Peyman Hessari

Machine Learning

Monograph

June 18, 2020

Springer

Contents

1

Introduction

1.1 Machine learning definition and types

Arthur Samuel defined machine learning as *"the field of study that gives computers the ability to learn without being explicitly programmed."*
Tom Mitchell defined machine learning by saying that, *"a computer program is said to learn from experience E, with respect to some class of tasks T, and some performance measure P, if its performance on T as measured by P improves with experience E."*
There are three different types of machine learning:

1. **Supervised Learning:** The main goal in supervised learning is to learn a model from labeled training data that allows us to make predictions about unseen or future data. A supervised learning task with discrete class labels is called a classification task. Another subcategory of supervised learning is regression, where the outcome signal is a continuous value.
   There are issues with supervised learning that must be taken into account. The bias-variance dilemma is one of them: how the machine learning model performs accurately using different training sets. High bias models contain restricted learning sets, whereas high variance models learn with complexity against noisy training data. There's a trade-off between the two models. The key is where to settle with the trade-off and when to apply which type of model.
2. **Unsupervised Learning:** In unsupervised learning, we are dealing with unlabeled data or data of unknown structure. Using unsupervised learning techniques, we are able to explore the hidden pattern and structure of our data to extract meaningful information without the guidance of a known outcome variable or reward function. Clustering and dimensionality reduction are two subfield of unsupervised learning.
   With unsupervised learning there is no right or wrong answer; it's just a case of running the machine learning algorithm and seeing what patterns and outcomes occur

3. **Reinforcement Learning:** In reinforcement learning, the goal is to develop a system (agent) that improves its performance based on interactions with the environment. Since the information about the current state of the environment typically also includes a so-called reward signal, we can think of reinforcement learning as a field related to supervised learning. However, in reinforcement learning this feedback is not the correct ground truth label or value, but a measure of how well the action was measured by a reward function. Through the interaction with the environment, an agent can then use reinforcement learning to learn a series of actions that maximizes this reward via an exploratory trial-and-error approach or deliberative planning.

1.2 Application of Machine Learning

Machine Learning is being used in different areas such as

- Software
- Spam Detection
- Voice Recognition
- Stock Trading
- Robotics
- Medicine and Health care
- Advertising
- Retail and E-commerce
- Gaming Analytics
- The Internet of Things

1.3 Machine Learning Plane

The Machine Learning Cycle:

A machine learning project is basically a cycle of actions that need to be performed

- **Data acquisition:** You can acquire data from many sources; it might be data that's held by your organization or open data from the Internet. There might be one dataset, or there could be ten or more.
- **Prepare data:** The data will need to be cleaned and checked for quality before any processing can take place.
- **Process:** The machine learning routines that you have created perform this phase.
- **Report:** Present the results

It All Starts with a Question
Defining the process:

1. **Planning:** Planning might take into account where the data is coming from, if it needs to be cleaned, what learning methods to use and what the output is going to look like. The main point is that these things can be changed at any time.
2. **Developing:** This process might involve algorithm development or code development. Agile development processes work best.
3 **Testing:** When you test, you have time to change things
4. **Reporting:** Sit down with the stakeholders and discuss the test results. Do the results make sense?
5. **Refining:** When everyone is happy with test results, it's time to refine code and, if possible, the algorithms and optimize it.
6. **Production:** When all is tested, reviewed, and refined by the team, move it to production Make sure the team reviews the first few production runs to ensure the results are as expected, and then look at the project as a whole and see if it's meeting the criteria of the stakeholders. Things might need to be refined.

1.4 Introduction

Suppose that we observe a quantitative response Y and p different predictors, $X_1, X_2, \cdots, X_p$. We assume that there is some relationship between Y and $X = (X_1, X_2, \cdots, X_p)$, which can be written in the very general form

$$Y = f(X) + \epsilon.$$

Here f is some fixed but unknown function of $X_1, X_2, \cdots, X_p$ and ϵ is a random error term, which is independent of X and has mean zero. In this formulation, f represents the systematic information that X provides about Y. However, the function f that connects the input variable to the output variable is in general unknown. In this situation one must estimate f based on the observed points. There are two main reasons that we may wish to estimate f : prediction and inference.

In many situations, a set of inputs X are readily available, but the output Y cannot be easily obtained. In this setting, since the error term averages to zero, we can predict Y using

$$\hat{Y} = \hat{f}(X),$$

where $\hat{f}$ represents our estimate for f, and $\hat{Y}$ represents the resulting prediction for Y.

The accuracy of $\hat{Y}$ as a prediction for Y depends on two quantities, which we will call the reducible error and the irreducible error. In general, $\hat{f}$ will not be a perfect estimate for f, and this inaccuracy will introduce some error. This error is reducible because we can potentially improve the accuracy of $\hat{f}$ by using the most appropriate statistical learning technique to estimate f.

However, even if it were possible to form a perfect estimate for f, so that our estimated response took the form $\hat{Y} = \hat{f}(X)$, our prediction would still have some error in it! This is because Y is also a function of ϵ, which, by definition, cannot be predicted using X. Therefore, variability associated with ϵ also affects the accuracy of our predictions. This is known as the irreducible error, because no matter how well we estimate f, we cannot reduce the error introduced by ϵ. The irreducible error will always provide an upper bound on the accuracy of our prediction for Y . This bound is almost always unknown in practice.

How Do We Estimate f?

We will always assume that we have observed a set of n different data points. These observations are called the training data. Our goal is to apply a statistical learning method to the training data in order to estimate the unknown function f. Broadly speaking, most statistical learning methods for this task can be characterized as either parametric or non-parametric.

- *Parametric Methods:* Parametric methods involve a two-step model-based approach
  1. First, we make an assumption about the functional form, or shape, of f. For example, one very simple assumption is that f is linear in X :

     $$f(X) = \beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p.$$

     Once we have assumed that f is linear, the problem of estimating f is greatly simplified. Instead of having to estimate an entirely arbitrary p-dimensional function $f(X)$, one only needs to estimate the $p + 1$ coefficients $\beta_0, \beta_1, \cdots, \beta_p$.
  2. After a model has been selected, we need a procedure that uses the training data to fit or train the model.

  The parametric method reduces the problem of estimating f down to one of estimating a set of parameters. The potential disadvantage of a parametric approach is that the model we choose will usually not match the true unknown form of f.

  Parametric model summarizes data with a set of parameters of fixed size (independent of the number of training examples). No matter how much data you throw at a parametric model, it won't change its mind about how many parameters it needs.

  Examples of parametric machine learning algorithms include ordinary least squares; Logistic Regression; Linear Discriminant Analysis; Perceptron; Naive Bayes; Simple Neural Networks.
- *Non-parametric Methods:* Non-parametric methods do not make explicit assumptions about the functional form of f. Instead they seek an estimate of f that gets as close to the data points as possible without being too rough or wiggly. Such approaches can have a major advantage over parametric approaches: by avoiding the assumption of a particular functional form for f, they have the potential to accurately fit a wider range

of possible shapes for f. Any parametric approach brings with it the possibility that the functional form used to estimate f is very different from the true f, in which case the resulting model will not fit the data well. In contrast, non-parametric approaches completely avoid this danger, since essentially no assumption about the form of f is made. But non-parametric approaches do suffer from a major disadvantage: since they do not reduce the problem of estimating f to a small number of parameters, a very large number of observations is required in order to obtain an accurate estimate for f.
Examples of popular nonparametric machine learning algorithms are: thin-plate splines; k-Nearest Neighbors; Decision Trees like CART and C4.5; Support Vector Machines;

Parametric methods have several advantages. They are often easy to fit, because one need estimate only a small number of coefficients. In the case of linear regression, the coefficients have simple interpretations, and tests of statistical significance can be easily performed. But parametric methods do have a disadvantage: by construction, they make strong assumptions about the form of $f(x)$. If the specified functional form is far from the truth, and prediction accuracy is our goal, then the parametric method will perform poorly.

In contrast, non-parametric methods do not explicitly assume a parametric form for $f(x)$, and thereby provide an alternative and more flexible approach for performing regression. The term non-parametric refers to the fact that the amount of stuff we need to keep in order to represent the hypothesis f grows linearly with the size of the training set.

The parametric approach will outperform the nonparametric approach if the parametric form that has been selected is close to the true form of f. As a general rule, parametric methods will tend to outperform non-parametric approaches when there is a small number of observations per predictor.

1.5 The Bias-Variance Trade-Off

It is possible to show that the expected test MSE, for a given value x, can always be decomposed into the sum of three fundamental quantities: the variance of $\hat{f}(x)$, the squared bias of $\hat{f}(x)$ and the variance of the error variance terms ϵ. That is

$$E(y - \hat{f}(x))^2 = Var(\hat{f}(x)) + [Bias(\hat{f}(x))]^2 + Var(\epsilon),$$

Here the notation $E(y - \hat{f}(x))^2$ defines the expected test MSE, and refers expected to the average test MSE that we would obtain if we repeatedly estimated test MSE f using a large number of training sets, and tested each at x. The equation tells us that in order to minimize the expected test error, we need to select a statistical learning method that simultaneously achieves low

variance and low bias. Note that variance is inherently a nonnegative quantity, and squared bias is also nonnegative. Hence, we see that the expected test MSE can never lie below $Var(\epsilon)$, the irreducible error.

Variance refers to the amount by which $\hat{f}$ would change if we estimated it using a different training data set. Since the training data are used to fit the statistical learning method, different training data sets will result in a different $\hat{f}$. But ideally the estimate for f should not vary too much between training sets. However, if a method has high variance then small changes in the training data can result in large changes in $\hat{f}$. In general, more flexible statistical methods have higher variance. Bias refers to the error that is introduced by approximating a real-life problem, which may be extremely complicated, by a much simpler model. Generally, more flexible methods result in less bias.

As a general rule, as we use more flexible methods, the variance will increase and the bias will decrease. The relative rate of change of these two quantities determines whether the test MSE increases or decreases. As we increase the flexibility of a class of methods, the bias tends to initially decrease faster than the variance increases. Consequently, the expected test MSE declines. However, at some point increasing flexibility has little impact on the bias but starts to significantly increase the variance. When this happens the test MSE increases. See figure 9.1.

2

Data Preprocessing

Preprocessing refers to the transformations applied to our data before feeding it to the algorithm. Data preprocessing is a technique that is used to convert the raw data into a clean data set. In other words, whenever the data is gathered from different sources it is collected in raw format which is not feasible for the analysis.

For achieving better results from the applied model in machine learning projects the format of the data has to be in a proper manner. Some specified machine learning model needs information in a specified format, for example, random forest algorithm does not support null values, therefore to execute random forest algorithm null values have to be managed from the original raw data set. Another aspect is that data set should be formatted in such a way that more than one machine learning and deep learning algorithms are executed in one data set, and best out of them is chosen.

2.1 Importing Libraries

```
# Importing the libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

2.2 Importing Data Set

We explain various methods to read data into Python. Data can be in any of the popular formats - CSV, TXT, XLS/XLSX (Excel), sas7bdat (SAS), Rdata (R), and json.

```
# Importing csv file
dataset = pd.read_csv('Data.csv')
# Import File from URL
dataset  = pd.read_csv("http://***.com/Data.csv")
# Read Text File
dataset = pd.read_table("C:\\Desktop\\Data.txt")
dataset  = pd.read_csv("C:\\Desktop\\Data.txt", sep ="\
    t")
# Read Excel File
dataset = pd.read_excel("Data.xls")
# Read SAS File
dataset = pd.read_sas('Data.sas7bdat')
# Read SQL Table
import sqlite3
from pandas.io import sql
conn = sqlite3.connect('C:/Data.db')
query = "SELECT * FROM Data;"
dataset = pd.read_sql(query, con=conn)

from sqlalchemy import create_engine
# Create a connection to the database
database_connection = create_engine('sqlite:///sample.
    db')
dataframe = pd.read_sql_query('SELECT * FROM data',
    database_connection)

import json
with open(path) as f:
json.loads(f.read())
```

2.3 Missing Data

Data can have missing values for a number of reasons such as observations that were not recorded and data corruption. Handling missing data is important as many machine learning algorithms do not support data with missing values.

The simplest strategy for handling missing data is to remove records that contain a missing value. Removing rows with missing values can be too limiting on some predictive modeling problems, an alternative is to impute missing values.

Imputing refers to using a model to replace missing values. There are many options we could consider when replacing a missing value, for example:

- A constant value that has meaning within the domain, such as 0, distinct from all other values.

- A value from another randomly selected record.
- A mean, median or mode value for the column.
- A value estimated by another predictive model.

Any imputing performed on the training dataset will have to be performed on new data in the future when predictions are needed from the finalized model. This needs to be taken into consideration when choosing how to impute the missing values.

For example, if you choose to impute with mean column values, these mean column values will need to be stored to file for later use on new data that has missing values.

```
from sklearn.preprocessing import Imputer
imputer = Imputer(missing_values = 'NaN', strategy = '
    mean', axis = 0)
imputer = imputer.fit(X[:, columns with missing data])
X[:, columns] = imputer.transform(X[:, columns])
```

ML algorithm: There are algorithms that can be made robust to missing data, such as k-Nearest Neighbors that can ignore a column from a distance measure when a value is missing.

```
from fancyimpute import KNN
features_knn_imputed = KNN(k=5, verbose=0).complete(
    standardized_features)
```

2.4 Categorical Data

The most common way to represent categorical variables is using the *one-hot encoding* or *one-out-of-N encoding*, also known as dummy variables. The idea behind dummy variables is to replace a categorical variable with one or more new features that can have the values 0 and 1. The values 0 and 1 make sense in all models in *scikit-learn*, and we can represent any number of categories by introducing one new feature per category.

This is important to ensure categorical values are represented in the same way in the training set and the test set.

```
from sklearn.preprocessing import LabelEncoder,
    OneHotEncoder
labelencoder_X = LabelEncoder()
X[:, 0] = labelencoder_X.fit_transform(X[:, 0])
onehotencoder = OneHotEncoder(categorical_features =
    [0])
X = onehotencoder.fit_transform(X).toarray()
```

```
labelencoder_y = LabelEncoder()
y = labelencoder_y.fit_transform(y)
```

Or we can use dummy variables in pandas

```
data_dummies = pd.get_dummies(data)
```

If the categorical features are integers, we should first change the data type to string by using astype(str) and then use get_dummies.

2.5 Splitting Data into training/test set

```
# Splitting dataset into the Training set and Test set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X,
    y, test_size = 0.2, random_state = 0)
```

2.6 Feature Scaling

Feature scaling through standardization or normalization can be an important preprocessing step for many machine learning algorithms. Standardization involves rescaling the features such that they have the properties of a standard normal distribution with a mean of zero and a standard deviation of one.

Since the range of values of raw data varies widely, in some machine learning algorithms, objective functions will not work properly without normalization. For example, many classifiers calculate the distance between two points by the Euclidean distance. If one of the features has a broad range of values, the distance will be governed by this particular feature. Therefore, the range of all features should be normalized so that each feature contributes approximately proportionately to the final distance.

Another reason why feature scaling is applied is that gradient descent converges much faster with feature scaling than without it.

Methods:

1. **Rescaling:** Also known as min-max scaling or min-max normalization, is the simplest method and consists in rescaling the range of features to scale the range in $[0, 1]$ or $[-1, 1]$. Selecting the target range depends on the nature of the data. The general formula for a min-max of $[0, 1]$ is given as:

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

where x is an original value, x' is the normalized value.

MinMaxScaler in *scikit-learn* does it and shifts the data such that all features are exactly between 0 and 1.

2. **Mean normalization:**

$$x' = \frac{x - \text{average}(x)}{\max(x) - \min(x)}$$

where x is an original value, x' is the normalized value.

3. **Standardization (Z-score Normalization):** In machine learning, we can handle various types of data, e.g. audio signals and pixel values for image data, and this data can include multiple dimensions. Feature standardization makes the values of each feature in the data have zero-mean and unit-variance. This method is widely used for normalization in many machine learning algorithms (e.g., support vector machines, logistic regression, and artificial neural networks).

$$x' = \frac{x - \bar{x}}{\sigma}$$

where x is the original feature vector, $\bar{x}$ is the mean of that feature vector, and σ is its standard deviation.
StandardScaler in *scikit-learn* does it.

4. **Robust Scaler:** The *RobustScaler* uses a rather similar method to the *MinMaxScaler*. However, it removes the median and uses the interquartile range, which makes it robust to outliers. It follows the following formula for each feature:

$$x' = \frac{x - \text{median}(x)}{Q_3(x) - Q_1(x)},$$

where Q is the quartile of the data.

5. **Normalization:** Normalization is the process of *scaling individual samples to have unit norm.* This process can be useful if you plan to use a quadratic form such as the dot-product or any other kernel to quantify the similarity of any pair of samples. This usually means dividing each component by the Euclidean length of the vector:

$$x' = \frac{x}{\|x\|}.$$

Normalizer in *scikit-learn* does it.

```
from sklearn.preprocessing import Normalizer
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
sc_y = StandardScaler()
y_train = sc_y.fit_transform(y_train)
```

2.7 Automatic Feature Selection

With so many ways to create new features, you might get tempted to increase the dimensionality of the data way beyond the number of original features. However, adding more features makes all models more complex, and so increases the chance of overfitting. When adding new features, it can be a good idea to reduce the number of features to only the most useful ones, and discard the rest. This can lead to simpler models that generalize better. There are three basic strategies: univariate statistics, model-based selection, and iterative selection. All of these methods are supervised methods, meaning they need the target for fitting the model. This means we need to split the data into training and test sets, and fit the feature selection only on the training part of the data.

a. **Univariate statistics:** In univariate statistics, we compute whether there is a statistically significant relationship between each feature and the target. Then the features that are related with the highest confidence are selected. In the case of classification, this is also known as analysis of variance (ANOVA). A key property of these tests is that they are univariate, meaning that they only consider each feature individually. Consequently, a feature will be discarded if it is only informative when combined with another feature. Univariate tests are often very fast to compute, and don't require building a model. They are completely independent of the model that you might want to apply after the feature selection.
To use univariate feature selection in *scikit-learn*, you need to choose a test, usually either *f_classif* for classification or *f_regression* for regression, and a method to discard features based on the p-values determined in the test. All methods for discarding parameters use a threshold to discard all features with too high a p-value. The methods differ in how they compute this threshold, with the simplest ones being *SelectKBest*, which selects a fixed number k of features, and *SelectPercentile*, which selects a fixed percentage of features.

```
from sklearn.feature_selection import
    SelectPercentile
from sklearn.model_selection import train_test_split
X, y = dataset
X_train, X_test, y_train, y_test = train_test_split(X
    , y, random_state=0, test_size=.5)
select = SelectPercentile(percentile=50)
select.fit(X_train, y_train)
X_train_selected = select.transform(X_train)
print(select.get_support()) #the selected features
```

b. **Model-based selection:** Model-based feature selection uses a supervised machine learning model to judge the importance of each feature,

and keeps only the most important ones. The supervised model that is used for feature selection doesn't need to be the same model that is used for the final supervised modeling. The feature selection model needs to provide some measure of importance for each feature, so that they can be ranked by this measure. Decision trees and decision tree–based models provide a *feature_importances_* attribute, which directly encodes the importance of each feature. Linear models have coefficients, which can also be used to capture feature importances by considering the absolute values. Linear models with L1 penalty learn sparse coefficients, which only use a small subset of features. This can be viewed as a form of feature selection for the model itself, but can also be used as a preprocessing step to select features for another model. In contrast to univariate selection, model-based selection considers all features at once, and so can capture interactions. To use model-based feature selection, we need to use the *SelectFromModel* transformer.

```
from sklearn.feature_selection import SelectFromModel
from sklearn.ensemble import RandomForestClassifier
select = SelectFromModel(RandomForestClassifier(
    n_estimators=100, random_state=42),threshold="
    median")
select.fit(X_train, y_train)
X_train_l1 = select.transform(X_train)
```

c. **Iterative selection:** In univariate testing we used no model, while in model-based selection we used a single model to select features. In iterative feature selection, a series of models are built, with varying numbers of features. There are two basic methods: starting with no features and adding features one by one until some stopping criterion is reached, or starting with all features and removing features one by one until some stopping criterion is reached. Because a series of models are built, these methods are much more computationally expensive than the methods we discussed previously. One particular method of this kind is recursive feature elimination (RFE), which starts with all features, builds a model, and discards the least important feature according to the model. Then a new model is built using all but the discarded feature, and so on until only a prespecified number of features are left. For this to work, the model used for selection needs to provide some way to determine feature importance, as was the case for the model-based selection.

```
from sklearn.feature_selection import RFE
select = RFE(RandomForestClassifier(n_estimators=100,
     random_state=42), n_features_to_select=40)
select.fit(X_train, y_train)
X_train_rfe= select.transform(X_train)
X_test_rfe= select.transform(X_test)
```

RFE can be used with cross validation too.

