Python\_Prob\_Stat\_Machine\_Learning\_Unpingco\_\_2E\_C04

238

4

Machine Learning

**4.1 Introduction**

Machine Learning is a huge and growing area. In this chapter, we cannot possibly even survey this area, but we can provide some context and some connections to probability and statistics that should make it easier to think about machine learning and how to apply these methods to real-world problems. The fundamental problem of statistics is basically the same as machine learning: given some data, how to make it actionable? For statistics, the answer is to construct analytic estimators using powerful theory. For machine learning, the answer is algorithmic prediction. Given a dataset, what forward-looking inferences can we draw? There is a subtle bit in this description: how can we know the future if all we have is data about the past? This is the crux of the matter for machine learning, as we will explore in the chapter.

**4.2 Python Machine Learning Modules**

Python provides many bindings for machine learning libraries, some specialized for technologies such as neural networks, and others geared toward novice users. For our discussion, we focus on the powerful and popular Scikit-learn module. Scikit-learn is distinguished by its consistent and sensible API, its wealth of machine learning algorithms, its clear documentation, and its readily available datasets that make it easy to follow along with the online documentation. Like Pandas, Scikit-learn relies on Numpy for numerical arrays. Since its release in 2007, Scikit-learn has become the most widely used, general-purpose, open-source machine learning modules that is popular in both industry and academia. As with all of the Python modules we use, Scikit-learn is available on all the major platforms.

To get started, let’s revisit the familiar ground of linear regression using Scikitlearn. First, let’s create some data.

>>> import numpy as np

>>> from matplotlib.pylab import subplots

>>> from sklearn.linear\_model import LinearRegression

>>> X = np.arange(10) # create some data

>>> Y = X+np.random.randn(10) # linear with noise

We next import and create an instance of the LinearRegression class from Scikit-learn.

>>> from sklearn.linear\_model import LinearRegression

>>> lr=LinearRegression() # create model

Scikit-learn has a wonderfully consistent API. All Scikit-learn objects use the fit method to compute model parameters and the predict method to evaluate the model. For the LinearRegression instance, the fit method computes the coefﬁcients of the linear ﬁt. This method requires a matrix of inputs where the rows are the samples and the columns are the features. The target of the regression are the Y values, which must be correspondingly shaped, as in the following:

>>> X,Y = X.reshape((-1,1)), Y.reshape((-1,1))

>>> lr.fit(X,Y)

LinearRegression(copy\_X=True, fit\_intercept=True, n\_jobs=None, normalize=False)

>>> lr.coef\_

array([[0.94211853]])

Programming Tip

The negative one in the reshape((-1,1)) call above is for the truly lazy. Using a negative one tells Numpy to ﬁgure out what that dimension should be given the other dimension and number of array elements.

the coef\_ property of the linear regression object shows the estimated parameters for the ﬁt. The convention is to denote estimated parameters with a trailing underscore. The model has a score method that computes the R 2 value for the regression. Recall from our statistics chapter (Sect.3.7) that the R 2 value is an indicator of the quality of the ﬁt and varies between zero (bad ﬁt) and one (perfect ﬁt).

>>> lr.score(X,Y) 0.9059042979442372

Now, that we have this ﬁtted, we can evaluate the ﬁt using the predict method,

>>> xi = np.linspace(0,10,15) # more points to draw

>>> xi = xi.reshape((-1,1)) # reshape as columns

>>> yp = lr.predict(xi) 4.2 Python Machine Learning Modules

239

Fig. 4.1 The Scikit-learn module can easily perform basic linear regression. The circles show the training data and the ﬁtted line is shown in black

The resulting ﬁt is shown in Fig.4.1.

Multilinear Regression. The Scikit-learn module easily extends linear regression to multiple dimensions. For example, for multilinear regression,

y = α 0 + α 1 x 1 + α 2 x 2 + ... + α n xn

The problem is to ﬁnd all of the α terms given the training set { x 1 , x 2 , . . . , x n , y } . We can create another example dataset and see how this works,

>>> X=np.random.randint(20,size=(10,2)) Figure 4.2 shows the two-dimensional regression example, where the size of the circles is proportional to the targetted Y value. Note that we salted the output with random noise just to keep things interesting. Nonetheless, the interface with Scikitlearn is the same,

>>> Y=X.dot([1,3])+1 + np.random.randn(X.shape[0])\*20

>>> lr=LinearRegression()

>>> lr.fit(X,Y)

LinearRegression(copy\_X=True, fit\_intercept=True, n\_jobs=None, normalize=False)

>>> print(lr.coef\_) [0.35171694 4.04064287]

The coef\_ variable now has two terms in it, corresponding to the two input dimensions. Note that the constant offset is already built-in and is an option on the LinearRegression constructor. Figure 4.3 shows how the regression performs.

