Parameters

Here the main parameters are:

- `hidden_layer_sizes`: This parameter is a list, with one element for each hidden layer, that gives the number of hidden units to use for that layer. So here we're passing a list with a single element. Meaning we want one hidden layer, using the number in the variable called

units. By default, if you don't specify the `hidden_layer_sizes` parameter, scikit-learn will create a single hidden layer with 100 hidden units. While a setting of 10 may work well for simple data sets, like the one we use as examples here. For really complex data sets, the number of hidden units could be in the thousands.

- `alpha`: controls the amount of regularization that helps constrain the complexity of the model, by constraining the magnitude of model weights.
- `activation` parameter: you can experiment with at least three different choices for the non-linear activation function.
- `solver`: Which specifies the algorithm to use for learning the weights of the network. The solver can end up at different local minima, which can have different validation scores.he default solver, adam, tends to be both efficient and effective on large data sets, with thousands of training examples. For small data sets, like many of the ones we use in these examples, the lbfgs solver tends to be faster, and find more effective weights.
- `random_state` parameter: It is fixed to zero. This is because for neural networks, their weights are initialized randomly, which can affect the model that is learned. That means that even when using exactly the same parameters, we can obtain very different models when using different random seeds

4.3.4 Deep Learning

One of the key challenges in machine learning is finding the right features to use as input to a learning model for a particular problem. This is called feature engineering and can be part art, and part science. It can also be the single most factor in doing well on a learning task. Sometimes, in fact, more often more important than the choice of the model itself. Because of the difficulty of feature engineering, there's been a lot of research on what's called feature learning or feature extraction algorithms that can find good features automatically. This brings us to deep learning.

- Deep learning combine a sophisticated automatic feature extraction phase with a supervised learning phase.
- Moreover, deep learning is called deep. Because this feature extraction typically doesn't use just one feature learning step, the feature extraction phase uses a hierarchy of multiple feature extraction layers. Here's one simplified example of what a deep learning architecture might look like in practice for an image recognition task. Fig.4.11
- Starting from primitive, low-level features in the initial layer, each feature layer's output provides the input features to the next higher feature layer.
- All features are used in the final supervised learning model.

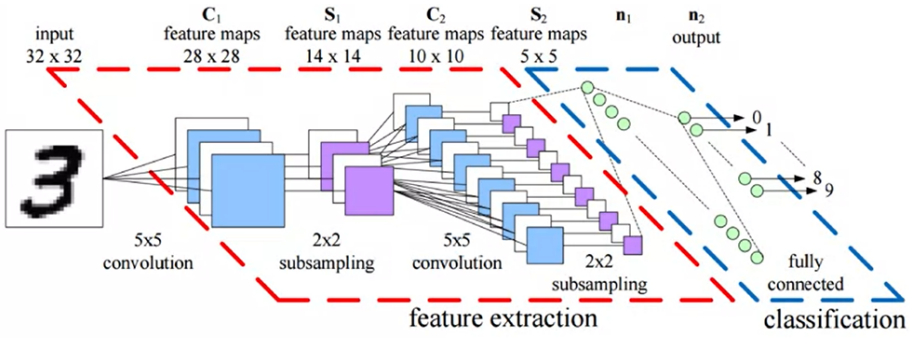


Image: M. Peemen, B. Mesman, and H. Corporaal. Efficiency Optimization of Trainable Feature Extractors for a Consumer Platform. Proceedings of the 13th International Conference on Advanced Concepts for Intelligent Vision Systems, 2011.

Figure 4.11: An example of a simple deep learning architecture

Strengths, weaknesses and parameters

- Pros:
  - Powerful: deep learning has achieved significant gains over other machine learning approaches on many difficult learning tasks, leading to state-of-art performance across many different domains.
  - Does effective automatic feature extraction, reducing the need for guesswork and heuristic on this key problem.
  - Current software provides flexible architecture that can be adapted for new domains fairly easily.
- Cons:
  - Can require huge amounts of training data.
  - Can require huge amounts of computing power.
  - Architectures can be complex and often must be highly tailored to a specific application.