```
from sklearn.feature_selection import RFECV
select = RFE(estimator=ols, step=1, scoring="
    neg_mean_squared_error")
select.fit(X_train, y_train)
X_train_rfe= select.transform(X_train)
X_test_rfe= select.transform(X_test)
```

3

Linear Regression

3.1 Assumptions of Linear Regression

Linear regression is an analysis that assesses whether one or more predictor variables explain the dependent variable. The regression has five key assumptions:

1. **Linear Relationship between the features and target:** The linear regression model assumes that there is a straight-line relationship between the predictors and the response. If the true relationship is far from linear, then virtually all of the conclusions that we draw from the fit are suspect. In addition, the prediction accuracy of the model can be significantly reduced.
   Residual plots are a useful graphical tool for identifying nonlinearity. Given a simple linear regression model, we can plot the residuals, $e_i = y_i - \hat{y}_i$ versus the predictor x_i. In the case of a multiple regression model since there are multiple predictors, we instead plot the residuals versus the predicted values $\hat{y}_i$. Ideally, the residual plot will show no fitted discernible pattern. The presence of a pattern may indicate a problem with some aspect of the linear model.
   If the residual plot indicates that there are non-linear associations in the data, then a simple approach is to use non-linear transformations of the featurea, such as $\log x$, $\sqrt{x}$, and x^2, in the regression model.
2. **Little or no Multicollinearity between the features:** Multicollinearity is a state of very high inter-correlations or inter-associations among the independent variables. The presence of collinearity can pose problems in the regression context, since it can be difficult to separate out the individual effects of collinear variables on the response.
   Since collinearity reduces the accuracy of the estimates of the regression coefficients, it causes the standard error to grow. Recall that the t-statistic for each predictor is calculated by dividing the coefficient by its standard error. Consequently, collinearity results in a decline in the t-statistic. As

a result, in the presence of collinearity, we may fail to reject the null hypothesis. This means that the power of the hypothesis test is reduced by collinearity.

A simple way to detect collinearity is to look at the correlation matrix of the predictors. An element of this matrix that is large in absolute value indicates a pair of highly correlated variables, and therefore a collinearity problem in the data. Unfortunately, not all collinearity problems can be detected by inspection of the correlation matrix: it is possible for collinearity to exist between three or more variables even if no pair of variables has a particularly high correlation. We call this situation multicollinearity. Instead of inspecting the correlation matrix, a better way to assess multi-collinearity is to compute the variance inflation factor (VIF). The VIF is the ratio of the variance of θ_j when fitting the full model divided by the variance of θ_j if fit on its own. The smallest possible value for VIF is 1, which indicates the complete absence of collinearity. As a rule of thumb, a VIF value that exceeds 5 or 10 indicates a problematic amount of collinearity.

Pair plots and heatmaps (correlation matrix) can be used for identifying highly correlated features.

If we have 2 features which are highly correlated we can drop one feature or combine the 2 features to form a new feature,which can further be used for prediction.

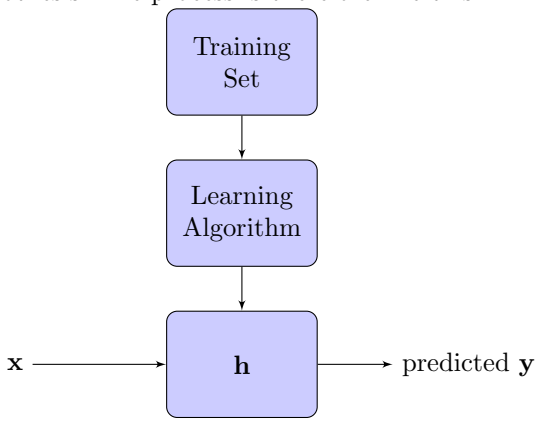
3. **Homoscedasticity Assumption:** Homoscedasticity describes a situation in which the error term is the same across all values of the independent variables. A scatter plot of residual values vs predicted values is a good way to check for homoscedasticity. There should be no clear pattern in the distribution and if there is a specific pattern, the data is heteroscedastic.
   When faced with this problem, one possible solution is to transform the response y using a concave function such as $\log y$ or $\sqrt{y}$. Such a transformation results in a greater amount of shrinkage of the larger responses, leading to a reduction in heteroscedasticity.
4. **Normal distribution of error terms:** The fourth assumption is that the error follow a normal distribution. However, a less widely known fact is that, as sample sizes increase, the normality assumption for the residuals is not needed.
5. **Little or No autocorrelation in the residuals:** Autocorrelation occurs when the residual errors are dependent on each other. The presence of correlation in error terms drastically reduces model's accuracy. If there is correlation among the error terms, then the estimated standard errors will tend to underestimate the true standard errors. As a result, confidence and prediction intervals will be narrower than they should be. For example, a 95% confidence interval may in reality have a much lower probability than 0.95 of containing the true value of the parameter. In addition, p-values associated with the model will be lower than they should be; this could cause us to erroneously conclude that a parameter is statistically

significant. In short, if the error terms are correlated, we may have an unwarranted sense of confidence in our model.
This usually occurs in time series models where the next instant is dependent on previous instant.

3.2 Model Representation

To establish notation for future use, we will use $x^{(i)}$ to denote the input variables, also called input features, and $y^{(i)}$ to denote the output or target variable that we are trying to predict. A pair $(x^{(i)}, y^{(i)})$ is called a training example, and the dataset that we will be using to learn — a list of m training examples $(x^{(i)}, y^{(i)})$; $i = 1, \ldots, m$—is called a training set. We will also use X to denote the space of input values, and Y to denote the space of output values.

To describe the supervised learning problem, our goal is, given a training set, to learn a function $h : X \to Y$ so that $h(x)$ is a good predictor for the corresponding value of y. For historical reasons, this function h is called a hypothesis. The process is therefore like this:



When the target variable that we are trying to predict is continuous, we call the learning problem a regression problem. When y can take on only a small number of discrete values, we call it a classification problem. The hypothesis function for the input feature x with output variable y is

$$h_{\theta_0,\theta_1}(x) = \theta_0 + \theta_1 x.$$

Here θ_i's are called parameters (weights) and need to be chosen.

Linear regression, or ordinary least squares (OLS), is the simplest and most classic linear method for regression. Linear regression finds the parameters θ that minimize the mean squared error between predictions and the true regression targets, y, on the training set. The mean squared error is the sum of the squared differences between the predictions and the true values. Linear

regression has no hyper-parameters, which is a benefit, but it also has no way to control model complexity.

3.2.1 Cost Function

We can measure the accuracy of our hypothesis function by using a cost function. This takes an average difference of all the results of the hypothesis with inputs from x's and the actual output y's.

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2.$$

This function is called the squared error function, or mean squared error. The mean is halved $\left(\frac{1}{2}\right)$ as a convenience for the computation of the gradient descent, as the derivative term of the square function will cancel out the $\frac{1}{2}$ term.

If we try to think of it in visual terms, our training data set is scattered on the xy plane. We are trying to make a straight line (defined by $h_\theta(x)$) which passes through these scattered data points.

Our objective is to get the best possible line. The best possible line will be such that the average squared vertical distances of the scattered points from the line will be the least. Ideally, the line should pass through all the points of our training data set. In such a case, the value of $J(\theta_0, \theta_1)$ will be 0. Thus as a goal, we should try to minimize the cost function.

3.2.2 Gradient Descent

We have our hypothesis function and we have a way of measuring how well it fits into the data. Now we need to estimate the parameters in the hypothesis function so to minimize $J(\theta_0, \theta_1)$. To do so, lets use a search algorithm that starts with some "initial guess" for θ, and that repeatedly changes θ to make $J(\theta)$ smaller, until hopefully we converge to a value of θ that minimizes $J(\theta)$. Specifically, lets consider the gradient descent algorithm, which starts with some initial θ, and repeatedly performs the update:

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

where $j = 0, 1$ represents the feature index number. At each iteration j, one should simultaneously update the parameters θ_0, θ_1. Updating a specific parameter prior to calculating another one on the j^{th} iteration would yield to a wrong implementation.

The parameter α is called the learning rate. A smaller α would result in a smaller step and a larger α results in a larger step. We should adjust our parameter α to ensure that the gradient descent algorithm converges in a

reasonable time. Failure to converge or too much time to obtain the minimum value imply that our step size is wrong. IF α is too small, gradient descent can be slow. If α is too large, it can overshot the minimum. It may fail to converge or even diverge.

3.2.3 Gradient Descent For Linear Regression

When gradient descent applied to the linear regression, a new form of the equation can be derived. We can substitute our actual cost function and our actual hypothesis function and modify the equation to:

repeat until convergence:

$$\begin{aligned} \theta_0 &:= \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \\ \theta_1 &:= \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x^{(i)}. \end{aligned} \tag{3.1}$$

The point of all this is that if we start with a guess for our hypothesis and then repeatedly apply these gradient descent equations, our hypothesis will become more and more accurate. This method looks at every example in the entire training set on every step, and is called batch gradient descent. Note that, while gradient descent can be susceptible to local minimum, the optimization problem we have posed here for linear regression has only one global optima; thus gradient descent always converges (assuming the learning rate is not too large) to the global minimum. Indeed, J is a convex quadratic function.

```
from sklearn.linear_model import LinearRegression
X, y =dataset
X_train, X_test, y_train, y_test = train_test_split(X,
    y, random_state=42)
lr = LinearRegression().fit(X_train, y_train)

print("Training set score: {:.2f}".format(lr.score(
    X_train, y_train)))
print("Test set score: {:.2f}".format(lr.score(X_test,
    y_test)))
```

3.3 Probabilistic interpretation

We will give a set of probabilistic assumptions, under which least-squares regression is derived as a very natural algorithm. Let us assume that the target variables and the inputs are related via the equation

$$y^{(i)} = \theta^T x^{(i)} + \epsilon^{(i)}$$

where $\epsilon^{(i)}$ is an error term that captures either unmodeled effects or random noise. Let us further assume that the $\epsilon^{(i)}$ are distributed IID (independently and identically distributed) according to a Gaussian distribution with mean zero and variance σ^2. The density of $\epsilon^{(i)}$ is given by

$$p(\epsilon^{(i)}) = \frac{1}{\sqrt{2\pi}\sigma} \exp(-\frac{(\epsilon^{(i)})^2}{2\sigma^2}).$$

This implies that

$$p(y^{(i)}|x^{(i)};\theta) = \frac{1}{\sqrt{2\pi}\sigma} \exp(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}).$$

Note that we should not condition on θ since θ is not a random variable. Given X and θ, what is the distribution of the $y^{(i)}$? The probability of the data is given by $p(\mathbf{y}|X;\theta)$. This quantity is typically viewed a function of $\mathbf{y}$ for a fixed value of θ. When we wish to explicitly view this as a function of θ, we will instead call it the likelihood function:

$$L(\theta) = L(\theta; X, \mathbf{y}) = p(\mathbf{y}|X;\theta).$$

Note that by the independence assumption on the $\epsilon^{(i)}$'s this can also be written

$$\begin{aligned} L(\theta) &= \prod_{i=1}^{m} p(y^{(i)}|x^{(i)};\theta) \\ &= \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi}\sigma} \exp(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}). \end{aligned}$$

Now, given this probabilistic model relating the $y^{(i)}$'s and the $x^{(i)}$'s, what is a reasonable way of choosing our best guess of the parameters θ? The principal of maximum likelihood says that we should should choose θ so as to make the data as high probability as possible. I.e., we should choose θ to maximize $L(\theta)$. Instead of maximizing $L(\theta)$, we maximize the log likelihood $\ell(\theta)$:

$$\begin{aligned} \ell(\theta) &= \log L(\theta) \\ &= \log \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi}\sigma} \exp(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}) \\ &= m \log \frac{1}{\sqrt{2\pi}\sigma} - \frac{1}{\sigma^2} \frac{1}{2} \sum_{i=1}^{m} (y^{(i)} - \theta^T x^{(i)})^2. \end{aligned}$$

Hence, maximizing $\ell(\theta)$ gives the same answer as minimizing

$$\frac{1}{2}\sum_{i=1}^{m}(y^{(i)} - \theta^T x^{(i)})^2 = J(\theta).$$

Under the previous probabilistic assumptions on the data, least-squares regression corresponds to finding the maximum likelihood estimate of θ. Note also that, th choice of θ did not depend on what was σ^2 and indeed we'd have arrived at the same result even if σ^2 were unknown.

3.4 Multivariate Linear Regression

Linear regression with multiple variables is also known as multivariate linear regression. We now introduce notation for equations where we can have any number of input variables.

$x_j^{(i)}$: value of feature j in the $i-$th training example
$x^{(i)}$: the input (features) of the $i-$th training example
x_j : the $j-$th feature
m : the number of training examples
n : the number of features

The multi-variable form of the hypothesis function accommodating these multiple features is as follows:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \cdots + \theta_n x_n.$$

Using the definition of matrix multiplication, our multivariable hypothesis function can be concisely represented as:

$$h_\theta(x) = [\theta_0 \ \theta_1 \ \cdots \ \theta_n] \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} = \theta^T x.$$

Note that for convenience reason we assume $x_0^{(i)} = 1$ for $(i \in \{1, \ldots, m\})$. The cost function for multiple features is

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2.$$

The gradient descent algorithm for multiple variables is as follows
repeat until convergence:

$$\theta_j := \theta_j - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} \qquad \text{for } j = 0, 1, \ldots, n. \tag{3.2}$$

To debug gradient descent plot the cost function, $J(\theta)$ over the number of iterations of gradient descent. If $J(\theta)$ ever increases, then you probably need to decrease α.

Automatic convergence test: Declare convergence if $J(\theta)$ decreases by less than ϵ in one iteration, where ϵ is some small value such as 10^{-3}. However in practice it is difficult to choose this threshold value. It has been proven that if learning rate α is sufficiently small, then $J(\theta)$ will decrease on every iteration.

There is an alternative to batch gradient descent that also works very well. Consider the following algorithm:

repeat until convergence:

$$\begin{array}{l} \text{for i=1 to m\{} \\ \theta_j := \theta_j - \alpha(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} \qquad \text{for } j = 0, 1, \ldots, n. \\ \} \end{array}$$

In this algorithm, we repeatedly run through the training set, and each time we encounter a training example, we update the parameters according to the gradient of the error with respect to that single training example only. This algorithm is called stochastic gradient descent (incremental gradient descent). Whereas batch gradient descent has to scan through the entire training set before taking a single step. Stochastic gradient descent can start making progress right away, and continues to make progress with each example it looks at. Often, stochastic gradient descent gets θ "close" to the minimum much faster than batch gradient descent. Note however that it may never converge to the minimum, and the parameters θ will keep oscillating around the minimum of $J(\theta)$, but in practice most of the values near the minimum will be reasonably good approximations to the true minimum. For these reasons, particularly when the training set is large, stochastic gradient descent is often preferred over batch gradient descent.

3.4.1 Extensions of Regression

We can improve our features and the form of our hypothesis function in a couple different ways.

- **Interaction:** We can combine multiple features into one. For example, we can combine x_1 and x_2 into a new feature x_3 by taking $x_1 \cdot x_2$.
  The hierarchical principle states that if we include an interaction in a model, we should also include the main effects, even if the p-values associated with their coefficients are not significant. In other words, if the interaction between x_1 and x_2 seems important, then we should include both x_1 and x_2 in the model even if their coefficient estimates have large p-values. The rationale for this principle is that if $x_1 \times x_2$ is related to the response, then whether or not the coefficients of x_1 or x_2 are exactly zero

is of little interest. Also $x_1 \times x_2$ is typically correlated with x_1 and x_2, and so leaving them out tends to alter the meaning of the interaction.

- **Polynomial Regression:** Our hypothesis function need not be linear (a straight line) if that does not fit the data well. We can change the behavior or curve of our hypothesis function by making it a quadratic, cubic or square root function (or any other form). For example, if our hypothesis function is $h_\theta(x) = \theta_0 + \theta_1 x_1$ then we can create additional features based on x_1, to get the quadratic function $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2$ or the cubic function $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2 + \theta_3 x_1^3$. In the cubic version, we have created new features x_2 and x_3 where $x_2 = x_1^2$ and $x_3 = x_1^3$. To make it a square root function, we could do: $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 \sqrt{x_1}$.
  One important thing to keep in mind is, if you choose your features this way then feature scaling becomes very important.

```
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
X, y = dataset
polynomial = PolynomialFeatures(degree=3, include_bias=
    False)
X_polynomial = polynomial.fit_transform(X)
reg = LinearRegression()
model = reg.fit(features_polynomial, y)
```

3.5 Normal Equation

Gradient descent gives one way of minimizing J. Let's discuss a second way of doing so, this time performing the minimization explicitly and without resorting to an iterative algorithm. In the normal equation method, we will minimize J by explicitly taking its derivatives with respect to the θ_j's, and setting them to zero. This allows us to find the optimum θ without iteration. The normal equation formula is given below:

$$\theta = (X^T X)^{-1} X^T y.$$

There is no need to do feature scaling with the normal equation.

Table 3.1 is a comparison of gradient descent and the normal equation.

With the normal equation, computing the inversion has complexity $\mathcal{O}(n^3)$. So if we have a very large number of features, the normal equation will be slow. In practice, when n exceeds $10,000$ it might be a good time to go from a normal solution to an iterative process.

Table 3.1. Normal Equation vs Gradient Descent

Gradient Descent	Normal Equation
Need to choose α	No need to choose α
Need many iterations	No need to iterate
Works well when n is large	Slow if n is very large
$\mathcal{O}(kn^2)$	$\mathcal{O}(n^3)$, need to compute $(X^TX)^{-1}$

3.6 Potential Problems

1. **Outlier.** An outlier is a point for which y_i is far from the value predicted by model. Outliers can arise for a variety of reasons, such as incorrect recording of an observation during data collection. Outliers can be removed if they have little effect on the model. Even if an outlier does not have much effect on the model fit, it can cause other problems, such as increasing RSE or decreasing R^2.
   Residual plots can be used to identify outliers. But in practice, it can be difficult to decide how large a residual needs to be before we consider the point to be an outlier. To address this problem, instead of plotting the residuals, we can plot the studentized residuals, computed by dividing each residual e_i by its estimated standard studentized error. Observations whose studentized residuals are greater than 3 in absolute value are possible outliers.
2. **High Leverage Points.** Observations with high leverage high leverage have an unusual value for x_i. Removing the high leverage observation has a much more substantial impact on the least squares line than removing the outlier. In fact, high leverage observations tend to have a sizable impact on the estimated regression line. It is cause for concern if the least squares line is heavily affected by just a couple of observations, because any problems with these points may invalidate the entire fit. For this reason, it is important to identify high leverage observations.
   In order to quantify an observation's leverage, we compute the leverage statistic. A large value of this statistic indicates an observation with high leverage leverage. For a simple linear regression,

$$h_i = \frac{1}{n} + \frac{(x_i - \bar{x})^2}{\sum_{j=1}^{n}(x_j - \bar{x})^2}.$$

   The leverage statistic h_i is always between $1/n$ and 1, and the average leverage for all the observations is always equal to $(p+1)/n$. So if a given observation has a leverage statistic that greatly exceeds $(p+1)/n$, then we may suspect that the corresponding point has high leverage.

3.7 Locally weighted linear regression

In the linear regression algorithm, to make a prediction at a point x we would fit θ to minimize $\sum_{i=1}^{m}(y^{(i)} - h_\theta(x^{(i)})^2$, and then output $\theta^T x$.

In contrast, the locally weighted linear regression algorithm fits θ to minimize $\sum_{i=1}^{m} w^{(i)}(y^{(i)} - h_\theta(x^{(i)})^2$, and then output $\theta^T x$.

Here, the $w^{(i)}$'s are non-negative valued weights. Intuitively, if $w^{(i)}$ is large for a particular value of i, then in picking θ, we'll try hard to make $(y^{(i)} - h_\theta(x^{(i)})^2$ small. If $w^{(i)}$ is small, then the $(y^{(i)} - h_\theta(x^{(i)})^2$ error term will be pretty much ignored in the fit.

A fairly standard choice for the weights is

$$w^{(i)} = \exp(-\frac{(x^{(i)} - x)^2}{2\tau^2})$$

Note that the weights depend on the particular point. Moreover, if $|x^{(i)} - x|$ is small, then $w^{(i)}$ is close to 1; and if $|x^{(i)} - x|$ is large, then $w^{(i)}$ is small. Hence, θ is chosen giving a much higher weight to the training examples close to the point x. The parameter τ controls how quickly the weight of a training example falls off with distance of its $x^{(i)}$ from the query point x; τ is called the bandwidth parameter.

3.8 Ridge regression

Ridge regression is also a linear model for regression, so the formula it uses to make predictions is the same one used for ordinary least squares. In ridge regression, though, the coefficients θ 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 θ should be close to zero. Intuitively, this means each feature should have as little effect on the outcome as possible, 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 known as L_2 regularization.

```
from sklearn.linear_model import Ridge
X, y =dataset
X_train, X_test, y_train, y_test = train_test_split(X,
    y, random_state=42)
ridge = Ridge(alpha=1.0).fit(X_train, y_train)
print("Training set score: {:.2f}".format(ridge.score(
    X_train, y_train)))
print("Test set score: {:.2f}".format(ridge.score(
    X_test, y_test)))
```

The Ridge model makes a trade-off between the simplicity of the model and its performance on the training set. How much importance the model places on simplicity versus training set performance can be specified by the user, using the alpha parameter. The optimum setting of alpha depends on the particular dataset we are using. Increasing alpha forces coefficients to move more toward zero, which decreases training set performance but might help generalization.

It should be noted that with enough training data, regularization becomes less important, and given enough data, ridge and linear regression will have the same performance. If more data is added, it becomes harder for a model to overfit, or memorize the data.

3.9 Lasso regression

An alternative to Ridge for regularizing linear regression is Lasso. The lasso restricts coefficients to be close to zero, L_1 regularization. The consequence of L_1 regularization is that when using the 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.

The Lasso also has a regularization parameter, alpha, that controls how strongly coefficients are pushed toward zero. To reduce underfitting, decrease alpha. When we do this, we also need to increase the default setting of max_iter, the maximum number of iterations to run.