Polynomial Regression. We can extend this to include polynomial regression by using the PolynomialFeatures in the preprocessing sub-module. To keep it simple, let’s go back to our one-dimensional example. First, let’s create some synthetic data, 240

4

Machine Learning

Fig. 4.2 Scikit-learn can easily perform multilinear regression. The size of the circles indicate the value of the two-dimensional function of (X 1 , X 2 )

Fig. 4.3 The predicted data is plotted in black. It overlays the training data, indicating a good ﬁt

>>> from sklearn.preprocessing import PolynomialFeatures

>>> X = np.arange(10).reshape(-1,1) # create some data

>>> Y = X+X\*\*2+X\*\*3+ np.random.randn(\*X.shape)\*80

Next, we have to create a transformation from X to a polynomial of X, 4.2 Python Machine Learning Modules

241

>>> qfit = PolynomialFeatures(degree=2) # quadratic

>>> Xq = qfit.fit\_transform(X)

>>> print(Xq) [[ 1. 0. 0.] [ 1. 1. 1.] [ 1. 2. 4.] [ 1. 3. 9.] [ 1. 4. 16.] [ 1. 5. 25.] [ 1. 6. 36.] [ 1. 7. 49.] [ 1. 8. 64.] [ 1. 9. 81.]]

Note there is an automatic constant term in the output 0 th column where fit\_ transform has mapped the single-column input into a set of columns representing the individual polynomial terms. The middle column has the linear term, and the last has the quadratic term. With these polynomial features stacked as columns of Xq, all we have to do is fit and predict again. The following draws a comparison between the linear regression and the quadratic repression (see Fig.4.4),

>>> lr=LinearRegression() # create linear model

>>> qr=LinearRegression() # create quadratic model

>>> lr.fit(X,Y) # fit linear model

LinearRegression(copy\_X=True, fit\_intercept=True, n\_jobs=None, normalize=False)

>>> qr.fit(Xq,Y) # fit quadratic model

LinearRegression(copy\_X=True, fit\_intercept=True, n\_jobs=None, normalize=False)

>>> lp = lr.predict(xi)

>>> qp = qr.predict(qfit.fit\_transform(xi))

This just scratches the surface of Scikit-learn. We will go through many more examples later, but the main thing is to concentrate on the usage (i.e., fit, predict) which is standardized across all of the machine learning methods in Scikit-learn.

4.3 Theory of Learning

There is nothing so practical as a good theory. In this section, we establish the formal framework for thinking about machine learning. This framework will help us think beyond particular methods for machine learning so we can integrate new methods or combine existing methods intelligently.

Both machine learning and statistics strive to develop understanding from data. Some historical perspective helps. Most of the methods in statistics were derived toward the start of the 20th century when data were hard to come by. Society was preoccupied with the potential dangers of human overpopulation and work was focused 242

4

Machine Learning

Fig. 4.4 The title shows the R 2 score for the linear and quadratic regressions

on studying agriculture and crop yields. At this time, even a dozen data points was considered plenty. Around the same time, the deep foundations of probability were being established by Kolmogorov. Thus, the lack of data meant that the conclusions had to be buttressed by strong assumptions and solid mathematics provided by the emerging theory of probability. Furthermore, inexpensive powerful computers were not yet widely available. The situation today is much different: there are lots of data collected and powerful and easily programmable computers are available. The important problems no longer revolve around a dozen data points on a farm acre, but rather millions of points on a square millimeter of a DNA microarray. Does this mean that statistics will be superseded by machine learning?

In contrast to classical statistics, which is concerned with developing models that characterize, explain, and describe phenomena, machine learning is overwhelmingly concerned with prediction. Areas like exploratory statistics are very closely related to machinelearning, but still not as focusedonprediction. Insomesense, this is unavoidable due to the size of the data machine learning can reduce. In other words, machine learning can help distill a table of a million columns into one hundred columns, but can we still interpret one hundred columns meaningfully? In classical statistics, this was never an issue because data were of a much smaller scale. Whereas mathematical models, usually normal distributions, ﬁtted with observations are common in statistics, machine learning uses data to construct models that sit on complicated data structures and exploit nonlinear optimizations that lack closed-form solutions. A common maxim is that statistics is data plus analytical theory and machine learning is data plus computable structures. This makes it seem like machine learning is completely ad hoc and devoid of underlying theory, but this is not the case, and both 4.3 Theory of Learning

243

Fig. 4.5 In the classical statistics problem, we observe a sample and model what the urn contains

Fig. 4.6 In the machine learning problem, we want the function that colors the marbles

machine learning and statistics share many important theoretical results. By way of contrast, let us consider a concrete problem.