Well, scikit-learn with the MLP classifier and MLP regressor classes provides a useful environment to learn about and apply simple neural networks. In particular, software packages usable with Python include Keras and Lasagne which in turn use libraries that include TensorFlow and Theano.

4.4 Data Leakage

In data science, the term data leakage sometimes just referred to as leakage, describes the situation where the data you're using to train the machine learning algorithm happens to include unexpected extra information about the very thing you're trying to predict. Basically, leakage occurs any time that information is introduced about the target label or value during training that would not legitimately be available during actual use.

- Including the true label of a data instance as a feature in the model.
- Having test data accidentally included in the training data which leads to over fitting.

When data leakage does occur, it typically causes results during your model development phase that are too optimistic, followed by the nasty surprise of disappointing results after the prediction model is actually deployed and evaluated on new data. In other words, leakage can cause your system to learn a sub optimal model that does much worse in actual deployment than a model developed in a leak free setting.

4.4.1 Detecting Data Leakage

- Before building the model
  - Exploratory data analysis to find surprises in the data.
  - Are there features very highly correlated with the target value?
- After building the model
  - Look for surprising feature behavior in the fitted model.
  - Are there features with very high weights, or high information gain?
  - Simple rule-based models like decision trees can help with features like account numbers, patient IDs
  - Is overall model performance surprisingly good compared to known results on the same dataset, or for similar problems on similar datasets?
- Limited real-world deployment of the trained model
  - Potentially expensive in terms of development time, but more realistic
  - Is the trained model generalizing well to new data?

There are practices you can follow to help reduce the chance of data leakage in your application. One important rule is to make sure that you perform any data preparation within each cross-validation fold, separately. In other words, if you're scaling or normalizing features, any statistics or parameters that you estimate for this should only be based on the data available in the cross-validation split and not the entire data set. You should also make sure that you use these same parameters on the corresponding held out test fold. If you're working with time series data, keep track of the time stamp that's associated with processing a particular data instance, such as a user's click on a webpage and make sure any data used to compute features for this instance does not include records with a later time than the cutoff value. This will help ensure you're not including information from the future in your current feature calculations or training data. If you have enough data, consider splitting off a completely separate test set before you even start working with a new dataset, and then evaluating your final model and this test data only as a very last step. The goal here is similar to doing a real world deployment to check that your train model does generalize reasonably well to new data. If there's no significant drop in performance, great. But if there is, leakage maybe one contributing factor, along with the usual suspects like classical over fitting.

Chapter 5

Unsupervised Machine Learning

The second family of machine learning algorithms is unsupervised learning algorithms. Unsupervised learning subsumes all kinds of machine learning where there is no known target values, labels or output, no teacher to instruct the learning algorithm. In unsupervised learning, the learning algorithm is just shown the input data or unlabeled samples and asked to extract knowledge from this data. The job of the unsupervised learning algorithm is to take the raw data and capture some interesting structure in it. Applications of unsupervised learning:

- Visualize structure of a complex dataset.
- Density estimation to predict probabilities of events.
- Compress and summarize the data.
- Extract features for supervised learning.
- Discover important structures like clusters or outliers in the data.

5.1 Types of Unsupervised Learning Methods

- Transformations of the dataset
  - Run the original data through some kind of useful processes that extract or compute information of some kind
  - A common application is dimensionality reduction, which takes a high-dimensional representation of the data, consisting of many features, and finds a new way to represent this data that summarizes the essential characteristics with fewer features. A common application is the reduction to two dimensions for visualization purposes.
- Clustering
  - Find groups in the data
  - Assign every point in the dataset to one of the groups

– Organize your pictures in a social media site, the sight might want to group together pictures that show the same person.

A major challenge in unsupervised learning is evaluating whether the algorithm learned something useful, because there is no way for us to tell the algorithm what we are looking for, and often the only way to evaluate the result of an unsupervised algorithm is to inspect manually.

As a consequence, unsupervised algorithms are used often in an exploratory setting, when a data scientist wants to understand the data better, rather than as part of a larger automatic system. Another common application is as a preprocessing step for supervised learning algorithms. Learning a new representation of the data can sometimes improve the accuracy of supervised algorithms, or can lead to reduced memory and time consumption.