```
from sklearn.linear_model import Lasso
X, y =dataset
X_train, X_test, y_train, y_test = train_test_split(X,
    y, random_state=42)

lasso = Lasso(alpha=0.01, max_iter=1000).fit(X_train,
    y_train)
print("Training set score: {:.2f}".format(lasso.score(
    X_train, y_train)))
print("Test set score: {:.2f}".format(lasso.score(
    X_test, y_test)))
print("Number of features used: {}".format(np.sum(lasso
    .coef_ != 0)))
```

A lower alpha allowed us to fit a more complex model, which worked better on the training and test data. If we set alpha too low, however, we again remove the effect of regularization and end up overfitting, with a result similar to Linear Regression.

4

Classification and Representation

To attempt classification, one method is to use linear regression and map all predictions greater than 0.5 as a 1 and all less than 0.5 as a 0. However, this method doesn't work well because classification is not actually a linear function.

The classification problem is just like the regression problem, except that the values we now want to predict take on only a small number of discrete values. For now, we will focus on the binary classification problem in which y can take on only two values, 0 and 1. Most of what we say here will also generalize to the multiple-class case. For instance, if we are trying to build a spam classifier for email, then $x^{(i)}$ may be some features of a piece of email, and y may be 1 if it is a spam mail, and 0 otherwise. Hence, $y \in \{0, 1\}$. 0 is called the negative class, and 1 the positive class. Given $x^{(i)}$, the corresponding $y^{(i)}$ is called the label for the training example.

4.1 Hypothesis Representation

We could approach the classification problem ignoring the fact that y is discrete-valued, and use our old linear regression algorithm to try to predict y for given x. However, it is easy to construct examples where this method performs very poorly. Intuitively, it also does not make sense for $h_\theta(x)$ to take values larger than 1 or smaller than 0 when we know that $y \in \{0, 1\}$. To fix this, let's change the form for our hypotheses $h_\theta(x)$ to satisfy $0 \leq h_\theta(x) \leq 1$. This is accomplished by plugging $\theta^T x$ into the Logistic Function.

Our new form uses the "Sigmoid Function," also called the "Logistic Function":

$$h_\theta(x) = g(\theta^T x) \text{ where } \ g(z) = \frac{1}{1 + e^{-z}} \text{ with } z = \theta^T x.$$

The sigmoid function is plotted in figure 4.1.

The function $g(z)$, shown here, maps any real number to the $(0, 1)$ interval, making it useful for transforming an arbitrary-valued function into a function

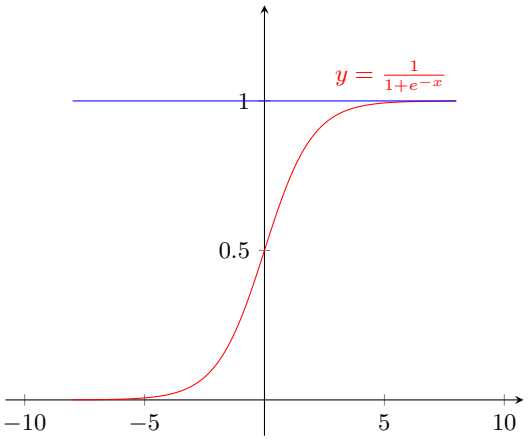


Fig. 4.1. Sigmoid Function

better suited for classification. $h_\theta(x)$ will give us the probability that our output is 1. For example, $h_\theta(x) = 0.7$ gives us a probability of 70% that our output is 1. The probability that our prediction is 0 is just the complement of the probability that it is 1 (e.g. if probability that it is 1 is 70%, then the probability that it is 0 is 30%).

$$h_\theta(x) = P(y = 1|x; \theta) = 1 - P(y = 0|x; \theta),$$

$$P(y = 0|x; \theta) + P(y = 1|x; \theta) = 1.$$

4.2 Decision Boundary

In order to get our discrete 0 or 1 classification, we can translate the output of the hypothesis function as follows:

- $h_\theta(x) \geq 0.5 \rightarrow y = 1$,
- $h_\theta(x) < 0.5 \rightarrow y = 0$.

The way our logistic function g behaves is that when its input is greater than or equal to zero, its output is greater than or equal to 0.5 :

$$g(z) \geq 0.5 \text{ when } z \geq 0.$$

So if our input to g is $\theta^T X$, then that means:

$$h_\theta(x) = g(\theta^T X) \geq 0.5 \text{ when } \theta^T X \geq 0.$$

From these statements we can now say:

- $\theta^T X \geq 0 \Rightarrow y = 1$,
- $\theta^T X < 0 \Rightarrow y = 0$.

The decision boundary is the line that separates the area where $y = 0$ and where $y = 1$. It is created by our hypothesis function. For example, for $\theta^T = [5 \ -1 \ 0]$, we have $y = 1$ if $5 - 1x_1 + 0x_2 \geq 0$ or $x_1 \leq 5$. In this case, our decision boundary is a straight vertical line placed on the graph where $x_1 = 5$, and everything to the left of that denotes $y = 1$, while everything to the right denotes $y = 0$.

The input to the sigmoid function $g(z)$ (e.g. $\theta^T X$) doesn't need to be linear, and could be a function that describes a circle (e.g. $z = \theta_0 + \theta_1 x_1^2 + \theta_2 x_2^2$) or any shape to fit our data. $\theta^T X$ is the decision boundary.

4.3 Logistic Regression Model

4.3.1 Cost Function

We cannot use the same cost function that we use for linear regression because the logistic function will cause the output to be wavy, causing many local optima. In other words, it will not be a convex function.

Instead, our cost function for logistic regression looks like:

$$\begin{aligned} J(\theta) &= \frac{1}{m}\sum_{i=1}^{m} Cost(h_\theta(x^{(i)}), y^{(i)}) \\ Cost(h_\theta(x, y) &= -\log(h_\theta(x)), \qquad \text{if } y = 1 \\ Cost(h_\theta(x, y) &= -\log(1 - h_\theta(x)), \qquad \text{if } y = 0 \end{aligned} \tag{4.1}$$

If our correct answer y is 0, then the cost function will be 0 if our hypothesis function also outputs 0. If our hypothesis approaches 1, then the cost function will approach infinity.

If our correct answer y is 1, then the cost function will be 0 if our hypothesis function outputs 1. If our hypothesis approaches 0, then the cost function will approach infinity.

Note that writing the cost function in this way guarantees that $J(\theta)$ is convex for logistic regression. We can compress our cost function's two conditional cases into one case:

$$Cost(h_\theta(x), y) = -y \log(h_\theta(x)) - (1 - y) \log(1 - h_\theta(x)).$$

We can fully write out our entire cost function as follows:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \big[-y^{(i)} \log(h_\theta(x^{(i)})) - (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})) \big].$$

4.3.2 Gradient Descent

Remember that the general form of gradient descent is:

Repeat:

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta), \qquad j = 0, 1, \ldots, n$$

We can work out the derivative part using calculus to get:

Repeat:

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}, \qquad j = 0, 1, \ldots, n.$$

Notice that this algorithm is identical to the one we used in linear regression. We still have to simultaneously update all values in θ.

For linear models for regression, the output, is a linear function of the features: a line, plane, or hyperplane. For linear models for classification, the decision boundary is a linear function of the input. In other words, a linear classifier is a classifier that separates two classes using a line, a plane, or a hyperplane.

By default, logistic regression applies an L_2 regularization, in the same way that Ridge does for regression and the trade-off parameter that determines the strength of the regularization is called C, and higher values of C correspond to less regularization. In other words, when you use a high value for the parameter C, Logistic Regression tries to fit the training set as best as possible, while with low values of the parameter C, the models put more emphasis on finding a coefficient vector that is close to zero. Using low values of C will cause the algorithms to try to adjust to the "majority" of data points, while using a higher value of C stresses the importance that each individual data point be classified correctly.

```
from sklearn.linear_model import LogisticRegression
X, y = dataset
X_train, X_test, y_train, y_test = train_test_split(X,
    y, random_state=0)
lr = LogisticRegression(C=1, penalty="l1")
lr.fit(X_train, y_train)
y_pred = lr.predict(X_test)
print("Test set accuracy: {:.2f}".format(lr.score(
    X_test, y_test)))
```

4.3.3 Probabilistic approach

Lets assume that

$$p(y = 1|x; \theta) = h_\theta(x)$$
$$p(y = 0|x; \theta) = 1 - h_\theta(x)$$

Note that this can be written more compactly as

$$p(y|x; \theta) = (h_\theta(x))^y (1 - h_\theta(x))^{1-y}.$$

Assuming that the n training examples were generated independently, we can then write down the likelihood of the parameters as

$$\begin{aligned} L(\theta) &= \prod_{i=1}^{m} p(y^{(i)}|x^{(i)}; \theta) \\ &= \prod_{i=1}^{m} (h_\theta(x^{(i)}))^{y^{(i)}} (1 - h_\theta(x^{(i)}))^{1-y^{(i)}}. \end{aligned}$$

Also,

$$\begin{aligned} \ell(\theta) &= \log L(\theta) \\ &= \sum_{i=1}^{m} \left[y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})) \right]. \end{aligned}$$

Applying gradient descent method we arrive at
Repeat:

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}, \qquad j = 0, 1, \ldots, n.$$

4.4 Multiclass Classification: One-vs-rest

Now we will approach the classification of data when we have more than two categories. Instead of $y = \{0, 1\}$, we will expand our definition so that $y = \{0, 1, \ldots, n\}$. Since $y = \{0, 1, \ldots, n\}$, we divide our problem into $n + 1$ binary classification problems; in each one, we predict the probability that y is a member of one of our classes.

$$\begin{aligned} &y = \{0, 1, \ldots, n\}, \\ &h_\theta^{(0)}(x) = P(y = 0|x; \theta); \\ &h_\theta^{(1)}(x) = P(y = 1|x; \theta); \\ &\quad \vdots \\ &h_\theta^{(n)}(x) = P(y = n|x; \theta); \\ &\text{prediction} = \max_i (h_\theta^{(i)}(x)). \end{aligned}$$

We are basically choosing one class and then lumping all the others into a single second class. We do this repeatedly, applying binary logistic regression to each case, and then use the hypothesis that returned the highest value as our prediction.

To summarize, train a logistic regression classifier $h_\theta(x)$ for each class to predict the probability that $y = i$. To make a prediction on a new x, pick the class that maximizes $h_\theta(x)$.

In scikit-learn, one-vs-rest can be implemented in the following way. Alternatively, in multinomial logistic regression (MLR), the logistic function is replaced with a softmax function and multi_class="multinomial".

```
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
X, y = dataset
X_train, X_test, y_train, y_test = train_test_split(X,
    y, random_state=0)
scaler = StandardScaler()
X_std = scaler.fit_transform(X_train)
lr = LogisticRegression(multi_class="ovr")
lr.fit(X_std, y_train)
y_pred = lr.predict(scaler.transform(X_test))
print("Test set accuracy: {:.2f}".format(lr.score(
    scaler.transform(X_test), y_test)))
```

Logistic regression can be applied with cross validation as well using LogisticRegressionCV. Imbalanced data can be handled using class_weight parameter to weight the classes to make certain we have a balanced mix of each class. Specifically, the 'balanced' argument will automatically weigh classes inversely proportional to their frequency

4.5 Strengths, weaknesses, and parameters for linear models

The main parameter of linear models is the regularization parameter, called alpha in the regression models and C in Logistic Regression. Large values for alpha or small values for C mean simple models. Usually C and alpha are searched for on a logarithmic scale. The other decision you have to make is whether you want to use L_1 regularization or L_2 regularization. If you assume that only a few of your features are actually important, you should use L_1. Otherwise, you should default to L_2. L_1 can also be useful if interpretability of the model is important. As L_1 will use only a few features, it is easier to explain which features are important to the model, and what the effects of these features are.

Linear models are very fast to train, and also fast to predict. They scale to very large datasets and work well with sparse data. If your data consists of

hundreds of thousands or millions of samples, you might want to investigate using the solver='sag' option in LogisticRegression and Ridge, which can be faster than the default on large datasets. Other options are the SGDClassifier class and the SGDRegressor class, which implement even more scalable versions of the linear models described here.

Another strength of linear models is that they make it relatively easy to understand how a prediction is made. Linear models often perform well when the number of features is large compared to the number of samples. They are also often used on very large datasets, simply because it's not feasible to train other models.

5

Decision Trees

5.1 Decision Trees

Learning a decision tree means learning the sequence of if/else questions that gets us to the true answer most quickly. In the machine learning setting, these questions are called tests. To build a tree, the algorithm searches over all possible tests and finds the one that is most informative about the target variable. Then split the dataset. We can build a more accurate model by repeating the process of looking for the best test in each regions. The top node, also called the root, represents the whole dataset. This recursive process yields a binary tree of decisions, with each node containing a test. Alternatively, you can think of each test as splitting the part of the data that is currently being considered along one axis. This yields a view of the algorithm as building a hierarchical partition. As each test concerns only a single feature, the regions in the resulting partition always have axis-parallel boundaries.

The recursive partitioning of the data is repeated until each region in the partition (each leaf in the decision tree) only contains a single target value. A leaf of the tree that contains data points that all share the same target value is called pure.

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 (or the single target in the case of pure leaves) 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 fulfilled or not. It is also possible to use trees for regression tasks, using exactly the same technique. To make a prediction, we traverse the tree based on the tests in each node and find the leaf the new data point falls into. The output for this data point is the mean target of the training points in this leaf.

Controlling the complexity:

Typically, building a tree as described here and continuing until all leaves are pure leads to models that are very complex and highly overfit to the training data. The presence of pure leaves mean that a tree is 100% accurate

on the training set; each data point in the training set is in a leaf that has the correct majority class.

There are two common strategies to prevent overfitting: stopping the creation of the tree early (also called pre-pruning), or building the tree but then removing or collapsing nodes that contain little information (also called post-pruning or just pruning). Possible criteria for pre-pruning include limiting the maximum depth of the tree, limiting the maximum number of leaves, or requiring a minimum number of points in a node to keep splitting it.

Decision trees in scikit-learn are implemented in the DecisionTreeRegressor and DecisionTreeClassifier classes. scikit-learn only implements pre-pruning, not post-pruning.

If we don't restrict the depth of a decision tree, the tree can become arbitrarily deep and complex. Unpruned trees are therefore prone to overfitting and not generalizing well to new data.

```
from sklearn.tree import DecisionTreeClassifier
X, y = dataset
X_train, X_test, y_train, y_test = train_test_split(
X, y, stratify=y, random_state=42)
tree = DecisionTreeClassifier(max_depth=4, random_state
    =0)
tree.fit(X_train, y_train)
print("Accuracy on training set: {:.3f}".format(tree.
    score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(tree.score(
    X_test, y_test)))
```

Decision tree learners attempt to find a decision rule that produces the greatest decrease in impurity at a node. While there are a number of measurements of impurity, by default DecisionTreeClassifier uses Gini impurity:

$$G(t) = 1 - \sum_{i=1}^{c} p_i^2$$

where $G(t)$ is the Gini impurity at node t and pi is the proportion of observations of class c at node t. This process of finding the decision rules that create splits to increase impurity is repeated recursively until all leaf nodes are pure or some arbitrary cut-off is reached. If we want to use a different impurity measurement we can use the criterion parameter in DecisionTreeClassifier such as criterion='entropy'.

The decision trees can be visualized.

```
from sklearn.tree import export_graphviz
import graphviz
export_graphviz(tree, out_file="tree.dot", class_names
    =[], feature_names=, impurity=False, filled=True)
```

```
with open("tree.dot") as f:
dot_graph = f.read()
graphviz.Source(dot_graph)
```

Decision tree regression works similarly to decision tree classification; however, instead of reducing Gini impurity or entropy, potential splits are by default measured on how much they reduce mean squared error.

Strengths, weaknesses, and parameters

The parameters that control model complexity in decision trees are the pre-pruning parameters that stop the building of the tree before it is fully developed. Usually, picking one of the pre-pruning strategies: setting either max_depth, max_leaf_nodes, or min_samples_leaf, is sufficient to prevent overfitting.

Decision trees have two advantages over many of the algorithms: the resulting model can easily be visualized and understood by non-experts, and the algorithms are completely invariant to scaling of the data. As each feature is processed separately, and the possible splits of the data don't depend on scaling, no preprocessing like normalization or standardization of features is needed for decision tree algorithms. In particular, decision trees work well when you have features that are on completely different scales, or a mix of binary and continuous features.

The main downside of decision trees is that even with the use of pre-pruning, they tend to overfit and provide poor generalization performance. Therefore, in most applications, the ensemble methods are usually used in place of a single decision tree.

5.2 Ensemble of Decision Trees

Ensembles are methods that combine multiple machine learning models to create more powerful models. There are many models in the machine learning literature that belong to this category, but there are two ensemble models that have 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 and gradient boosted decision trees.

5.2.1 Random forests

A random forest is essentially a collection of decision trees, where each tree is slightly different from the others. The idea behind random forests is that each tree might do a relatively good job of predicting, but will likely overfit on part of the data. If we build many trees, all of which work well and overfit in different ways, we can reduce the amount of overfitting by averaging their results.

To implement this strategy, we need to build many decision trees. Each tree should do an acceptable job of predicting the target, and should also be different from the other trees. Random forests get their name from injecting randomness into the tree building to ensure each tree is different. There are two ways in which the trees in a random forest are randomized: by selecting the data points used to build a tree and by selecting the features in each split test.

Building Random forests:

To build a random forest model, you need to decide on the number of trees to build (n_estimators parameter). Let's say we want to build 10 trees. These trees will be built completely independently from each other, and the algorithm will make different random choices for each tree to make sure the trees are distinct. To build a tree, we first take a bootstrap sample of our data. That is, from our n_samples data points, we repeatedly draw an example randomly with replacement, n_samples times. This will create a dataset that is as big as the original dataset, but some data points will be missing from it (approximately one third), and some will be repeated.

Next, a decision tree is built based on this newly created dataset. Instead of looking for the best test for each node, in each node the algorithm randomly selects a subset of the features, and it looks for the best possible test involving one of these features. The number of features that are selected is controlled by the max_features parameter. This selection of a subset of features is repeated separately in each node, so that each node in a tree can make a decision using a different subset of the features.

The bootstrap sampling leads to each decision tree in the random forest being built on a slightly different dataset. Because of the selection of features in each node, each split in each tree operates on a different subset of features. Together, these two mechanisms ensure that all the trees in the random forest are different.

A critical parameter in this process is max_features. If we set max_features to n_features, that means that each split can look at all features in the dataset, and no randomness will be injected in the feature selection. If we set max_features to 1, that means that the splits have no choice at all on which feature to test, and can only search over different thresholds for the feature that was selected randomly. Therefore, a high max_features means that the trees in the random forest will be quite similar, and they will be able to fit the data easily, using the most distinctive features. A low max_features means that the trees in the random forest will be quite different, and that each tree might need to be very deep in order to fit the data well.

To make a prediction using the random forest, the algorithm first makes a prediction for every tree in the forest. For regression, we can average these results to get our final prediction. For classification, a "soft voting" strategy is used. This means each algorithm makes a "soft" prediction, providing a probability for each possible output label. The probabilities predicted by all the trees are averaged, and the class with the highest probability is predicted

```
from sklearn.ensemble import RandomForestClassifier
X, y = dataset
X_train, X_test, y_train, y_test = train_test_split(X,
    y, random_state=0)
forest = RandomForestClassifier(n_estimators=100,
    random_state=0)
forest.fit(X_train, y_train)
print("Accuracy on training set: {:.3f}".format(forest.
    score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(forest.
    score(X_test, y_test)))
```

Strengths, weaknesses:

Random forests for regression and classification are currently among the most widely used machine learning methods. They are very powerful, often work well without heavy tuning of the parameters, and don't require scaling of the data.

Essentially, random forests share all of the benefits of decision trees, while making up for some of their deficiencies. One reason to still use decision trees is if you need a compact representation of the decision-making process.

Random forests don't tend to perform well on very high dimensional, sparse data, such as text data. For this kind of data, linear models might be more appropriate. Random forests usually work well even on very large datasets, and training can easily be parallelized over many CPU cores within a powerful computer. However, random forests require more memory and are slower to train and to predict than linear models.

5.2.2 Gradient boosted regression trees (gradient boosting machines)

The gradient boosted regression tree is another ensemble method that combines multiple decision trees to create a more powerful model. Despite the "regression" in the name, these models can be used for regression and classification. In contrast to the random forest approach, gradient boosting works by building trees in a serial manner, where each tree tries to correct the mistakes of the previous one. By default, there is no randomization in gradient boosted regression trees; instead, strong pre-pruning is used. Gradient boosted trees often use very shallow trees, of depth one to five, which makes the model smaller in terms of memory and makes predictions faster.

The main idea behind gradient boosting is to combine many simple models (weak learners), like shallow trees. Each tree can only provide good predictions on part of the data, and so more and more trees are added to iteratively improve performance.

Apart from the pre-pruning and the number of trees in the ensemble, another important parameter of gradient boosting is the learning_rate, which

controls how strongly each tree tries to correct the mistakes of the previous trees. A higher learning rate means each tree can make stronger corrections, allowing for more complex models. Adding more trees to the ensemble, which can be accomplished by increasing n_estimators, also increases the model complexity, as the model has more chances to correct mistakes on the training set.