Let’s consider the classic balls in urns problem (see Fig.4.5): we have an urn containing red and blue balls and we draw ﬁve balls from the urn, note the color of each ball, and then try to determine the proportion of red and blue balls in the urn. We have already studied many statistical methods for dealing with this problem. Now, let’s generalize the problem slightly. Suppose the urn is ﬁlled with white balls and there is some target unknown function f that paints each selected ball either red or blue (see Fig.4.6). The machine learning problem is how to ﬁnd the f function, given only the observed red/blue balls. So far, this doesn’t sound much different from the statistical problem. However, now we want to take our estimated f function, say, f ˆ , and use it to predict the next handful of balls from another urn. Now, here’s where the story takes a sharp turn. Suppose the next urn already has some red and blue balls in it? Then, applying the function f may result in purple balls which were not seen in the training data (see Fig.4.7). What can we do? We have just scraped the surface of the issues machine learning must confront using methods that are not part of the statistics canon. 244

4

Machine Learning

Fig. 4.7 The problem is further complicated because we may see colored marbles that were not present in the original problem

4.3.1 Introduction to Theory of Machine Learning

Some formality and an example can get us going. We deﬁne the unknown target function, f : X ↦ → Y . The training set is { (x, y) } which means that we only see the function’s inputs/outputs. The hypothesis set H is the set of all possible guesses at f . This is the set from which we will ultimately draw our ﬁnal estimate, f ˆ . The machine learning problem is how to derive the best element from the hypothesis set by using the training set. Let’s consider a concrete example in the code below. Suppose X consists of all three-bit vectors (i.e., X = { 000, 001, . . . , 111 } ) as in the code below,

>>> import pandas as pd

>>> import numpy as np

>>> from pandas import DataFrame

>>> df=DataFrame(index=pd.Index(['{0:04b}'.format(i)

... for i in range(2\*\*4)], ... dtype='str', ... name='x'),columns=['f'])

Programming Tip

The string speciﬁcation above uses Python’s advanced string formatting minilanguage. In this case, the speciﬁcation says to convert the integer into a ﬁxedwidth, four-character (04b) binary representation.

Next, we deﬁne the target function f below which just checks if the number of zeros in the binary representation exceeds the number of ones. If so, then the function outputs 1 and 0 otherwise (i.e., Y = { 0, 1 } ).

>>> df.f=np.array(df.index.map(lambda i:i.count('0'))

... > df.index.map(lambda i:i.count('1')),dtype=int)

>>> df.head(8) # show top half only 4.3 Theory of Learning

245

f

x 0000 0001 0010 0011 0100 0101 0110 0111

1

1

1

0

1

0

0

0

The hypothesis set for this problem is the set of all possible functions of X. The set D represents all possible input/output pairs. The corresponding hypothesis set H has 2 16 elements, one of which matches f . There are 2 16 elements in the hypothesis set because for each of sixteen input elements, there are two possible corresponding values (zero or one) for each input. Thus, the size of the hypothesis set is 2×2×. . .× 2 = 2 16 . Now, presented with a training set consisting of the ﬁrst eight input/output pairs, our goal is to minimize errors over the training set (E in ( f ˆ )). There are 28 elements from the hypothesis set that exactly match f over the training set. But how to pick among these 2 8 elements? It seems that we are stuck here. We need another element from the problem in order to proceed. The extra piece we need is to assume that the training set represents a random sampling (in-sample data) from a greater population (out-of-sample data) that would be consistent with the population that f ˆ would ultimately predict upon. In other words, we are assuming a stable probability structure for both the in-sample and out-of-sample data. This is a major assumption!

There is a subtle consequence of this assumption—whatever the machine learning method does once deployed, in order for it to continue to work, it cannot disturb the data environment that it was trained on. Said differently, if the method is not to be trained continuously, then it cannot break this assumption by altering the generative environment that produced the data it was trained on. For example, suppose we develop a model that predicts hospital readmissions based on seasonal weather and patient health. Because the model is so effective, in the next six months, the hospital forestalls readmissions by delivering interventions that improve patient health. Clearly using the model cannot change seasonal weather, but because the hospital used the model to change patient health, the training data used to build the model is no longer consistent with the forward-looking health of the patients. Thus, there is little reason to think that the model will continue to work as well going forward.