```
from sklearn.ensemble import GradientBoostingClassifier
X, y = dataset
X_train, X_test, y_train, y_test = train_test_split(X,
    y, random_state=0)
gbrt = GradientBoostingClassifier(random_state=0,
    max_depth=1, learning_rate=0.01)
gbrt.fit(X_train, y_train)
print("Accuracy on training set: {:.3f}".format(gbrt.
    score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(gbrt.score(
    X_test, y_test)))
```

Strengths, weaknesses:

The main drawback is that they require careful tuning of the parameters and may take a long time to train. Similarly to other tree-based models, the algorithm works well without scaling and on a mixture of binary and continuous features. As with other tree-based models, it also often does not work well on high-dimensional sparse data.

Decision tree is a statistical model that is used in classification. This machine learning approach is used to classify data into classes and to represent the results in a flowchart, such as a tree structure. This model classifies data in a dataset by flowing through a query structure from the root until it reaches the leaf, which represents one class. The root represents the attribute that plays a main role in classification, and the leaf represents the class. The decision tree model follows the steps outlined below in classifying data:

1. It puts all training examples to a root.
2. It divides training examples based on selected attributes.
3. It selects attributes by using some statistical measures.
4. Recursive partitioning continues until no training example remains, or until no attribute remains, or the remaining training examples belong to the same class.

Decision tree learning is supervised, because it constructs decision tree from class-labeled training tuples. We will discuss two decision tree algorithms as follows:

1. ID3 (Iterative Dichotomiser 3)
2. C4.5 (Successor of ID3)

The statistical measure used to select attribute (that best splits the dataset in terms of given classes) in ID3 is information gain, whereas in C4.5, the criterion is gain ratio. Both measures have a close relationship with another concept called entropy.

Entropy is a measure of uncertainty in a random variable. Shannon entropy quantifies this uncertainty in terms of expected value of the information present in the message.

$$H(X) = \sum_i p(x_i) I(x_i) = -\sum_i p(x_i) \log_2 p(x_i).$$

To build a decision tree for classification, the target is to find the attribute that can reduce the entropy or information content of the whole system by splitting a single dataset into multiple datasets.

5.3 Information gain in ID3

We use information gain in ID3 to select attributes to build a decision tree. The formula of information gain for attribute A is given below:

$$Gain(A) = Info(D) - Info_A(D)$$

where:

$$Info_A(D) = \sum_{j=1}^{v} \frac{|D_j|}{|D|} Info(D_j).$$

The problem with information gain as a measure to select the attribute for partition is that in the quest of pure partitions, it can select the attributes that are meaningless from the machine learning point of view.

This drawback is due to the inherent deficiency in the measure information gain that gives preference to the attribute that can divide the parent dataset to datasets with the least amount of entropy. This bias problem leads to a proposal of another measure called gain ratio to solve this problem.

5.4 Gain ratio of C4.5

In order to solve the bias problem of the measure of information gain, another measure called gain ratio was proposed. The idea behind the new measure is very simple and intuitive. Just penalize those attributes that make many splits. The extent of partitioning is calculated by SplitInfo. This normalizes the information gain, resulting in the calculation of gain ratio.

$$SplitInfo_A(D) = \sum_{j=1}^{v} \frac{|D_j|}{|D|} \log_2(\frac{|D_j|}{|D|}),$$

$$GainRation(A) = \frac{Gain(A)}{SplitInfo(A).}$$

6

Naive Bayes

Bayes' theorem is the premier method for understanding the probability of some event, $P(A|B)$, given some new information, $P(B|A)$, and a prior belief in the probability of the event, $P(A)$:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)},$$

$P(A|B)$ is the posterior, $P(B|A)$ is the likelihood, $P(A)$ is prior, and $P(B)$ is marginal probability.

Algorithms that try to learn $p(y|x)$ directly (such as logistic regression), or algorithms that try to learn mappings directly from the space of inputs X to the labels $\{0, 1\}$, (such as the perceptron algorithm) are called discriminative learning algorithms. Here, we consider algorithms that instead try to model $p(x|y)$ and $p(y)$. These algorithms are called generative learning algorithms. For instance, if y indicates whether an example is a dog (0) or an elephant (1), then $p(x|y = 0)$ models the distribution of dogs' features, and $p(x|y = 1)$ models the distribution of elephants' features. After modeling $p(y)$ and $p(x|y)$, our algorithm can then use Bayes rule to derive the posterior distribution on y given x

$$p(y|x) = \frac{p(x|y)p(y)}{p(x)}.$$

6.1 Linear Discriminant Analysis

Logistic regression involves directly modeling $p(y = k|x)$ using the logistic function. We model the conditional distribution of the response y, given the predictor(s) x. In LDA, instead, we model the distribution of the predictors x separately in each of the response classes, and then use Bayes' theorem to flip these around into estimates for $p(y = k|x)$. When these distributions are assumed to be normal, it turns out that the model is very similar in form

to logistic regression. Let π_k represent the overall or prior probability that a randomly chosen observation comes from the k-th class. Let $f_k(x) = p(x|y = k)$ denote the density function of x for an observation that comes from the k-th class. Then Bayes' theorem states that

$$p(y = k|x) = \frac{\pi_k f_k(x)}{\sum_{l=1}^{K} \pi_l f_l(x)} := p_k(x).$$

Estimating π_k is easy, we simply compute the fraction of the training observations that belong to the k-th class. However, estimating $f_k(x)$ tends to be more challenging, unless we assume some simple forms for these densities. Bayes classifier, which classifies an observation to the class for which $p_k(x)$ is largest, has the lowest possible error rate out of all classifiers.

Suppose there is one feature and $f_k(x)$ is Gaussian. Then

$$f_k(x) = \frac{1}{\sqrt{2\pi}\sigma_k} \exp(-\frac{1}{2\sigma_k^2}(x - \mu_k)^2),$$

where μ_k and σ_k are the mean and variance for the k-th class. Assume $\sigma_1^2 = \ldots = \sigma_k^2 = \sigma^2$. Plugging into the previous equation we get

$$p_k(x) = \frac{\pi_k \frac{1}{\sqrt{2\pi}\sigma} \exp(-\frac{1}{2\sigma^2}(x - \mu_k)^2)}{\sum_{l=1}^{K} \pi_l \frac{1}{\sqrt{2\pi}\sigma} \exp(-\frac{1}{2\sigma^2}(x - \mu_l)^2)}.$$

The Bayes classifier involves assigning an observation x to the class for which $p_k(x)$ is largest. Taking the log of $p_k(x)$ and rearranging the terms, it is equivalent to assigning the observation to the class for which

$$\delta_k(x) = x \cdot \frac{\mu_k}{\sigma^2} - \frac{\mu_k^2}{2\sigma^2} + \log(\pi_k)$$

is largest. In practice, we do not have π_k, μ_k, and σ^2 and we should approximate them using the data.

For multiple features, the Gaussian density is defined as

$$f(x) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\Big(-\frac{1}{2}(x - \mu)^T \Sigma^{-1}(x - \mu)\Big).$$

Similarly, the Bayes classifier assigns an observation x to the class for which

$$\delta_k(x) = x^T \Sigma^{-1}\mu_k - \frac{1}{2}\mu_k^T \Sigma^{-1}\mu_k + \log(\pi_k)$$

is largest. The word linear in the classifier's name stems from the fact that the discriminant functions $\delta_k(x)$ are linear functions of x.

Let assume that the variance among different classes differ. Then the Bayes classifier assigns an observation x to the class for which

$$\delta_k(x) = -\frac{1}{2}(x - \mu_k)^T \Sigma_k^{-1}(x - \mu_k) - \frac{1}{2}\log|\Sigma_k| + \log(\pi_k)$$

is largest. It is called Quadratic Discriminant Analysis (QDA).

6.1.1 Comparison

LDA is a much less flexible classifier than QDA, and so has substantially lower variance. This can potentially lead to improved prediction performance. But there is a trade-off: if LDA's assumption that the K classes share a common covariance matrix is badly off, then LDA can suffer from high bias. Roughly speaking, LDA tends to be a better bet than QDA if there are relatively few training observations and so reducing variance is crucial. In contrast, QDA is recommended if the training set is very large, so that the variance of the classifier is not a major concern, or if the assumption of a common covariance matrix for the K classes is clearly untenable.

LDA assumes that the observations are drawn from a Gaussian distribution with a common covariance matrix in each class, and so can provide some improvements over logistic regression when this assumption approximately holds. Conversely, logistic regression can outperform LDA if these Gaussian assumptions are not met.

KNN is a completely non-parametric approach: no assumptions are made about the shape of the decision boundary. Therefore, we can expect this approach to dominate LDA and logistic regression when the decision boundary is highly non-linear. On the other hand, KNN does not tell us which predictors are important.

Finally, QDA serves as a compromise between the non-parametric KNN method and the linear LDA and logistic regression approaches. Since QDA assumes a quadratic decision boundary, it can accurately model a wider range of problems than can the linear methods. Though not as flexible as KNN, QDA can perform better in the presence of a limited number of training observations because it does make some assumptions about the form of the decision boundary

No one method will dominate the others in every situation. When the true decision boundaries are linear, then the LDA and logistic regression approaches will tend to perform well.When the boundaries are moderately non-linear, QDA may give better results. Finally, for much more complicated decision boundaries, a non-parametric approach such as KNN can be superior. But the level of smoothness for a non-parametric approach must be chosen carefully.

6.2 Naive Bayes

In LDA, the feature vectors x were continuous, real-valued vectors, but in Naive Bayes the x_j's are discrete-valued.

Naive Bayes classifiers are a family of classifiers that are quite similar to the linear. However, they tend to be even faster in training. The price paid for this efficiency is that naive Bayes models often provide generalization performance

that is slightly worse than that of linear classifiers like logistic regression and linear support vector classifier.

The reason that naive Bayes models are so efficient is that they learn parameters by looking at each feature individually and collect simple per-class statistics from each feature. There are three kinds of naive Bayes classifiers implemented in *scikit-learn*:

- **GaussianNB:** GaussianNB can be applied to any continuous data.

```
from sklearn.naive_bayes import GaussianNB
X, y = dataset
clf = GaussianNB()
model = clf.fit(X, y)
```

  In Gaussian naive Bayes, we assume that the likelihood of the feature values, x, given an observation is of class y, follows a normal distribution:

$$p(x|y) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp(-\frac{(x-\mu)^2}{2\sigma^2}).$$

  One of the interesting aspects of naive Bayes classifiers is that they allow us to assign a prior belief over the respected target classes. We can do this using GaussianNB's pri ors parameter, which takes in a list of the probabilities assigned to each class of the target vector:

```
clf = GaussianNB(priors=[0.25, 0.25, 0.5])
model = clf.fit(features, target)
```

  If we do not add any argument to the priors parameter, the prior is adjusted based on the data.
- **BernoulliNB:** The Bernoulli naive Bayes classifier assumes that all our features are binary such that they take only two values.
- **MultinomialNB:** MultinomialNB assumes count data, that is, that each feature represents an integer count of something, like how often a word appears in a sentence.

To make a prediction, a data point is compared to the statistics for each of the classes, and the best matching class is predicted. Interestingly, for both MultinomialNB and BernoulliNB, this leads to a prediction formula that is of the same form as in the linear models.

Strengths, weaknesses, and parameters:

MultinomialNB and BernoulliNB have a single parameter, alpha, which controls model complexity. The way alpha works is that the algorithm adds to the data alpha many virtual data points that have positive values for all the features. This results in a "smoothing" of the statistics. A large alpha means more smoothing, resulting in less complex models. The algorithm's performance is relatively robust to the setting of alpha, meaning that setting alpha is not critical for good performance. However, tuning it usually improves

accuracy somewhat. GaussianNB is mostly used on very high-dimensional data, while the other two variants of naive Bayes are widely used for sparse count data such as text.

The naive Bayes models share many of the strengths and weaknesses of the linear models. They are very fast to train and to predict, and the training procedure is easy to understand. The models work very well with high-dimensional sparse data and are relatively robust to the parameters. Naive Bayes models are great baseline models and are often used on very large datasets, where training even a linear model might take too long.

Calibrating: Class probabilities are a common and useful part of machine learning models. In scikit-learn, most learning algortihms allow us to see the predicted probabilities of class membership using predict_proba. This can be extremely useful if, for instance, we want to only predict a certain class if the model predicts the probability that it is that class is over 90%. However, some models, including naive Bayes classifiers, output probabilities that are not based on the real world. That is, predict_proba might predict an observation has a 0.70 chance of being a certain class, when the reality is that it is 0.10 or 0.99. To obtain meaningful predicted probabilities we need conduct what is called calibration.

```
from sklearn.naive_bayes import GaussianNB
from sklearn.calibration import CalibratedClassifierCV
X, y = dataset
clf = GaussianNB()
clf_sigmoid = CalibratedClassifierCV(clf, cv=2, method=
    'sigmoid')
clf_sigmoid.fit(X,y)

classifer.fit(X, y).predict_proba(new_observation)
```

7

k-Nearest Neighbors

The k-NN algorithm is arguably the simplest machine learning algorithm. Building the model consists only of storing the training dataset. To make a prediction for a new data point, the algorithm finds the k closest data points in the training dataset.

7.1 k-Neighbors classification

In its simplest version, the k-NN algorithm only considers exactly one nearest neighbor, which is the closest training data point to the point we want to make a prediction for. The prediction is then simply the known output for this training point. Instead of considering only the closest neighbor, we can also consider an arbitrary number, k, of neighbors. When considering more than one neighbor, we use voting to assign a label. 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: the majority class among the k-nearest neighbors.

Given a positive integer K and a test observation x_0, the KNN classifier first identifies the neighbors K points in the training data that are closest to x_0, represented by N_0. It then estimates the conditional probability for class j as the fraction of points in N_0 whose response values equal j:

$$P(Y = j|X = x_0) = \frac{1}{K} \sum_{i \in N_0} I(y_i = j).$$

Finally, KNN applies Bayes rule and classifies the test observation x_0 to the class with the largest probability. Despite the fact that it is a very simple approach, KNN can often produce classifiers that are surprisingly close to the optimal Bayes classifier.

```
from sklearn.neighbors import KNeighborsClassifier
X, y = dataset
```

```
X_train, X_test, y_train, y_test = train_test_split(X,
    y, random_state=0)
clf = KNeighborsClassifier(n_neighbors=k)
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
print("Test set accuracy: {:.2f}".format(clf.score(
    X_test, y_test)))
```

Using few neighbors corresponds to high model complexity and using many neighbors corresponds to low model complexity. If you consider the extreme case where the number of neighbors is the number of all data points in the training set, each test point would have exactly the same neighbors and all predictions would be the same: the class that is most frequent in the training set.

When $K = 1$, the decision boundary is overly flexible and finds patterns in the data that don't correspond to the Bayes decision boundary. This corresponds to a classifier that has low bias but very high variance. As K grows, the method becomes less flexible and produces a decision boundary that is close to linear. This corresponds to a low-variance but high-bias classifier.

7.2 k-Neighbors regression

This is a regression variant of the k-nearest neighbors algorithm. We evaluate the model using the R^2 score. The R^2 score, also known as the coefficient of determination, is a measure of goodness of a prediction for a regression model, and yields a score between 0 and 1. 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.

```
from sklearn.neighbors import KNeighborsRegressor
X, y = dataset
X_train, X_test, y_train, y_test = train_test_split(X,
    y, random_state=0)
reg = KNeighborsRegressor(n_neighbors=k)
reg.fit(X_train, y_train)
y_pred = reg.predict(X_test)
print("Test set accuracy: {:.2f}".format(reg.score(
    X_test, y_test)))
```

Using only a single neighbor, each point in the training set has an obvious influence on the predictions, and the predicted values go through all of the data points. This leads to a very unsteady prediction. Considering more neighbors leads to smoother predictions, but these do not fit the training data as well.
Strengths, weaknesses, and parameters:

In principle, there are two important parameters to the KNeighbors classifier: the number of neighbors, k, and how you measure distance between data points. In practice, using a small number of neighbors like three or five often works well, but you should certainly adjust this parameter. By default, Euclidean distance is used as distance, which works well in many settings.

One of the strengths of k-NN is that the model is very easy to understand, and often gives reasonable performance without a lot of adjustments. Using this algorithm is a good baseline method to try before considering more advanced techniques. Building the nearest neighbors model is usually very fast, but when your training set is very large (either in number of features or in number of samples) prediction can be slow. When using the k-NN algorithm, it's important to preprocess your data. This approach often does not perform well on datasets with many features, and it does particularly badly with sparse datasets.

While the nearest k-neighbors algorithm is easy to understand, it is not often used in practice, due to prediction being slow and its inability to handle many features.

8

Neural Network

8.1 Model Representation

Let's examine how we will represent a hypothesis function using neural networks. At a very simple level, neurons are basically computational units that take inputs (dendrites) as electrical inputs (called "spikes") that are channeled to outputs (axons). In our model, our dendrites are like the input features $x_1, \ldots, x_n$, and the output is the result of our hypothesis function. In this model our x_0 input node is sometimes called the "bias unit." It is always equal to 1. In neural networks, we use the same logistic function as in classification, $\frac{1}{1+e^{-\theta^T x}}$, yet we sometimes call it a sigmoid (logistic) activation function. In this situation, our θ parameters are sometimes called "weights".

Visually, a simplistic representation looks like:

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \Rightarrow [\ \ ] \Rightarrow h_\theta(x).$$

Our input nodes (layer 1), also known as the "input layer", go into another node (layer 2), which finally outputs the hypothesis function, known as the "output layer".

We can have intermediate layers of nodes between the input and output layers called the "hidden layers."

In this example, we label these intermediate or "hidden" layer nodes $a_0^2, \ldots, a_n^2$ and call them "activation units."

$a_i^{(j)}$ = "activation" of unit i in layer j
$\Theta^{(j)}$ = matrix of weights controlling function mapping from layer j to layer $j+1$

If we had one hidden layer, it would look like:

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \Rightarrow \begin{bmatrix} a_1^{(2)} \\ a_2^{(2)} \\ a_3^{(2)} \end{bmatrix} \Rightarrow h_\theta(x).$$

The values for each of the "activation" nodes is obtained as follows:

$$\begin{aligned} a_1^{(2)} &= g(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3) \\ a_2^{(2)} &= g(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3) \\ a_3^{(2)} &= g(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3) \\ h_\Theta(x) &= a_1^{(3)} = g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)}) \end{aligned}$$

This is saying that we compute our activation nodes by using a 3×4 matrix of parameters. We apply each row of the parameters to our inputs to obtain the value for one activation node. Our hypothesis output is the logistic function applied to the sum of the values of our activation nodes, which have been multiplied by another parameter matrix $\Theta^{(2)}$ containing the weights for our second layer of nodes. Each layer gets its own matrix of weights, $\Theta^{(j)}$.

If network has s_j units in layer j and s_{j+1} units in layer $j+1$, then $\Theta^{(j)}$ will be of dimension $s_{j+1} \times (s_j + 1)$.

The $+1$ comes from the addition in $\Theta^{(j)}$ of the "bias nodes," x_0 and $\Theta_0^{(j)}$. In other words the output nodes will not include the bias nodes while the inputs will. The image 8.1 summarizes our model representation:

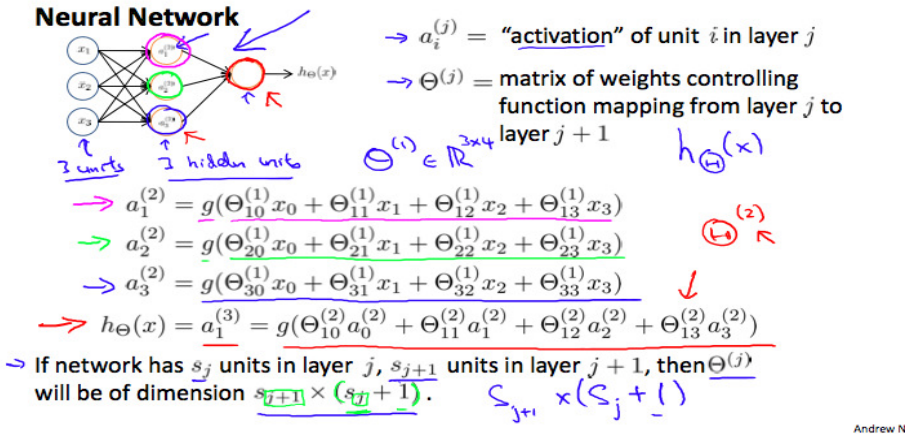


Fig. 8.1. Neural network weights

Vectorization: We are going to define a new variable $z_k^{(j)}$ that encompasses the parameters inside our g function. In our previous example if we replaced by the variable z for all the parameters we would get:

$$a_1^{(2)} = g(z_1^{(2)})$$
$$a_2^{(2)} = g(z_2^{(2)})$$
$$a_3^{(2)} = g(z_2^{(2)})$$

In other words, for layer $j = 2$ and node k, the variable z will be:

$$z_k^{(2)} = \Theta_{k,0}^{(1)} x_0 + \Theta_{k,1}^{(1)} x_1 + \ldots + \Theta_{k,n}^{(1)} x_n.$$

The vector representation of x and $z^{(j)}$ is:

$$x = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} \qquad z^{(j)} = \begin{bmatrix} z_1^{(j)} \\ z_2^{(j)} \\ \vdots \\ z_n^{(j)} \end{bmatrix}.$$

Setting $a^{(1)} = x$, we can rewrite the equation as:

$$z^{(j)} = \Theta^{(j-1)} a^{(j-1)},$$

and

$$a^{(j)} = g(z^{(j)}).$$

To compute our final hypothesis, compute:

$$z^{(j+1)} = \Theta^{(j)} a^{(j)}, \qquad h_\Theta(x) = a^{(j+1)} = g(z^{(j+1)}).$$

8.1.1 Cost Function

Let's first define a few variables that we will need to use:

L = total number of layers in the network
s_l = number of units (not counting bias unit) in layer l
K = number of output units/classes

Recall that in neural networks, we may have many output nodes. We denote $h_\Theta(x)_k$ as being a hypothesis that results in the k^{th} output. Our cost function for neural networks is going to be a generalization of the one we used for logistic regression, but more complicated.

$$\begin{aligned} J(\Theta) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} \left[y_k^{(i)} \log((h_\Theta(x^{(i)})_k) + (1 - y_k^{(i)}) \log(1 - (h_\Theta(x^{(i)})_k) \right] \\ + \frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (\Theta_{j,i}^{(l)})^2. \end{aligned} \tag{8.1}$$

We have added a few nested summations to account for our multiple output nodes. In the first part of the equation, before the square brackets, we have an additional nested summation that loops through the number of output nodes.

In the regularization part, after the square brackets, we must account for multiple theta matrices. The number of columns in our current Θ matrix is equal to the number of nodes in our current layer (including the bias unit). The number of rows in our current theta matrix is equal to the number of nodes in the next layer (excluding the bias unit). As before with logistic regression, we square every term.

8.1.2 Backpropagation Algorithm

"Backpropagation" is neural-network terminology for minimizing our cost function, just like what we were doing with gradient descent in logistic and linear regression. Our goal is to compute:

$$\min_{\Theta} J(\Theta).$$

That is, we want to minimize our cost function J using an optimal set of parameters in Θ. In this section we will look at the equations we use to compute the partial derivative of $J(\Theta)$:

$$\frac{\partial}{\partial \Theta_{i,j}^{(l)}} J(\Theta).$$

To do so, we use the algorithm in figure 8.2:

Implementation Note: Unrolling Parameters With neural networks, we are working with sets of matrices:

$$\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)} \dots \; ; \; D^{(1)}, D^{(2)}, D^{(3)} \dots .$$

In order to use optimizing functions such as "fminunc()", we will want to "unroll" all the elements and put them into one long vector:

```
thetaVector = [ Theta1(:); Theta2(:); Theta3(:) ];
deltaVector = [ D1(:); D2(:); D3(:) ];
```

If the dimensions of Theta1 is 10×11, Theta2 is 10×11 and Theta3 is 1×11, then we can get back our original matrices from the "unrolled" versions as follows:

```
Theta1 = reshape(thetaVector(1:110),10,11);
Theta2 = reshape(thetaVector(111:220),10,11);
Theta3 = reshape(thetaVector(221:231),1,11);
```

Gradient Checking: Gradient checking will assure that our backpropagation works as intended. We can approximate the derivative of our cost function with:

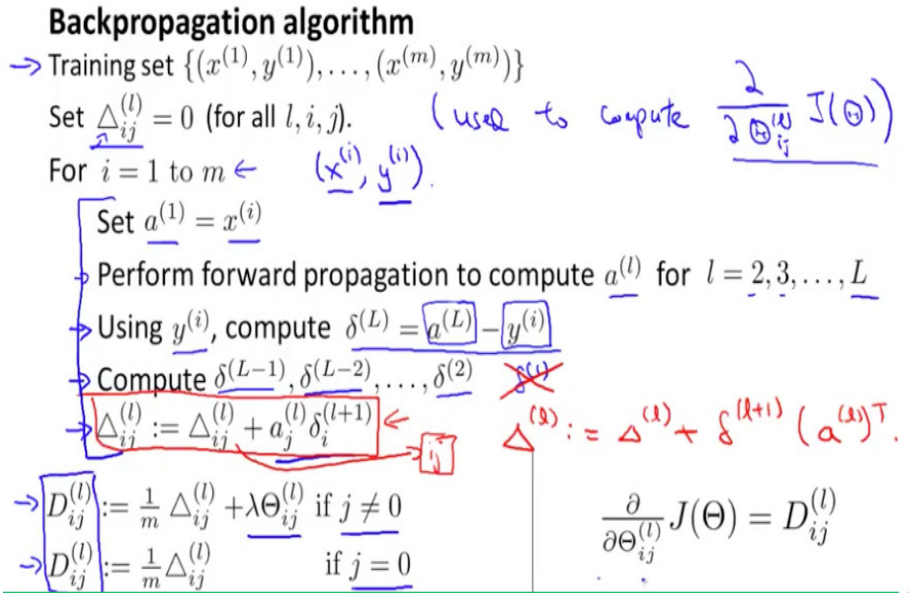


Fig. 8.2. Backpropagation Algorithm

$$\frac{\partial}{\partial \Theta} J(\Theta) \approx \frac{J(\Theta + \epsilon) - J(\Theta - \epsilon)}{2\epsilon}.$$

With multiple theta matrices, we can approximate the derivative with respect to Θ_j as follows:

$$\frac{\partial}{\partial \Theta_j} J(\Theta) \approx \frac{J(\Theta_1, \ldots, \Theta_j + \epsilon, \ldots, \Theta_n) - J(\Theta_1, \ldots, \Theta_j - \epsilon, \ldots, \Theta_n)}{2\epsilon}.$$

A small value for ϵ such as $\epsilon = 10^{-4}$, guarantees that the math works out properly. If the value for ϵ is too small, we can end up with numerical problems.

Hence, we are only adding or subtracting epsilon to the Θ_j matrix. In Matlab we can do it as follows:

Random Initialization: Initializing all theta weights to zero does not work with neural networks. When we backpropagate, all nodes will update to the same value repeatedly. Instead we can randomly initialize our weights for our Θ matrices. We initialize each $\Theta_{ij}^{(l)}$ to a random value between $[-\epsilon, \epsilon]$.

Putting it Together: First, pick a network architecture; choose the layout of your neural network, including how many hidden units in each layer and how many layers in total you want to have.

- Number of input units = dimension of features $x^{(i)}$
- Number of output units = number of classes

- Number of hidden units per layer = usually more the better (must balance with cost of computation as it increases with more hidden units)
- Defaults: 1 hidden layer. If you have more than 1 hidden layer, then it is recommended that you have the same number of units in every hidden layer.

Training a Neural Network:

1. Randomly initialize the weights
2. Implement forward propagation to get $h_\Theta(x^{(i)})$ for any $x^{(i)}$
3. Implement the cost function
4. Implement backpropagation to compute partial derivatives
5. Use gradient checking to confirm that your backpropagation works. Then disable gradient checking.
6. Use gradient descent or a built-in optimization function to minimize the cost function with the weights in theta.

8.2 Training Neural Network

Neural networks consist of layers of units. To construct a feedforward neural network in Keras, we need to make a number of choices about both the network architecture and training process.

- First, for each layer in the hidden and output layers we must define the number of units to include in the layer and the activation function.
- Second, we need to define the number of hidden layers to use in the network. More layers allow the network to learn more complex relationships, but with a computational cost.
- Third, we have to define the structure of the activation function (if any) of the output layer. The nature of the output function is often determined by the goal of the network. Here are some common output layer patterns:
  - Binary classification: One unit with a sigmoid activation function.
  - Multiclass classification: k units (where k is the number of target classes) and a softmax activation function.
  - Regression: One unit with no activation function.
- Fourth, we need to define a loss function: Binary cross-entropy, Categorical cross-entropy, and Mean square error.
- Fifth, we have to define an optimizer.
- Sixth, we can select one or more metrics to use to evaluate the performance.

```
from keras import models, layers
np.random.seed(0)
number_of_features = 1000
(X_train, y_train), (X_test, y_test) = datasets
network = models.Sequential()
```

```
network.add(layers.Dense(units=16, activation="relu",
    input_shape=(number_of_features,)))
network.add(layers.Dense(units=16, activation="relu"))
network.add(layers.Dense(units=1,activation="sigmoid"))
network.compile(loss="binary_crossentropy", optimizer="
    rmsprop", metrics=["accuracy"])
history = network.fit(X_train, y_train, epochs=3,
    verbose=1, batch_size=100, validation_data=(X_test,
    y_test))
predicted_y = network.predict(X_test)
```

In case of multi-class classification, the activation function of the last layer will change to "softmax" and loss in compile will be "categorical_crossentropy". In regression there is NO activation function in the last layer and loss in compile will be "mse".

8.2.1 Reducing overfitting

You want to reduce overfitting.

- weight regularization: One strategy to combat overfitting neural networks is by penalizing the parameters of the neural network. This method is called weight regularization or weight decay. In Keras, we can add a weight regularization by including using kernel_regular izer=regularizers.l2(0.01) in a layer's parameters. In this example, 0.01 determines how much we penalize higher parameter values.
- Early Stopping: Typically in the first training epochs both the training and test errors will decrease, but at some point the network will start "memorizing" the training data, causing the training error to continue to decrease even while the test error starts increasing. Because of this phenomenon, one of the most common and very effective methods to counter overfitting is to monitor the training process and stop training when the test error starts to increase. This strategy is called early stopping.
  We can implement early stopping as a callback function. Callbacks are functions that can be applied at certain stages of the training process, such as at the end of each epoch. Specifically, we include EarlyStopping(monitor='val_loss', patience=2) to define that we want to monitor the test loss at each epoch and after the test loss has not improved after two epochs, training is interrupted. However, since we set patience=2, we won't get the best model, but the model two epochs after the best model. Therefore, optionally, we can include a second operation, ModelCheckpoint, which saves the model to a file after every checkpoint. It would be helpful for us, if we set save_best_only=True, because then ModelCheckpoint will only save the best model

```
from keras.callbacks import EarlyStopping,
    ModelCheckpoint
network.compile(loss="binary_crossentropy", optimizer
    ="rmsprop", metrics=["accuracy"])
callbacks = [EarlyStopping(monitor="val_loss",
    patience=2), ModelCheckpoint(filepath="best_model.
    h5", monitor="val_loss", save_best_only=True)]
history = network.fit(X_train, y_train, epochs=20,
    callbacks=callbacks, verbose=0, batch_size=100,
    validation_data=(fX_test, y_test))
```

- Dropout: Dropout is a popular and powerful method for regularizing neural networks. In dropout, every time a batch of observations is created for training, a proportion of the units in one or more layers is multiplied by zero (i.e., dropped). In this setting, every batch is trained on the same network, but each batch is confronted by a slightly different version of that network's architecture.
  Dropout is effective because by constantly and randomly dropping units in each batch, it forces units to learn parameter values able to perform under a wide variety of network architectures. That is, they learn to be robust to disruptions in the other hidden units, and this prevents the network from simply memorizing the training data.
  It is possible to add dropout to both the hidden and input layers. A common choice for the portion of units to drop is 0.2 for input units and 0.5 for hidden units. We can implement dropout by adding Dropout layers into our network architecture.

```
# Add a dropout layer for input layer
network.add(layers.Dropout(0.2, input_shape=(
    number_of_features,)))
network.add(layers.Dropout(0.5))
```

8.2.2 Classifying Images

You want to classify images using a convolutional neural network.

```
import numpy as np
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers.convolutional import Conv2D,
    MaxPooling2D
from keras.utils import np_utils
from keras import backend as K
# Set that the color channel value will be first
```

```
K.set_image_data_format("channels_first")
np.random.seed(0)
channels = 1
height = 28
width = 28
(X_train, y_train), (X_test, y_test) = data
X_train = X_train.reshape(X_train.shape[0], channels,
    height, width)
X_test = X_test.reshape(X_test.shape[0], channels,
    height, width)
features_train = X_train / 255
features_test = X_test / 255
target_train = np_utils.to_categorical(y_train)
target_test = np_utils.to_categorical(y_test)
number_of_classes = target_test.shape[1]
network = Sequential()
network.add(Conv2D(filters=64, kernel_size=(5, 5),
    input_shape=(channels, width, height), activation='
    relu'))
network.add(MaxPooling2D(pool_size=(2, 2)))
network.add(Dropout(0.5))
network.add(Flatten())
network.add(Dense(128, activation="relu"))
network.add(Dropout(0.5))
network.add(Dense(number_of_classes, activation="
    softmax"))
network.compile(loss="categorical_crossentropy",
    optimizer="rmsprop", metrics=["accuracy"])
network.fit(features_train, target_train, epochs=2,
    verbose=0, batch_size=1000, validation_data=(
    features_test, target_test))
```

8.3 Multilayer Perceptrons

Multilayer perceptrons (MLPs) are also known as (vanilla) feed-forward neural networks, or sometimes just neural networks. MLPs can be viewed as generalizations of linear models that perform multiple stages of processing to come to a decision.

In an MLP this process of computing weighted sums is repeated multiple times, first computing hidden units that represent an intermediate processing step, which are again combined using weighted sums to yield the final result.

Computing a series of weighted sums is mathematically the same as computing just one weighted sum, so to make this model truly more powerful than

a linear model, we need one extra trick. After computing a weighted sum for each hidden unit, a nonlinear function is applied to the result—usually the rectifying nonlinearity (relu) or the tangent hyperbolic. The result of this function is then used in the weighted sum that computes the output, $\hat{y}$. Either nonlinear function allows the neural network to learn much more complicated functions than a linear model could.

There are many ways to control the complexity of a neural network: the number of hidden layers, the number of units in each hidden layer, and the regularization (alpha).

An important property of neural networks is that their weights are set randomly before learning is started, and this random initialization affects the model that is learned.

```
from sklearn.neural_network import MLPClassifier
X, y = dataset
X_train, X_test, y_train, y_test = train_test_split(X,
    y, stratify=y, random_state=42)
mlp = MLPClassifier(algorithm='l-bfgs', random_state=0,
      hidden_layer_sizes=[10, 10], activation='tanh',
    alpha=.3)
mlp.fit(X_train, y_train)
print("Accuracy on training set: {:.2f}".format(mlp.
    score(X_train, y_train)))
print("Accuracy on test set: {:.2f}".format(mlp.score(
    X_test, y_test)))
```

Strengths, weaknesses, and parameters:

One of their main advantages is that they are able to capture information contained in large amounts of data and build incredibly complex models. Given enough computation time, data, and careful tuning of the parameters, neural networks often beat other machine learning algorithms.

Neural networks, particularly the large and powerful ones, often take a long time to train. They also require careful preprocessing of the data. Similarly to SVMs, they work best with "homogeneous" data, where all the features have similar meanings. Tuning neural network parameters is also an art unto itself.

The most important parameters are the number of layers and the number of hidden units per layer. You should start with one or two hidden layers, and possibly expand from there. The number of nodes per hidden layer is often similar to the number of input features, but rarely higher than in the low to mid-thousands.

A common way to adjust parameters in a neural network is to first create a network that is large enough to overfit, making sure that the task can actually be learned by the network. Then, once you know the training data can be learned, either shrink the network or increase alpha to add regularization, which will improve generalization performance.

9

Evaluating a Learning Algorithm

Debugging a Learning Algorithm: Suppose you have implemented regularized linear regression. However, when you test hypothesis on a new set of data you find that it makes unacceptably large error in its prediction. What should you try next?

- Getting more training examples
- Trying smaller sets of features
- Trying additional features
- Trying polynomial features
- Increasing/decreasing λ

Machine Learning Diagnostic: A test that you can run to gain insight what is/isnot working with a learning algorithm, and gain guidance as to how best to improve its performance.

Diagnostic can take time to implement, but doing so can be very good use of time. It can sometimes rule out certain courses of action as being unlikely to improve its performance.

9.1 Evaluating a Hypothesis

Once we have done some trouble shooting for errors in our predictions by:

- Getting more training examples
- Trying smaller sets of features
- Trying additional features
- Trying polynomial features
- Increasing or decreasing λ

We can move on to evaluate our new hypothesis. A hypothesis may have a low error for the training examples but still be inaccurate (because of overfitting). Thus, to evaluate a hypothesis, given a dataset of training examples, we can split up the data into two sets: a training set and a test set. Typically, the

training set consists of 70% of your data and the test set is the remaining 30%.

The new procedure using these two sets is then:

1 Learn Θ and minimize $J_{train}(\Theta)$ using the training set
2 Compute the test set error $J_{test}(\Theta)$.

The test set error:

- For linear regression: $J_{test}(\Theta) = \frac{1}{2m_{test}} \sum_{i=1}^{m_{test}} (h_\Theta(x_{test}^{(i)}) - y_{test}^{(i)})^2$.
- For classification/misclassification error (0/1 misclassification error):

$$err(h_\Theta(x), y) = \begin{cases} 1 & \text{if } h_\Theta(x) \geq 0.5 \text{ and } y = 0 \text{ or } h_\Theta(x) < 0.5 \text{ and } y = 1, \\ 0, & otherwise. \end{cases}$$

This gives us a binary 0 or 1 error result based on a misclassification. The average test error for the test set is:

$$\text{Test Error } = \frac{1}{m_{test}} \sum_{i=1}^{m_{test}} err(h_\Theta(x^{(i)}), y^{(i)}).$$

This gives us the proportion of the test data that was misclassified.

9.1.1 Model Selection and Train/Validation/Test Sets

Just because a learning algorithm fits a training set well, that does not mean it is a good hypothesis. It could over fit and as a result your predictions on the test set would be poor. The error of your hypothesis as measured on the data set with which you trained the parameters will be lower than the error on any other data set.

Given many models with different polynomial degrees, we can use a systematic approach to identify the 'best' function. In order to choose the model of your hypothesis, you can test each degree of polynomial and look at the error result.

One way to break down our dataset into the three sets is:

- Training set: 60%
- Cross validation set: 20%
- Test set: 20%

We can now calculate three separate error values for the three different sets using the following method:

1 Optimize the parameters in Θ using the training set for each polynomial degree.

- Find the polynomial degree d with the least error using the cross validation set.
- Estimate the generalization error using the test set with $J_{test}(\Theta^{(d)})$ (d = theta from polynomial with lower error);

This way, the degree of the polynomial d has not been trained using the test set.

The training set is used to fit the models; the validation set is used to estimate prediction error for model selection; the test set is used for assessment of the generalization error of the final chosen model. Ideally, the test set should be kept in a "vault," and be brought out only at the end of the data analysis. The cross validation set is used to help detect over-fitting and to assist in hyper-parameter search. The test set is used to measure the performance of the model.

9.1.2 Diagnosing Bias versus Variance

In this section we examine the relationship between the degree of the polynomial d and the underfitting or overfitting of our hypothesis. We need to distinguish whether bias or variance is the problem contributing to bad predictions. High bias is underfitting and high variance is overfitting. Ideally, we need to find a golden mean between these two. The training error will tend to decrease as we increase the degree d of the polynomial. At the same time, the cross validation error will tend to decrease as we increase d up to a point, and then it will increase as d is increased, forming a convex curve.

- High bias (underfitting): both $J_{train}(\Theta)$ and $J_{CV}(\Theta)$ will be high. Also, $J_{CV}(\Theta) \approx J_{train}(\Theta)$.
- High variance (overfitting): $J_{train}(\Theta)$ will be low and $J_{CV}(\Theta)$ will be much greater than $J_{train}(\Theta)$.

The is summarized in the figure 9.1:

9.1.3 Regularization and Bias/Variance

In order to choose the model and the regularization term λ, we need to:

1. Create a list of lambdas (i.e. $\lambda \in \{0, 0.01, 0.02, 0.04, 0.08, 0.16, 0.32, 0.64, \ldots\}$);
2. Create a set of models with different degrees or any other variants.
3. Iterate through the λ's and for each λ go through all the models to learn some Θ.
4. Compute the cross validation error using the learned Θ (computed with λ) on the $J_{CV}(\Theta)$ without regularization or $\lambda = 0$.
5. Select the best combo that produces the lowest error on the cross validation set.
6. Using the best combo Θ and λ, apply it on $J_{test}(\Theta)$ to see if it has a good generalization of the problem.

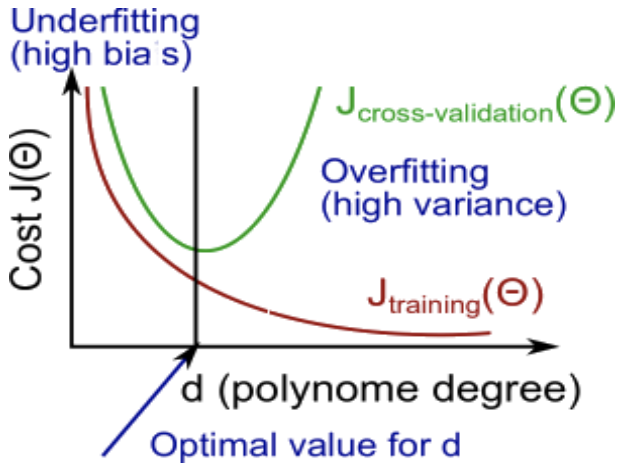


Fig. 9.1. Bias vs Variance

9.1.4 Learning Curves

Training an algorithm on a very few number of data points (such as $1, 2$ or 3) will easily have 0 errors because we can always find a quadratic curve that touches exactly those number of points. Hence:

- As the training set gets larger, the error for a quadratic function increases.
- The error value will plateau out after a certain m, or training set size.

Experiencing high bias:

- *Low training set size:* causes $J_{train}(\Theta)$ to be low and $J_{CV}(\Theta)$ to be high.
- *Large training set size:* causes both $J_{train}(\Theta)$ and $J_{CV}(\Theta)$ to be high with $J_{train}(\Theta) \approx J_{CV}(\Theta)$.

If a learning algorithm is suffering from high bias, getting more training data will not (by itself) help much.

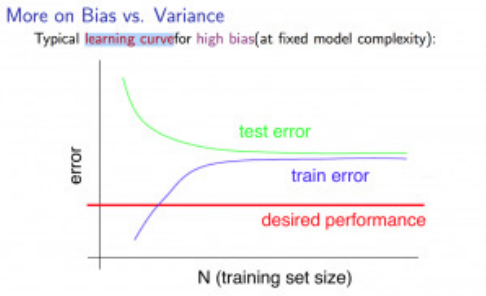


Fig. 9.2. Bias vs Variance

Experiencing high variance:

- *Low training set size:* $J_{train}(\Theta)$ will be low and $J_{CV}(\Theta)$ will be high.
- *Large training set size:* $J_{train}(\Theta)$ increases with training set size and $J_{CV}(\Theta)$ continues to decrease without leveling off. Also, $J_{train}(\Theta) < J_{CV}(\Theta)$ but the difference between them remains significant.

If a learning algorithm is suffering from high variance, getting more training data is likely to help.

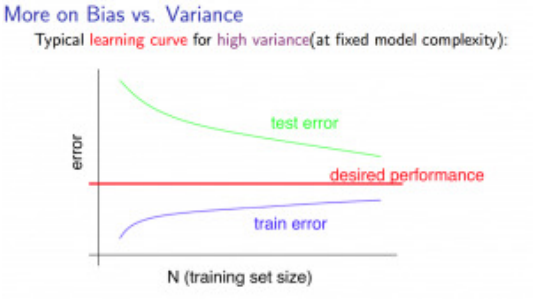


Fig. 9.3. Bias vs Variance

9.1.5 Deciding What to Do Next Revisited

Our decision process can be broken down as follows:

- Getting more training examples ⇒ fixes high variance
- Trying smaller sets of features ⇒ fixes high variance
- Trying additional features ⇒ fixes high bias
- Trying polynomial features ⇒ fixes high bias
- Increasing/decreasing λ ⇒ fixes high variance/bias.

Diagnosing Neural Networks:

- A neural network with fewer parameters is prone to underfitting. It is also computationally cheaper.
- A large neural network with more parameters is prone to overfitting. It is also computationally expensive. In this case you can use regularization (increase λ) to address the overfitting.

Using a single hidden layer is a good starting default. You can train your neural network on a number of hidden layers using your cross validation set. You can then select the one that performs best.

Model Complexity Effects:

- Low-order polynomials (low model complexity) have high bias and low variance. In this case, the model fits poorly consistently.

- Higher-order polynomials (high model complexity) fit the training data extremely well and the test data extremely poorly. These have low bias on the training data, but very high variance.
- In reality, we would want to choose a model somewhere in between, that can generalize well but also fits the data reasonably well.

9.1.6 Building a Spam Classifier

Given a data set of emails, we could construct a vector for each email. Each entry in this vector represents a word. The vector normally contains $10,000$ to $50,000$ entries gathered by finding the most frequently used words in our data set. If a word is to be found in the email, we would assign its respective entry a 1, else if it is not found, that entry would be a 0. Once we have all our x vectors ready, we train our algorithm and finally, we could use it to classify if an email is a spam or not.

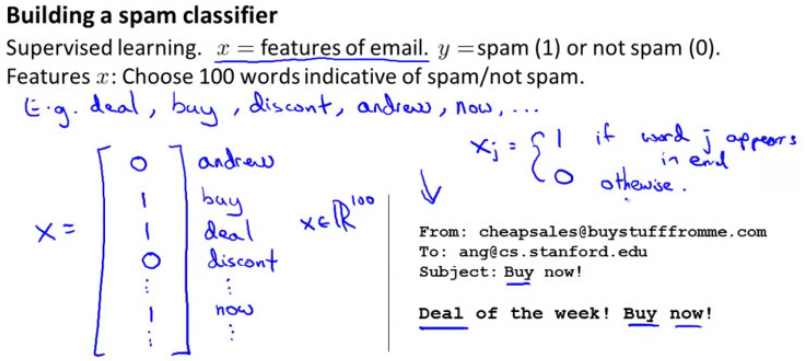


Fig. 9.4. Spam classifier

So how could you spend your time to improve the accuracy of this classifier?

- Collect lots of data (for example "honeypot" project but does not always work)
- Develop sophisticated features (for example: using email header data in spam emails)
- Develop algorithms to process your input in different ways (recognizing misspellings in spam).

It is difficult to tell which of the options will be most helpful.

9.1.7 Error Analysis

The recommended approach to solving machine learning problems is to:

- Start with a simple algorithm, implement it quickly, and test it early on your cross validation data.
- Plot learning curves to decide if more data, more features, etc. are likely to help.
- Manually examine the errors on examples in the cross validation set and try to spot a trend where most of the errors were made.

For example, assume that we have 500 emails and our algorithm misclassifies a 100 of them. We could manually analyze the 100 emails and categorize them based on what type of emails they are. We could then try to come up with new cues and features that would help us classify these 100 emails correctly. Hence, if most of our misclassified emails are those which try to steal passwords, then we could find some features that are particular to those emails and add them to our model. We could also see how classifying each word according to its root changes our error rate. Should discount/discounted/discounts/discounting be treated as the same word?

It is very important to get error results as a single, numerical value. Otherwise it is difficult to assess your algorithm's performance. For example if we use stemming, which is the process of treating the same word with different forms (fail/failing/failed) as one word (fail), and get a 3% error rate instead of 5%, then we should definitely add it to our model. However, if we try to distinguish between upper case and lower case letters and end up getting a 3.2% error rate instead of 3%, then we should avoid using this new feature. Hence, we should try new things, get a numerical value for our error rate, and based on our result decide whether we want to keep the new feature or not.

9.2 Model Evaluation

To evaluate supervised models, we split our dataset into a training set and a test set using the *train_test_split* function, built a model on the training set by calling the *fit* method, and evaluate it on the test set using the *score* method.

The reason we split our data into training and test sets is that we are interested in measuring how well our model generalizes to new, previously unseen data. We will expand on two aspects of this evaluation. We will first introduce cross-validation, a more robust way to assess generalization performance, and discuss methods to evaluate classification and regression performance. We will also discuss grid search, an effective method for adjusting the parameters in supervised models for the best generalization performance.

Cross-Validation

Cross-validation is a statistical method of evaluating generalization performance that is more stable and thorough than using a split into a training and a 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. When performing 5-fold cross-validation, for example, 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. The model is built using the data in folds 2–5, and then the accuracy is evaluated on fold 1. 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. For each of these five splits of the data into training and test sets, we compute the accuracy. A common way to summarize the cross-validation accuracy is to compute the mean.

It is important to keep in mind that cross-validation is not a way to build a model that can be applied to new data. Cross-validation does not return a model. The purpose of cross-validation is only to evaluate how well a given algorithm will generalize when trained on a specific dataset.

```
from sklearn.model_selection import cross_val_score
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression
iris = load_iris()
logreg = LogisticRegression()
scores = cross_val_score(logreg, iris.data, iris.target,
    cv=5)
print("Cross-validation scores: {}".format(scores))
```

Having multiple splits of the data also provides some information about how sensitive our model is to the selection of the training dataset. The main disadvantage of cross-validation is increased computational cost.

Splitting the dataset into k folds by starting with the first one-k-th part of the data, might not always be a good idea. It might contains only one class. Instead, we use stratified cross-validation. In stratified cross-validation, we split the data such that the proportions between classes are the same in each fold as they are in the whole dataset. It is usually a good idea to use stratified k-fold cross-validation instead of k-fold cross-validation to evaluate a classifier, because it results in more reliable estimates of generalization performance.

```
from sklearn.model_selection import KFold
kfold = KFold(n_splits=3, shuffle=True, random_state=0)
scores = cross_val_score(logreg, iris.data, iris.target
    , cv=kfold)
```

Leave-one-out cross-validation

```
from sklearn.model_selection import LeaveOneOut
loo = LeaveOneOut()
scores = cross_val_score(logreg, iris.data, iris.target
    , cv=loo)
```

```
print("Mean accuracy: {:.2f}".format(scores.mean()))
```

Grid search

Finding the values of the important parameters of a model (the ones that provide the best generalization performance) is a tricky task, but necessary for almost all models and datasets. The most commonly used method is grid search, which basically means trying all possible combinations of the parameters of interest.

Because grid search with cross-validation is such a commonly used method to adjust parameters, scikit-learn provides the GridSearchCV class, which implements it in the form of an estimator. To use the GridSearchCV class, you first need to specify the parameters you want to search over using a dictionary. GridSearchCV will then perform all the necessary model fits. The keys of the dictionary are the names of parameters we want to adjust (as given when constructing the model), and the values are the parameter settings we want to try out.

```
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
param_grid = {'C': [0.001, 0.01, 0.1], 'gamma': [ 0.01,
    0.1]}
grid_search = GridSearchCV(SVC(), param_grid, cv=5)
X, y = dataset
X_train, X_test, y_train, y_test = train_test_split(X,
    y, random_state=0)
grid_search.fit(X_train, y_train)
print("Test set score: {:.2f}".format(grid_search.score
    (X_test, y_test)))
print("Best parameters: {}".format(grid_search.
    best_params_))
print("Best cross-validation score: {:.2f}".format(
    grid_search.best_score_))
```

9.3 Evaluation Metrics and Scoring

In the regression setting, the most commonly-used measure is the mean squared error (MSE), given by

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{f}(x_i))^2,$$

or R^2 given by

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{f}(x_i))^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2},$$

which measures the amount of variance in the target vector that is explained by the model. The closer to 1, the better model. In classification case, we compute accuracy

$$\frac{1}{n}\sum_{i=1}^{n} I(y_i \neq \hat{y}_i),$$

which is the proportion of mistakes that male. Here I is the indicator function.

However, these are not the only way to summarize how well a supervised model performs on a given dataset. In practice, these evaluation metrics might not be appropriate for your application, and it is important to choose the right metric when selecting between models and adjusting parameters.

Confusion matrices

One of the most comprehensive ways to represent the result of evaluating binary classification is using confusion matrices

Table 9.1. Confusion matrix

Negative class	TN	FP
Positive class	FN	TP
	predicted negative	predicted positive

Accuracy

Accuracy is computed as

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

In other words, accuracy is the number of correct predictions divided by the number of all samples.

Precision

Precision measures how many of the samples predicted as positive are actually positive:

$$\text{Precision} = \frac{\text{TP}}{\text{TP+FP}}.$$

Precision is used as a performance metric when the goal is to limit the number of false positives.

Recall

Recall, on the other hand, measures how many of the positive samples are captured by the positive predictions:

$$\text{Recall} = \frac{\text{TP}}{\text{TP+FN}}.$$

Recall is used as performance metric when we need to identify all positive samples; that is, when it is important to avoid false negatives.

F1-score

F1-score is the harmonic mean of precision and recall:

$$\text{F1-score} = 2\frac{\text{precision * Recall}}{\text{Precision+Recall}}.$$

As it takes precision and recall into account, it can be a better measure than accuracy on imbalanced binary classification datasets.

```
from sklearn.metrics import confusion_matrix
confusion = confusion_matrix(y_test, y_pred)
print("Confusion matrix:\n{}".format(confusion))

from sklearn.metrics import f1_score
print("f1 score most frequent: {:.2f}".format(f1_score(
    y_test, y_pred)))

print(classification_report(y_test, y_pred,
    target_names=["negative", "positive"]))
```

Precision-recall curves and ROC curves

Changing the threshold that is used to make a classification decision in a model is a way to adjust the trade-off of precision and recall for a given classifier. Maybe you want to miss less than 10% of positive samples, meaning a desired recall of 90%. This decision depends on the application, and it should be driven by business goals. Setting a requirement on a classifier like 90% recall is often called setting the operating point. Fixing an operating point is often helpful in business settings to make performance guarantees to customers. Often, when developing a new model, it is not entirely clear what the operating point will be. For this reason, and to understand a modeling problem better, it is instructive to look at all possible thresholds, or all possible trade-offs of precision and recalls at once. This is possible using a tool called the precision-recall curve.

```
from sklearn.metrics import precision_recall_curve
X, y = dataset
X_train, X_test, y_train, y_test = train_test_split(X,
    y, random_state=0)
svc = SVC(gamma=.05).fit(X_train, y_train)
precision, recall, thresholds = precision_recall_curve(
    y_test, svc.decision_function(X_test))

# find threshold closest to zero
close_zero = np.argmin(np.abs(thresholds))
plt.plot(precision[close_zero], recall[close_zero], 'o'
    , markersize=10,label="threshold zero", fillstyle="
    none", c='k', mew=2)
plt.plot(precision, recall, label="precision recall
    curve")
plt.xlabel("Precision")
plt.ylabel("Recall")
```

The closer a curve stays to the upper-right corner, the better the classifier. A point at the upper right means high precision and high recall for the same threshold.

Receiver operating characteristics (ROC) and AUC

Another tool to analyze the behavior of classifiers at different thresholds is the receiver operating characteristics curve, or ROC curve. Similar to the precision-recall curve, the ROC curve considers all possible thresholds for a given classifier, but it shows the false positive rate (FPR) against the true positive rate (TPR). Recall that the true positive rate is recall, while the false positive rate is the fraction of false positives out of all negative samples:

$$\text{FPR} = \frac{\text{FP}}{\text{TN+FP}}.$$

```
from sklearn.metrics import roc_curve
fpr, tpr, thresholds = roc_curve(y_test, svc.
    decision_function(X_test))
plt.plot(fpr, tpr, label="ROC Curve")
plt.xlabel("FPR")
plt.ylabel("TPR (recall)")

close_zero = np.argmin(np.abs(thresholds))
plt.plot(fpr[close_zero], tpr[close_zero], 'o',
    markersize=10,label="threshold zero", fillstyle="
    none", c='k', mew=2)
```

```
plt.legend(loc=4)
```

For the ROC curve, the ideal curve is close to the top left: you want a classifier that produces a high recall while keeping a low false positive rate.

we often want to summarize the ROC curve using a single number, the area under the curve. We can compute the area under the ROC curve using the roc_auc_score function.

```
from sklearn.metrics import roc_auc_score
rf_auc = roc_auc_score(y_test, rf.predict_proba(X_test)
    [:, 1])
svc_auc = roc_auc_score(y_test, svc.decision_function(
    X_test))
print("AUC for Random Forest: {:.3f}".format(rf_auc))
print("AUC for SVC: {:.3f}".format(svc_auc))
```

Recall that because average precision is the area under a curve that goes from 0 to 1, average precision always returns a value between 0 (worst) and 1 (best). Predicting randomly always produces an AUC of 0.5, no matter how imbalanced the classes in a dataset are. This makes AUC a much better metric for imbalanced classification problems than accuracy. The AUC can be interpreted as evaluating the ranking of positive samples. It's equivalent to the probability that a randomly picked point of the positive class will have a higher score according to the classifier than a randomly picked point from the negative class. So, a perfect AUC of 1 means that all positive points have a higher score than all negative points. For classification problems with imbalanced classes, using AUC for model selection is often much more meaningful than using accuracy.

10

Support Vector Machines

Another powerful and widely used learning algorithm is the support vector machine (SVM). Using the regression algorithm, we minimized misclassification errors. However, in SVMs, our optimization objective is to maximize the margin. The margin is defined as the distance between the separating hyperplane (decision boundary) and the training samples that are closest to this hyperplane, which are the so-called support vectors.

10.1 Introduction

In logistic regression, the probability $p(y = 1|x; \theta)$ is modeled by $h_\theta(x) = g(\theta^T x)$. We predict "1" on an input x if and only if $h_\theta(x) \geq 0.5$, or if and only if $\theta^T x \geq 0$. Consider a positive training example ($y = 1$). The larger $\theta^T x$ is, the larger also is $p(y = 1|x; \theta)$, and thus also the higher our degree of confidence that the label is 1. Thus, informally we can think of our prediction as being a very confident one that $y = 1$ if $\theta^T x \gg 0$. Similarly, we think of logistic regression as making a very confident prediction of $y = 0$, if $\theta^T x \ll 0$. Given a training set, informally it seems that we'd have found a good fit to the training data if we can find θ so that $\theta^T x^{(i)} \gg 0$ whenever $y^{(i)} = 1$ and $\theta^T x^{(i)} \ll 0$ whenever $y^{(i)} = 0$, since this would reflect a very confident and correct set of classifications for all the training examples.

Let $y \in \{1, -1\}$ and $h_{w,b} = g(w^T x + b)$, with

$$g(z) = \begin{cases} 1 & \text{if } z \geq 0, \\ -1 & \text{otherwise.} \end{cases}$$

The functional margin of $(w; b)$ with respect to the training example $(x^{(i)}, y^{(i)})$ is defined as

$$\hat{\gamma}^{(i)} = y^{(i)}(w^T x^{(i)} + b). \tag{10.1}$$

Note that if $y^{(i)} = 1$, then for the functional margin to be large then we need $w^T x + b$ to be a large positive number. Conversely, if $y^{(i)} = -1$, then for the

functional margin to be large, then we need $w^T x + b$ to be a large negative number. Moreover, if $y^{(i)}(w^T x+b) > 0$, then our prediction on this example is correct. Hence, a large functional margin represents a confident and a correct prediction.

We define the geometric margin of $(w; b)$ with respect to a training example $(x^{(i)}; y^{(i)})$ to be

$$\gamma^{(i)} = y^{(i)} \left(\left(\frac{w}{\|w\|^2} \right)^T x^{(i)} + \frac{b}{\|w\|^2} \right). \tag{10.2}$$

Note that if $\|w\| = 1$, then the functional margin equals the geometric margin. Also, the geometric margin is invariant to rescaling of the parameters.

Given a training set $S = \{(x^{(i)},\ y^{(i)}); i = 1, \cdots, m\}$, we define the function margin and geometric margin of $(w; b)$ with respect to S as

$$\hat{\gamma} = \min_i \hat{\gamma}^{(i)}, \qquad \gamma = \min_i \gamma^{(i)}.$$

Given a training set, we try to find a decision boundary that maximizes the geometric margin. Suppose that we are given a training set that is linearly separable. To find the maximum geometric margin, we pose the following optimization problem:

$$\begin{aligned} \max_{\gamma,w,b} \quad & \gamma \\ s.t. \quad & y^{(i)}(w^T x^{(i)} + b) \geq \gamma, \quad 1, \ldots, m \\ & \|w\| = 1. \end{aligned} \tag{10.3}$$

We want to maximize γ, subject to each training example having functional margin at least γ. The $\|w\| = 1$ constraint ensures that the functional margin equals to the geometric margin. But the constraint $\|w\| = 1$ is non-convex and makes the problem difficult to solve. Hence we consider

$$\begin{aligned} \max_{\hat{\gamma},w,b} \quad & \frac{\hat{\gamma}}{\|w\|} \\ s.t. \quad & y^{(i)}(w^T x^{(i)} + b) \geq \hat{\gamma}, \quad 1, \ldots, m. \end{aligned} \tag{10.4}$$

Here, the objective is non-convex and it is hard to solve this problem. We introduce the scaling constraint that the functional margin of $(w; b)$ with respect to the training set must be 1, i.e., $\hat{\gamma} = 1$. Since multiplying w and b by some constant results in the functional margin being multiplied by that same constant, this is indeed a scaling constraint, and can be satisfied by rescaling $w; b$. Plugging this into our problem above, we have the following optimization problem

$$\begin{aligned} \min_{\hat{\gamma},w,b} \quad & \frac{1}{2}\|w\|^2 \\ s.t. \quad & y^{(i)}(w^T x^{(i)} + b) \geq 1, \quad 1, \ldots, m. \end{aligned} \tag{10.5}$$

In order to solve it, we need to Karush-Kuhn-Tucker theorem.

10.1.1 Lagrange Duality

Consider the following the primal optimization problem:

$$\begin{aligned} \min_{w} \quad & f(w) \\ s.t. \quad & g_i(w) \geq 0, \quad 1, \ldots, k \\ & h_i(w) = 0, \quad 1, \ldots, l. \end{aligned} \tag{10.6}$$

Define the generalized Lagrangian

$$L(w, \alpha, \beta) = f(w) + \sum_{i=1}^{k} \alpha_i g_i(w) + \sum_{i=1}^{l} \beta_i h_i(w). \tag{10.7}$$

Consider the quantity

$$\theta_p(w) = \max_{\alpha, \beta : \alpha_i \geq 0} L(w, \alpha, \beta).$$

Then the minimization problem

$$\min_{w} \theta_p(w) = \min_{w} \max_{\alpha, \beta : \alpha_i \geq 0} L(w, \alpha, \beta)$$

is the same as the primal one. Consider the dual problem

$$\max_{\alpha, \beta : \alpha_i \geq 0} \theta_d(\alpha, \beta) = \max_{\alpha, \beta : \alpha_i \geq 0} \min_{w} L(w, \alpha, \beta),$$

where $\theta_d(\alpha, \beta) = \min_w L(w, \alpha, \beta)$. Under some conditions, the solution of primal and dual problems are the same. Suppose f and the g_i's are convex, and the h_i's are affine. Suppose further that the constraints g_i are (strictly) feasible; this means that there exists some w so that $g_i(w) < 0$ for all i. Then there exist w^*, α^*, β^* so that w^* is the solution to the primal problem, and α^*, β^* are the solution to the dual problem, and satisfy the Karush-Kuhn-Tucker (KKT) conditions, which are as follows:

$$\begin{aligned} \frac{\partial}{\partial w_i} L(w^*, \alpha^*, \beta^*) &= 0, \qquad i = 1, \ldots, n \\ \frac{\partial}{\partial \beta_i} L(w^*, \alpha^*, \beta^*) &= 0, \qquad i = 1, \ldots, l \\ \alpha_i g_i(w^*) &= 0, \qquad i = 1, \ldots, k \\ g_i(w^*) &\leq 0, \qquad i = 1, \ldots, k \\ \alpha_i &\geq 0, \qquad i = 1, \ldots, k \end{aligned} \tag{10.8}$$

Moreover, if some w^*, α^*, β^* satisfy the KKT conditions, then it is also a solution to the primal and dual problems. The third equation is called the KKT dual complementarity condition.

10.1.2 Optimal margin classifiers

In order to apply KKT theorem to 10.5, we rewrite its condition as

$$g_i(w) = -y^{(i)}(w^T x^{(i)} + b) + 1 \leq 0.$$

From the KKT complementarity condition we have $\alpha_i > 0$ only for training example that have functional margin equal to 1. The Lagrangian for our optimization problem is

$$L(w, b, \alpha) = \frac{1}{2}\|w\|^2 - \sum_{i=1}^{m} \alpha_i [y^{(i)}(w^T x^{(i)} + b) - 1].$$

To find the dual problem, we need

$$\theta_D(\alpha) = \min_{w,b} L(w, b, \alpha).$$

We have

$$\nabla_w L(w, b, \alpha) = w - \sum_{i=1}^{m} \alpha_i y^{(i)} x^{(i)} = 0 \Longrightarrow w = \sum_{i=1}^{m} \alpha_i y^{(i)} x^{(i)}. \tag{10.9}$$

$$\frac{\partial}{\partial b} L(w, b, \alpha) = \sum_{i=1}^{m} \alpha_i y^{(i)} = 0.$$

Replacing into the Lagrangian and simplifying, we get

$$L(w, b, \alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y^{(i)} y^{(j)} \alpha_i \alpha_j (x^{(i)})^T x^{(j)}.$$

Hence we have the following dual optimization problem

$$\begin{aligned} \max_{\alpha} \quad & W(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y^{(i)} y^{(j)} \alpha_i \alpha_j \langle x^{(i)}, x^{(j)} \rangle \\ s.t. \quad & \alpha_i \geq 0, \quad 1, \ldots, m \\ & \sum_{i=1}^{m} \alpha_i y^{(i)} = 0. \end{aligned} \tag{10.10}$$

The conditions required for optimal solution and the KKT conditions to hold are indeed satisfied in our optimization problem. Hence, we can solve the dual in lieu of solving the primal problem. Specifically, in the dual problem above, we have a maximization problem in which the parameters are the α_i's, then we can find the optimal value of w. Having found w^* we can find

$$b^* = -\frac{\max_{i:y^{(i)}=-1} w^{*T} x^{(i)} + \min_{i:y^{(i)}=1} w^{*T} x^{(i)}}{2}.$$

In order to make a prediction at a new point input x, we have to compute $h_(w,b)(x) = g(w^T x + b)$. Using 10.9, we have

$$w^T x + b = \left(\sum_{i=1}^{m} \alpha_i y^{(i)} x^{(i)} \right)^T x + b = \sum_{i=1}^{m} \alpha_i y^{(i)} \langle x^{(i)}, x \rangle + b.$$

In order to make a prediction, we have to calculate a quantity that depends only on the inner product between x and the points in the training set. Moreover, since α_i's all are zero except for the support vectors. Thus, many of the terms in the sum above will be zero, and we really need to find only the inner products between x and the support vectors in order to make prediction.

10.1.3 Kernels

Let ϕ denote the feature mapping. Rather than applying SVMs using the original input x, we instead want to learn using some features $\phi(x)$. To do so, we replace x everywhere in the algorithm with $\phi(x)$. Since the algorithm can be written entirely in terms of the inner products $\langle x, z \rangle$, this means that we would replace all those inner products with $\langle \phi(x), \phi(z) \rangle$. Specifically, given a feature mapping ϕ, we define the corresponding Kernel to be

$$K(x, z) = \phi(x)^T \phi(z).$$

Then, everywhere we had $\langle x, z \rangle$ in our algorithm, we could simply replace it with $K(x, z)$, and our algorithm would now be learning using the features ϕ. Now, given ϕ, we could easily compute $K(x, z)$ by finding $\phi(x)$ and $\phi(z)$ and taking their inner product. But often, $K(x, z)$ may be very inexpensive to calculate, even though ϕ itself may be very expensive to calculate.

Suppose for now that K is a valid kernel corresponding to some feature mapping ϕ. Now, consider some finite set of m points $\{x^{(1)}, \ldots, x^{(m)}\}$, and let a square, m-by-m matrix K be defined so that its (i, j)-entry is given by $K_{ij} = K(x^{(i)}, x^{(j)})$. This matrix is called the Kernel matrix. The following result is due to Mercer.

Theorem (Mercer). Let $K : \Re^n \times \Re^n \to \Re$ be given. Then for K to be a valid (Mercer) kernel, it is necessary and sufficient that for any $\{x^{(1)}, \ldots, x^{(m)}\}$, $(m < 1)$, the corresponding kernel matrix is symmetric positive semi-definite.

10.1.4 Regularization and the non-separable case

The derivation of the SVM assumed that the data is linearly separable. While mapping data to a high dimensional feature space via ϕ does generally increase the likelihood that the data is separable, we can't guarantee that it always will be so. Also, in some cases it is not clear that finding a separating hyperplane is exactly what we'd want to do, since that might be susceptible to outliers.

To make the algorithm work for non-linearly separable datasets as well as be less sensitive to outliers, we reformulate our optimization as follows:

$$\begin{aligned} \min_{\xi,w,b} \quad & \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{m}\xi_i \\ s.t. \quad & y^{(i)}(w^T x^{(i)} + b) \geq 1 - \xi_i, \quad 1,\ldots,m \\ & \xi_i \geq 0, \quad 1,\ldots,m. \end{aligned} \tag{10.11}$$

Thus, examples are now permitted to have functional margin less than 1, and if an example whose functional margin is $1 - \xi_i$, we would pay a cost of the objective function being increased by $C\xi_i$. The parameter C controls the relative weighting between the twin goals of making the $\|w\|^2$ large and of ensuring that most examples have functional margin at least 1.

The Lagrangian is

$$L(w,b,\xi,\alpha,r) = \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{m}\xi_i - \sum_{i=1}^{m}\alpha_i[y^{(i)}(w^T x^{(i)} + b) - 1 + \xi_i] - \sum_{i=1}^{m} r_i\xi_i.$$

The dual optimization problem is

$$\begin{aligned} \max_{\alpha} \quad & W(\alpha) = \sum_{i=1}^{m}\alpha_i - \frac{1}{2}\sum_{i,j=1}^{m} y^{(i)}y^{(j)}\alpha_i\alpha_j\langle x^{(i)}, x^{(j)}\rangle \\ s.t. \quad & 0 \leq \alpha_i \leq 0, \quad 1,\ldots,m \\ & \sum_{i=1}^{m}\alpha_i y^{(i)} = 0. \end{aligned} \tag{10.12}$$

As before, w can be expressed in terms of the α_i's as given in Equation 10.9. The KKT dual-complementarity conditions are

$$\begin{aligned} \alpha_i = 0 \quad &\Rightarrow \quad y^{(i)}(w^T x^{(i)} + b) \geq 1 \\ \alpha_i = C \quad &\Rightarrow \quad y^{(i)}(w^T x^{(i)} + b) \leq 1 \\ 0 < \alpha_i < C \quad &\Rightarrow \quad y^{(i)}(w^T x^{(i)} + b) = 1. \end{aligned}$$

Note that the dual problems can be solved by sequential minimal optimization algorithm.

10.2 Kernelized Support Vector Machines

Adding nonlinear features to the representation of our data can make linear models much more powerful. However, often we don't know which features to add, and adding many features might make computation very expensive.

Luckily, there is a clever mathematical trick that allows us to learn a classifier in a higher-dimensional space without actually computing the new, possibly very large representation. This is known as the kernel trick, and it works by directly computing the distance (more precisely, the scalar products) of the data points for the expanded feature representation, without ever actually computing the expansion.

There are two ways to map your data into a higher-dimensional space that are commonly used with support vector machines:

- The polynomial kernel, which computes all possible polynomials up to a certain degree of the original features
- The radial basis function (RBF) kernel, also known as the Gaussian kernel. It considers all possible polynomials of all degrees, but the importance of the features decreases for higher degrees

During training, the SVM learns how important each of the training data points is to represent the decision boundary between the two classes. Typically only a subset of the training points matter for defining the decision boundary: the ones that lie on the border between the classes. These are called support vectors and give the support vector machine its name.

To make a prediction for a new point, the distance to each of the support vectors is measured. A classification decision is made based on the distances to the support vector, and the importance of the support vectors that was learned during training.

```
from sklearn.svm import SVC
X, y = dataset
X_train, X_test, y_train, y_test = train_test_split(X,
    y, random_state=0)
svm = SVC(kernel='rbf', C=10, gamma=0.1)
svc.fit(X_train, y_train)
print("Accuracy on training set: {:.2f}".format(svc.
    score(X_train, y_train)))
print("Accuracy on test set: {:.2f}".format(svc.score(
    X_test, y_test)))
```

Strengths, weaknesses, and parameters:

Kernelized support vector machines are powerful models and perform well on a variety of datasets. SVMs allow for complex decision boundaries, even if the data has only a few features. They work well on low-dimensional and high-dimensional data (few and many features), but don't scale very well with the number of samples. Running an SVM on data with up to 10,000 samples might work well, but working with datasets of size 100,000 or more can become challenging in terms of runtime and memory usage. Another downside of SVMs is that they require careful preprocessing of the data and tuning of the parameters. Furthermore, SVM models are hard to inspect; it can be

difficult to understand why a particular prediction was made, and it might be tricky to explain the model to a non-expert.

The important parameters in kernel SVMs are the regularization parameter C, the choice of the kernel, and the kernel-specific parameters. The RBF kernel has only one parameter, gamma, which is the inverse of the width of the Gaussian kernel. gamma and C both control the complexity of the model, with large values in either resulting in a more complex model. Therefore, good settings for the two parameters are usually strongly correlated, and C and gamma should be adjusted together.

11

Unsupervised Learning

In unsupervised learning the training set is $\{x^{(1)}, x^{(2)}, \ldots, x^{(m)}\}$ where there is no "right answers" $y^{(i)}$. The goal is to find a structure in the data. The learning algorithm is just shown the input data and asked to extract knowledge from this data. There are two types of unsupervised learning:

- **transformations of the dataset:** Unsupervised transformations of a dataset are algorithms that create a new representation of the data which might be easier for humans or other machine learning algorithms to understand compared to the original representation of the data. A common application of unsupervised transformations 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.
- **clustering:** Clustering algorithms, partition data into distinct groups of similar items.

A major challenge in unsupervised learning is evaluating whether the algorithm learned something useful. Unsupervised learning algorithms are usually applied to data that does not contain any label information, so we don't know what the right output should be. Therefore, it is very hard to say whether a model "did well." The only way to evaluate the result of an unsupervised algorithm is to inspect it manually.

Unsupervised algorithms are used often in an exploratory setting, when a data scientist wants to understand the data better. Another common application for unsupervised algorithms is as a preprocessing step for supervised 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.

11.1 Clustering

The goal of clustering is to find a natural grouping in data such that items in the same cluster are more similar to each other than those from different clusters.

11.1.1 K-means

We will discuss one of the most popular clustering algorithms, K-means which is widely used in academia as well as in industry. Clustering (or cluster analysis) is a technique that allows us to find groups of similar objects, objects that are more related to each other than to objects in other groups. K-means is an iterative algorithm and it does

1. Cluster assignment step: assign point which is closer to cluster point
2. Move centroid step: move the centroid to the center of each cluster.

K-means algorithm

It has inputs K, the number of clusters and the training set $\{x^{(1)}, x^{(2)}, \ldots, x^{(m)}\}$, where $x^{(i)} \in \mathbf{R}^n$. No need the convention $x_0 = 1$.

- Randomly initialize K cluster centroids $\mu_1, \mu_2, \ldots, \mu_k \in \mathbf{R}^n$.
- Repeat {
  for $i = 1 : m$ (cluster assignment step)
  $c^{(i)}$:= index of cluster centroid closest to $x^{(i)}$ ($c^{(i)} = \min_k \|x^{(i)} - \mu_k\|^2$)

  for $k = 1 : K$ (move centroid step)
  μ_k := average of points assigned to cluster k.
  }

Note that if there is no point in a cluster you can remove it.

K-means optimization objective

We have the following parameters.

- $c^{(i)}$:= index of cluster $\{1, 2, \ldots, K\}$ to which example $x^{(i)}$ is currently assigned.
- μ_k := cluster centroid k.
- μ_{c^i} := cluster centroid of cluster to which example $x^{(i)}$ has been assigned.

The cost function is

$$J(c^{(1)}, c^{(2)}, \ldots, c^{(m)}, \mu_1, \mu_2, \ldots, \mu_K) = \frac{1}{m} \sum_{i=1}^{m} \|x^{(i)} - \mu_{c^{(i)}}\|^2.$$

The optimization objective is

$$\min_{c,\mu} J(c^{(1)}, c^{(2)}, \ldots, c^{(m)}, \mu_1, \mu_2, \ldots, \mu_K).$$

In cluster assignment step of K-means algorithm, we minimize J with respect to $c^{(1)}, c^{(2)}, \ldots, c^{(m)}$ while holding $\mu_1, \mu_2, \ldots, \mu_K$ fixed. In move centroid step, we minimize J with respect to $\mu_1, \mu_2, \ldots, \mu_K$.

Random initialization

- We should have $K < m$.
- Randomly pick K training example.
- Set $\mu_1, \mu_2, \ldots, \mu_K$ equal to those K examples.

Choosing the cluster:

For $i = 1 : 100($ or $50 - 1000)$

Randomly initialize K-means
Run K-means to get c and μ.
Compute cost function (distortion) J.

Pick clustering that gave lowest cost J.

```
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
X = dataset
scaler = StandardScaler()
X_std = scaler.fit_transform(X)
kmeans = KMeans(n_clusters=3)
kmeans.fit(X_std)
kmeans.predict(X_std)
```

In order to speed up the KMeans clustering, we can use MiniBatchK-Means. MiniBatchKMeans works similarly to KMeans, with one significant difference: the batch_size parameter. batch_size controls the number of randomly selected observations in each batch. The larger the size of the batch, the more computationally costly the training process.

Failure of K-means

Even if you know the "right" number of clusters for a given dataset, k-means might not always be able to recover them. Each cluster is defined solely by its center, which means that each cluster is a convex shape. As a result of this, k-means can only capture relatively simple shapes. k-means also assumes that all clusters have the same "diameter" in some sense; it always draws the

boundary between clusters to be exactly in the middle between the cluster centers. That can sometimes lead to surprising results.

k-means also assumes that all directions are equally important for each cluster. k-means also performs poorly if the clusters have more complex shapes.

k-means is a very popular algorithm for clustering, not only because it is relatively easy to understand and implement, but also because it runs relatively quickly. k-means scales easily to large datasets, and *scikit-learn* even includes a more scalable variant in the *MiniBatchKMeans* class, which can handle very large datasets.

One of the drawbacks of k-means is that it relies on a random initialization, which means the outcome of the algorithm depends on a random seed. By default, *scikit-learn* runs the algorithm 10 times with 10 different random initializations, and returns the best result. Further downsides of k-means are the relatively restrictive assumptions made on the shape of clusters, and the requirement to specify the number of clusters you are looking for.

11.1.2 Agglomerative Clustering

Agglomerative clustering refers to a collection of clustering algorithms that all build upon the same principles: the algorithm starts by declaring each point its own cluster, and then merges the two most similar clusters until some stopping criterion is satisfied. The stopping criterion implemented in *scikit-learn* is the number of clusters, so similar clusters are merged until only the specified number of clusters are left. There are several linkage criteria that specify how exactly the "most similar cluster" is measured. This measure is always defined between two existing clusters.

The following three choices are implemented in *scikit-learn*:

- *ward:* The default choice, ward picks the two clusters to merge such that the variance within all clusters increases the least. This often leads to clusters that are relatively equally sized.
- *average:* average linkage merges the two clusters that have the smallest average distance between all their points.
- *complete:* complete linkage (maximum linkage) merges the two clusters that have the smallest maximum distance between their points.

```
from sklearn.cluster import AgglomerativeClustering
X = dataset
agg = AgglomerativeClustering(n_clusters=3)
assignment = agg.fit_predict(X)
```

11.1.3 Hierarchical clustering and dendrograms

11.1.4 DBSCAN

Another very useful clustering algorithm is DBSCAN (density based spatial clustering of applications with noise). The main benefits of DBSCAN are that it does not require the user to set the number of clusters a priori, it can capture clusters of complex shapes, and it can identify points that are not part of any cluster. DBSCAN is somewhat slower than agglomerative clustering and k-means, but still scales to relatively large datasets.

DBSCAN works by identifying points that are in "crowded" regions of the feature space, where many data points are close together. These regions are referred to as dense regions in feature space. The idea behind DBSCAN is that clusters form dense regions of data, separated by regions that are relatively empty.

Points that are within a dense region are called core samples (core points), and they are defined as follows. There are two parameters in DBSCAN: *min_samples* and *eps*. If there are at least *min_samples* many data points within a distance of *eps* to a given data point, that data point is classified as a core sample. Core samples that are closer to each other than the distance *eps* are put into the same cluster by DBSCAN.

The algorithm works by picking an arbitrary point to start with. It then finds all points with distance *eps* or less from that point. If there are less than *min_samples* points within distance *eps* of the starting point, this point is labeled as noise, meaning that it does not belong to any cluster. If there are more than *min_samples* points within a distance of eps, the point is labeled a core sample and assigned a new cluster label. Then, all neighbors (within *eps*) of the point are visited. If they have not been assigned a cluster yet, they are assigned the new cluster label that was just created. If they are core samples, their neighbors are visited in turn, and so on. The cluster grows until there are no more core samples within distance eps of the cluster. Then another point that hasn't yet been visited is picked, and the same procedure is repeated.

In the end, there are three kinds of points: core points, points that are within distance *eps* of core points (called boundary points), and noise. When the DBSCAN algorithm is run on a particular dataset multiple times, the clustering of the core points is always the same, and the same points will always be labeled as noise. However, a boundary point might be neighbor to core samples of more than one cluster. Therefore, the cluster membership of boundary points depends on the order in which points are visited. Usually there are only few boundary points, and this slight dependence on the order of points is not important.

Increasing *eps* means that more points will be included in a cluster. This makes clusters grow, but might also lead to multiple clusters joining into one. Increasing *min_samples* means that fewer points will be core points, and more points will be labeled as noise.

While DBSCAN doesn't require setting the number of clusters explicitly, setting *eps* implicitly controls how many clusters will be found. Finding a good setting for *eps* is sometimes easier after scaling the data using *StandardScaler* or *MinMaxScaler*, as using these scaling techniques will ensure that all features have similar ranges.

```
from sklearn.cluster import DBSCAN
X = dataset
scaler = StandardScaler()
scaler.fit(X)
X_scaled = scaler.transform(X)
dbscan = DBSCAN()
clusters = dbscan.fit_predict(X_scaled)
```

11.1.5 Comparing and Evaluating Clustering Algorithms

There are metrics that can be used to assess the outcome of a clustering algorithm relative to a ground truth clustering, the most important ones being the adjusted rand index (ARI) and normalized mutual information (NMI), which both provide a quantitative measure between 0 and 1.

There are scoring metrics for clustering that don't require ground truth, like the silhouette coefficient. However, these often don't work well in practice. The silhouette score computes the compactness of a cluster, where higher is better, with a perfect score of 1.

A slightly better strategy for evaluating clusters is using robustness-based clustering metrics.

11.2 Dimensionality Reduction

The motivation for dimensionality reduction is data compression and data visualization. Data compression not only allows us to compress the data and use less computer memory, but it will also allow us to speed up our learning algorithms.

11.2.1 Principle Component Analysis (PCA)

Principal component analysis is a method that rotates the dataset in a way such that the rotated features are statistically uncorrelated. This rotation is often followed by selecting only a subset of the new features, according to how important they are for explaining the data. The algorithm proceeds by first finding the direction of maximum variance, as "Component 1." This is the direction (or vector) in the data that contains most of the information, or in other words, the direction along which the features are most correlated

with each other. Then, the algorithm finds the direction that contains the most information while being orthogonal to the first direction. The directions found using this process are called principal components, as they are the main directions of variance in the data. In general, there are as many principal components as original features.

The aim of PCA is to reduce the n-dimensional data to k-dimension with $k < n$. To do so we need to find k vectors $u^{(1)}, u^{(2)}, \ldots, u^{(k)}$ onto which to project the data so as to minimize the projection error.

We first need to do data preprocessing

- Training set: $x^{(1)}, x^{(2)}, \ldots, x^{(m)}$
- preprocessing (feature scaling/ mean normalization) $\mu_j = \frac{1}{m} \sum_{i=1}^{m} x_j^{(i)}$
- Replace $x_j^{(i)}$ with $x_j^{(i)} - \mu_j$.

PCA Algorithm:

Reduce data from n-dimensions to k-dimensions

- Compute covariance matrix $\Sigma = \frac{1}{m} \sum_{i=1}^{n} (x^{(i)})(x^{(i)})^T$
- Compute eigenvectors of Σ : $[U, S, V] = svd(\Sigma)$
  The matrix U is an $n \times n$ matrix. Choose the first k column of U and set it as $U_{reduce} = [U^{(1)}\ U^{(2)}\ \cdots\ U^{(k)}]$.
- Find $z^{(i)}$: $x \in \mathbf{R}^n \rightarrow z \in \mathbf{R}^k$

$$z = (U_{reduce}^T)x.$$

Reconstruction from compressed representation

Inorder to get back from compressed data to the original one, we do

$$x_{approx} = U_{reduce} z.$$

Choosing the number of PCA (k)

Average squared projection error: $\frac{1}{m} \sum_{i=1}^{m} \|x^{(i)} - x_{approx}^{(i)}\|^2$.
Total variation in the data: $\frac{1}{m} \sum_{i=1}^{m} \|x^{(i)}\|^2$.

Choose k to be smallest value so that

$$\frac{\frac{1}{m} \sum_{i=1}^{m} \|x^{(i)} - x_{approx}^{(i)}\|^2}{\frac{1}{m} \sum_{i=1}^{m} \|x^{(i)}\|^2} < 0.01(0.05),$$

which means "99%(95%) of variance is retained."

Here is the algorithm to choose k :

1. Try PCA with $k = 1$.

2. Compute $U_{reduce}, z^{(i)}, \ldots, z^{(m)}, x^{(1)}_{approx}, \ldots, x^{(m)}_{approx}$
3. Check if
$$\frac{\frac{1}{m}\sum_{i=1}^{m} \|x^{(i)} - x^{(i)}_{approx}\|^2}{\frac{1}{m}\sum_{i=1}^{m} \|x^{(i)}\|^2} < 0.01(0.05)?$$
4. If the above condition is true then $k = 1$, otherwise, set $k = 2$ and repeat 2 and 3.

Another application of PCA is feature extraction. The idea behind feature extraction is that it is possible to find a representation of your data that is better suited to analysis than the raw representation you were given. A great example of an application where feature extraction is helpful is with images.

```
from sklearn.decomposition import PCA
pca = PCA(n_components=2, whiten=True)
pca.fit(X)
X_pca = pca.transform(X)
```

n_components has two operations, depending on the argument provided. If the argument is greater than 1, n_components will return that many features. This leads to the question of how to select the number of features that is optimal. Fortunately for us, if the argument to n_components is between 0 and 1, pca returns the minimum amount of features that retain that much variance. It is common to use values of 0.95 and 0.99, meaning 95% and 99% of the variance of the original features has been retained, respectively. whiten=True transforms the values of each principal component so that they have zero mean and unit variance.

Standard PCA uses linear projection to reduce the features. If the data is linearly separable then PCA works well. However, if your data is not linearly separable, the linear transformation will not work as well. Kernels allow us to project the linearly inseparable data into a higher dimension where it is linearly separable; this is called the kernel trick. There are a number of kernels we can use in scikit-learn's kernelPCA, specified using the kernel parameter. Common kernels to use are the Gaussian radial basis function kernel rbf, the polynomial kernel (poly), sigmoid kernel (sigmoid), and linear (linear). One downside of kernel PCA is that there are a number of parameters we need to specify.

11.2.2 Non-Negative Matrix Factorization (NMF)

Non-negative matrix factorization is another unsupervised learning algorithm that aims to extract useful features. It works similarly to PCA and can also be used for dimensionality reduction. In PCA, the components are orthogonal and explaines the variance of the data, in NMF, we want the components and the coefficients to be nonnegative; that is, we want both the components and the coefficients to be greater than or equal to zero. Consequently, this method

can only be applied to data where each feature is non-negative. NMF leads to more interpretable components than PCA.

```
from sklearn.decomposition import NMF
nmf = NMF(n_components=10, random_state=1)
features_nmf = nmf.fit_transform(features)
```

11.2.3 Manifold Learning with t-SNE

While PCA is often a good first approach for transforming your data so that you might be able to visualize it using a scatter plot, the nature of the method limits its usefulness. There is a class of algorithms for visualization called manifold learning algorithms that allow for much more complex mappings, and often provide better visualizations. A particularly useful one is the t-SNE algorithm.

Manifold learning algorithms are mainly aimed at visualization, and so are rarely used to generate more than two new features. Some of them, including t-SNE, compute a new representation of the training data, but don't allow transformations of new data. This means these algorithms cannot be applied to a test set: rather, they can only transform the data they were trained for. Manifold learning can be useful for exploratory data analysis, but is rarely used if the final goal is supervised learning. The idea behind t-SNE is to find a two-dimensional representation of the data that preserves the distances between points as best as possible. t-SNE starts with a random two-dimensional representation for each data point, and then tries to make points that are close in the original feature space closer, and points that are far apart in the original feature space farther apart. t-SNE puts more emphasis on points that are close by, rather than preserving distances between far-apart points. In other words, it tries to preserve the information indicating which points are neighbors to each other.

12

Anomaly Detection

Anomaly detection is a technique used to identify unusual patterns that do not conform to expected behavior, called outliers. It has many applications in business, from intrusion detection (identifying strange patterns in network traffic that could signal a hack) to system health monitoring (spotting a malignant tumor in an MRI scan), and from fraud detection in credit card transactions to fault detection in operating environments.

12.1 Gaussian Distribution

If $x \in \mathbf{R}$ is a distributed Gaussian with mean μ and variance σ^2, then we say x is "distributed as" Gassian with parameter μ and σ^2, and denoted

$$x \sim \mathcal{N}(\mu, \sigma^2).$$

The formula for the Gaussian density is

$$p(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi}} \exp(-\frac{(x-\mu)^2}{2\sigma^2}).$$

Parameter estimation

Suppose we have a dataset $\{x^{(1)}, \ldots, x^{(m)}\}$ with $x^{(i)} \in \mathbf{R}$ and $x^{(i)} \sim \mathcal{N}(\mu, \sigma^2)$. We can estimate the parameters as follows:

$$\mu = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}, \qquad \sigma^2 = \frac{1}{m} \sum_{i=1}^{m} (x^{(i)} - \mu)^2.$$

Anomaly Detection Algorithm

Consider the training set $\{x^{(1)}, \ldots, x^{(m)}\}$ with $x^{(i)} \in \mathbf{R}^n$ and $x_i \sim \mathcal{N}(\mu_i, \sigma_i^2)$. We are going to model $p(x)$ as

$$p(x) = p(x_1; \mu_1, \sigma_1^2)p(x_2; \mu_2, \sigma_2^2) \cdots p(x_n; \mu_n, \sigma_n^2) = \Pi_{j=1}^{n} p(x_j; \mu_j, \sigma_j^2).$$

Algorithm:

- Choose feature x_i that you think might be indicative of anomalous example.
- Fit parameters $\mu_1, \ldots, \mu_n, \sigma_1^2, \ldots, \sigma_n^2$
- Given new example x, compute

$$p(x) = \Pi_{j=1}^{n} p(x_j; \mu_j, \sigma_j^2).$$

  It is anomaly if $p(x) < \epsilon$.

Table 12.1. Supervised learning vs Anomaly detection

Anomaly Detection	Supervised Learning
• Very small number of positive examples ($y = 1$) $0 - 20$ is common. Large number of negative examples ($y = 0$).	• Large number of positive and negative examples.
• Many different types of anomalies. Hard for any algorithm to learn from positive examples what the anomalies look like. Future anomalies may look nothing like any of the anomalous examples we have seen.	• Enough positive examples to get a sense of what positive examples are like. Future positive examples likely to be similar to ones in the training set.

Collaborative Filtering

In this part of the exercise, you will implement the collaborative filtering learning algorithm and apply it to a dataset of movie ratings. This dataset consists of ratings on a scale of 1 to 5. The dataset has $n_u = 943$ users, and $n_m = 1682$ movies.

Fist, we will load the dataset *data_movies.mat*, providing the variables Y and R in MATLAB environment. The matrix Y (a (# movies)×(# users) matrix) stores the ratings $y(i, j)$ (from 1 to 5). The matrix R is an binary-valued indicator matrix, where $R(i, j) = 1$ if user j gave a rating to movie i, and $R(i, j) = 0$ otherwise. The objective of collaborative filtering is to predict movie ratings for the movies that users have not yet rated, that is, the entries with $R(i, j) = 0$. This will allow us to recommend the movies with the highest predicted ratings to the user.

We also use the matrices X and $Theta$ where

$$X = \begin{bmatrix} -(x^{(1)})^T - \\ -(x^{(2)})^T - \\ \vdots \\ -(x^{(n_m)})^T - \end{bmatrix} \qquad Theta = \begin{bmatrix} -(\theta^{(1)})^T - \\ -(\theta^{(2)})^T - \\ \vdots \\ -(\theta^{(n_m)})^T - \end{bmatrix}$$

The i-th row of X corresponds to the feature vector $x^{(i)}$ for the i-th movie, and the j-th row of $Theta$ corresponds to one parameter vector $\theta^{(j)}$, for the j-th user. Both $x^{(i)}$ and $\theta^{(j)}$ are n-dimensional vectors.

The collaborative filtering algorithm in the setting of movie recommendations considers a set of n-dimensional parameter vectors $x^{(1)}, x^{(2)}, \ldots, x^{(n_m)}$ and $\theta^{(1)}, \theta^{(2)}, \ldots, \theta^{(n_u)}$, where the model predicts the rating for movie i by user j as $y^{(i,j)} = (\theta^{(j)})^T x^{(i)}$. Given a dataset that consists of a set of ratings produced by some users on some movies, you wish to learn the parameter vectors $x^{(1)}, x^{(2)}, \ldots, x^{(n_m)}$ and $\theta^{(1)}, \theta^{(2)}, \ldots, \theta^{(n_u)}$ that produce the best fit (minimizes the squared error).

The cost function for collaborative filtering with regularization is given by

$$\begin{aligned} J(x^{(1)}, \ldots, x^{(n_m)}, \theta^{(1)}, \ldots, \theta^{(n_u)}) =& \frac{1}{2} \sum_{(i,j):r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 \\ &+ \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} (\theta_k^{(j)})^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} (x_k^{(i)})^2. \end{aligned} \tag{12.1}$$

The gradients of the cost function is given by:

$$\begin{aligned} \frac{\partial J}{\partial x_k^{(i)}} &= \sum_{j:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})(\theta_k^{(j)}) + \lambda x_k^{(i)} \\ \frac{\partial J}{\partial \theta_k^{(j)}} &= \sum_{i:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})(x_k^{(i)}) + \lambda \theta_k^{(j)}. \end{aligned} \tag{12.2}$$

13

Large Scale Machine Learning

13.1 Stochastic Gradient Descent

If the number of features are is large, say $300,000,000$, applying batch gradient descent would be so expensive. Instead, we use stochastic gradient descent, where the cost function is defined as

$$\begin{aligned} Cost(\theta,(x^{(i)},y^{(i)})) &= \frac{1}{2}(h_\theta(x^{(i)}-y^{(i)}))^2, \\ J_{train}(\theta) &= \frac{1}{m}\sum_{i=1}^{m} Cost(\theta,(x^{(i)},y^{(i)})). \end{aligned} \tag{13.1}$$

Stochastic Gradient Descent Algorithm

- Randomly shuffle data set.
- Repeat {
  for i = 1 : m {
  $\theta_j := \theta_j - \alpha(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} \qquad j = 0,1,\ldots,n$
  }
  }

Note that 1 to 10 iteration is enough for the second part. When m is very large stochastic gradient descent is faster. The cost function of batch gradient descent should go down in every iteration while it is not necessarily true for stochastic gradient descent.

14

Pipelines

For many machine learning algorithms, the particular representation of the data is very important. This starts with scaling the data and combining features by hand and goes all the way to learning features using unsupervised machine learning. Consequently, most machine learning applications require not only the application of a single algorithm, but the chaining together of many different processing steps and machine learning models. We use the Pipeline class to simplify the process of building chains of transformations and models.

The splitting of the dataset during cross-validation should be done before doing any preprocessing. Any process that extracts knowledge from the dataset should only be applied to the training portion of the dataset, so any cross-validation should be the "outermost loop" in your processing.

The Pipeline class is a class that allows "gluing" together multiple processing steps into a single scikit-learn estimator. The Pipeline class itself has fit, predict, and score methods and behaves just like any other model in scikit-learn. The most common use case of the Pipeline class is in chaining preprocessing steps (like scaling of the data) together with a supervised model like a classifier.

```
from sklearn.pipeline import Pipeline
pipe = Pipeline([("scaler", MinMaxScaler()), ("svm",
    SVC())])
pipe.fit(X_train, y_train)
print("Test score: {:.2f}".format(pipe.score(X_test,
    y_test)))
```

We can use pipeline with grid searches. When specifying the parameter grid, there is a slight change.

```
param_grid = {'svm__C': [0.001, 0.01, 0.1, 1, 10, 100],
     'svm__gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
grid = GridSearchCV(pipe, param_grid=param_grid, cv=5)
```

```
grid.fit(X_train, y_train)
print("Best cross-validation accuracy: {:.2f}".format(
    grid.best_score_))
print("Test set score: {:.2f}".format(grid.score(X_test
    , y_test)))
print("Best parameters: {}".format(grid.best_params_))
```

The Pipeline class is not restricted to preprocessing and classification, and can join any number of estimators together. For example, you could build a pipeline containing feature extraction, feature selection, scaling, and classification, for a total of four steps. Similarly, the last step could be regression or clustering instead of classification. The only requirement for estimators in a pipeline is that all but the last step need to have a transform method, so they can produce a new representation of the data that can be used in the next step.

```
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import LogisticRegression
pipe = make_pipeline(StandardScaler(),
    LogisticRegression())
param_grid = {'logisticregression__C': [0.01, 0.1, 1,
    10, 100]}
X, y = dataset
X_train, X_test, y_train, y_test = train_test_split(X,
    y, random_state=4)
grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(X_train, y_train)
print("Best estimator:\n{}".format(grid.best_estimator_
    ))
print("Logistic regression step:\n{}".format(grid.
    best_estimator_.named_steps["logisticregression"]))
print("Logistic regression coefficients:\n{}".format(
grid.best_estimator_.named_steps["logisticregression"].
    coef_))
```

Using pipelines, we can encapsulate all the processing steps in our machine learning workflow in a single scikit-learn estimator. Another benefit of doing this is that we can now adjust the parameters of the preprocessing using the outcome of a supervised task like regression or classification

```
X, y = dataset
X_train, X_test, y_train, y_test = train_test_split(X,
    y,random_state=0)
from sklearn.preprocessing import PolynomialFeatures
pipe = make_pipeline(StandardScaler(),
    PolynomialFeatures(), Ridge())
```

```
param_grid = {'polynomialfeatures__degree': [1, 2], '
    ridge__alpha': [0.1, 1]}
grid = GridSearchCV(pipe, param_grid=param_grid, cv=5,
    n_jobs=-1)
grid.fit(X_train, y_train)
print("Test-set score: {:.2f}".format(grid.score(X_test
    , y_test)))
```

```
from sklearn import metrics
from sklearn.model_selection import KFold,
    cross_val_score
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
X, y = dataset
standardizer = StandardScaler()
logit = LogisticRegression()
pipeline = make_pipeline(standardizer, logit)
kf = KFold(n_splits=10, shuffle=True, random_state=1)
cv_results = cross_val_score(pipeline, X, y, cv=kf,
    scoring="accuracy", n_jobs=-1)
cv_results.mean()
```

Very often we will need to preprocess our data before using it to train a model. We have to be careful to properly handle preprocessing when conducting model selection. First, GridSearchCV uses cross-validation to determine which model has the highest performance. However, in cross-validation we are in effect pretending that the fold held out, as the test set is not seen, and thus not part of fitting any preprocessing steps. For this reason, we cannot preprocess the data and then run GridSearchCV. Rather, the preprocessing steps must be a part of the set of actions taken by GridSearchCV. While this might appear complex, the reality is that scikit-learn makes it simple. FeatureUnion allows us to combine multiple preprocessing actions properly. In our solution we use FeatureUnion to combine two preprocessing steps: StandardScaler and PCA. This object is called preprocess and contains both of our preprocessing steps. We then include preprocess into a pipeline with our learning algorithm. The end result is that this allows us to outsource the proper handling of fitting, transforming, and training the models with combinations of hyperparameters to scikit-learn.

```
import numpy as np
from sklearn import datasets
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline, FeatureUnion
```

```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
np.random.seed(0)
X, y = datasets
preprocess = FeatureUnion([("std", StandardScaler()), (
    "pca", PCA())])
pipe = Pipeline([("preprocess", preprocess), ("
    classifier", LogisticRegression())])
search_space = [{"preprocess__pca__n_components": [1,
    2, 3], "classifier__penalty": ["l1", "l2"], "
    classifier__C": np.logspace(0, 4, 10)}]
clf = GridSearchCV(pipe, search_space, cv=5, verbose=0,
     n_jobs=-1)
best_model = clf.fit(features, target)
```

15

Text Data

There are four kinds of string data you might see:

- Categorical data
- Free strings that can be semantically mapped to categories
- Structured string data
- Text data

Categorical data is data that comes from a fixed list. Text data that consists of phrases or sentences. Examples include tweets, chat logs, and hotel reviews, and the collected works of Shakespeare. In the context of text analysis, the dataset is often called the corpus, and each data point, represented as a single text, is called a document.

15.1 Representing Text Data as a Bag of Words

One of the most simple but effective and commonly used ways to represent text for machine learning is using the bag-of-words representation. When using this representation, we discard most of the structure of the input text, like chapters, paragraphs, sentences, and formatting, and only count how often each word appears in each text in the corpus.

Computing the bag-of-words representation for a corpus of documents consists of the following three steps:

1. Tokenization. Split each document into the words (tokens) that appear in it.
2. Vocabulary building. Collect a vocabulary of all words that appear in any of the documents, and number them.
3. Encoding. For each document, count how often each of the words in the vocabulary appear in this document.

```
from sklearn.feature_extraction.text import
    CountVectorizer
vect = CountVectorizer()
vect.fit(corps)
print("Vocabulary size: {}".format(len(vect.vocabulary_
    )))
print("Vocabulary content:\n {}".format(vect.
    vocabulary_))
bag_of_words = vect.transform(bards_words)
print("bag_of_words: {}".format(repr(bag_of_words)))
```

Stopwords

Another way that we can get rid of uninformative words is by discarding words that are too frequent to be informative. There are two main approaches: using a language specific list of stopwords, or discarding words that appear too frequently. scikitlearn has a built-in list of English stopwords in the feature_extraction.text module:

```
from sklearn.feature_extraction.text import
    ENGLISH_STOP_WORDS
print("Number of stop words: {}".format(len(
    ENGLISH_STOP_WORDS)))
print("Every 10th stopword:\n{}".format(list(
    ENGLISH_STOP_WORDS)[::10]))
```

Rescaling the Data with tf–idf

Instead of dropping features that are deemed unimportant, another approach is to rescale features by how informative we expect them to be. One of the most common ways to do this is using the term frequency–inverse document frequency (tf–idf) method. The intuition of this method is to give high weight to any term that appears often in a particular document, but not in many documents in the corpus. If a word appears often in a particular document, but not in very many documents, it is likely to be very descriptive of the content of that document. scikit-learn implements the tf–idf method in two classes: TfidfTransformer, which takes in the sparse matrix output produced by CountVectorizer and transforms it, and TfidfVectorizer, which takes in the text data and does both the bag-of-words feature extraction and the tf–idf transformation. The tf–idf score for word w in document d as implemented in both the TfidfTransformer and TfidfVectorizer classes is given by:

$$\text{tfidf}(w,d) = \text{tf}\ \log(\frac{N+1}{N_w+1}) + 1$$

where N is the number of documents in the training set, N_w is the number of documents in the training set that the word w appears in, and tf (the term frequency) is the number of times that the word w appears in the query document d (the document you want to transform or encode). Both classes also apply L2 normalization after computing the tf–idf representation. Rescaling in this way means that the length of a document (the number of words) does not change the vectorized representation.

```
from sklearn.feature_extraction.text import
    TfidfVectorizer
from sklearn.pipeline import make_pipeline
pipe = make_pipeline(TfidfVectorizer(min_df=5, norm=
    None),LogisticRegression())
param_grid = {'logisticregression__C': [0.1, 1]}
grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(text_train, y_train)
print("Best cross-validation score: {:.2f}".format(grid
    .best_score_))
```

n-Grams

One of the main disadvantages of using a bag-of-words representation is that word order is completely discarded. Therefore, the two strings "it's bad, not good at all" and "it's good, not bad at all" have exactly the same representation, even though the meanings are inverted. Putting "not" in front of a word is only one example (if an extreme one) of how context matters. Fortunately, there is a way of capturing context when using a bag-of-words representation, by not only considering the counts of single tokens, but also the counts of pairs or triplets of tokens that appear next to each other. Pairs of tokens are known as bigrams, triplets of tokens are known as trigrams, and more generally sequences of tokens are known as n-grams. We can change the range of tokens that are considered as features by changing the ngram_range parameter of CountVectorizer or TfidfVectorizer. The ngram_range parameter is a tuple, consisting of the minimum length and the maximum length of the sequences of tokens that are considered.

For most applications, the minimum number of tokens should be one, as single words often capture a lot of meaning. Adding bigrams helps in most cases. Adding longer sequences—up to 5-grams—might help too, but this will lead to an explosion of the number of features and might lead to overfitting, as there will be many very specific features.

```
pipe = make_pipeline(TfidfVectorizer(min_df=5),
    LogisticRegression())
param_grid = {"logisticregression__C": [0.1, 1], "
    tfidfvectorizer__ngram_range": [(1, 1), (1, 2)]}
```

```
grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(text_train, y_train)
print("Best cross-validation score: {:.2f}".format(grid
    .best_score_))
print("Best parameters:\n{}".format(grid.best_params_))
```

16

Deploy ML algorithm

The first step in using a model in production is to save that model as a file that can be loaded by another application or workflow. We can accomplish this by saving the model as a pickle file, a Python-specific data format.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.externals import joblib
classifer = RandomForestClassifier()
model = classifer.fit(features, target)
joblib.dump(model, "model.pkl")
```

Once the model is saved we can use scikit-learn in our destination application to load the model:

```
classifer = joblib.load("model.pkl")
```

Keras does not recommend you save models using pickle. Instead, models are saved as an HDF5 file. The HDF5 file contains everything you need to not only load the model to make predictions, but also to restart training.

```
# Save neural network
network.save("model.h5")

# Load neural network
network = load_model("model.h5")
```

References

1. Andreas C. Müller and Sarah Guido, Introduction to Machine Learning with Python, 2017.
2. Chris Albon, Machine Learning with Python Cookbook, 2018.
3. Gareth James, Daniela Witten, Trevor Hastie, Robert Tibshirani, An Introduction to Statistical Learning with Applications in R, 2017.