Returning to our example, let’s suppose that the ﬁrst eight elements from X are twice as likely as the last eight. The following code is a function that generates elements from X according to this distribution.

>>> np.random.seed(12)

>>> def get\_sample(n=1):

... ... ... ... ... ...

if

n==1:

return '{0:04b}'.format(np.random.choice(list(range(8))\*2 +list(range(8,16)))) else:

return [get\_sample(1) for \_ in range(n)] 246

4

Machine Learning

Programming Tip

The function that returns random samples uses the np.random.choice function from Numpy which takes samples (with replacement) from the given iterable. Because we want the ﬁrst eight numbers to be twice as frequent as the rest, we simply repeat them in the iterable using range(8)\*2. Recall that multiplying a Python list by an integer duplicates the entire list by that integer. It does not do element-wise multiplication as with Numpy arrays. If we wanted the ﬁrst eight to be 10 times more frequent, then we would use range(8)\*10, for example. This is a simple but powerful technique that requires very little code. Note that the p keyword argument in np.random.choice also provides an explicit way to specify more complicated distributions.

The next block applies the function deﬁnition f to the sampled data to generate the training set consisting of eight elements.

>>> train=df.loc[get\_sample(8),'f'] # 8-element training set

>>> train.index.unique().shape # how many unique elements? (6,)

Notice that even though there are eight elements, there is redundancy because these are drawn according to an underlying probability. Otherwise, if we just got all sixteen different elements, we would have a training set consisting of the complete speciﬁcation of f and then we would therefore know what h ∈ H to pick! However, this effect gives us a clue as to how this will ultimately work. Given the elements in the training set, consider the set of elements from the hypothesis set that exactly match. How to choose among these? The answer is it does not matter! Why? Because under the assumption that the prediction will be used in an environment that is determined by the same probability, getting something outside of the training set is just as likely as getting something inside the training set. The size of the training set is key herethe bigger the training set, the less likely that there will be real-world data that fall outside of it and the better f ˆ will perform. 1 The following code shows the elements of the training set in the context of all possible data.

>>> df['fhat']=df.loc[train.index.unique(),'f']

>>> df.fhat x 0000 NaN 0001 NaN 0010 1.0 0011 0.0 0100 1.0 0101 NaN

1 This assumes that the hypothesis set is big enough to capture the entire training set (which it is for this example). We will discuss this trade-off in greater generality shortly. 4.3 Theory of Learning

247

0110 0.0 0111 NaN 1000 1.0 1001 0.0 1010 NaN 1011 NaN 1100 NaN 1101 NaN 1110 NaN 1111 NaN Name: fhat, dtype: float64

Note that there are NaN symbols where the training set had no values. For deﬁniteness, we ﬁll these in with zeros, although we can ﬁll them with anything we want so long as whatever we do is not determined by the training set.

>>> df.fhat.fillna(0,inplace=True) #final specification of fhat

Now, let’s pretend we have deployed this and generate some test data.

>>> test= df.loc[get\_sample(50),'f']

>>> (df.loc[test.index,'fhat'] != test).mean() 0.18

The result shows the error rate, given the probability mechanism that generates the data. The following Pandas-fu compares the overlap between the training set and the test set in the context of all possible data. The NaN values show the rows where the test data had items absent in the training data. Recall that the method returns zero for these items. As shown, sometimes this works in its favor, and sometimes not.

>>> pd.concat([test.groupby(level=0).mean(), ... train.groupby(level=0).mean()], ... axis=1, ... keys=['test','train']) test train 0000 1 NaN 0001 1 NaN 0010 1 1.0 0011 0 0.0 0100 1 1.0 0101 0 NaN 0110 0 0.0 0111 0 NaN 1000 1 1.0 1001 0 0.0 1010 0 NaN 1011 0 NaN 248

4

Machine Learning

1100

1101

1110

1111

0

0

0

0

NaN NaN NaN NaN

Note that where the test data and training data shared elements, the prediction matched; but when the test set produced an unseen element, the prediction may or may not have matched.

Programming Tip

The pd.concat function concatenates the two Series objects in the list. The axis=1 means join the two objects along the columns where each newly created column is named according to the given keys. The level=0 in the groupby for each of the Series objects means group along the index. Because the index corresponds to the 4-bit elements, this accounts for repetition in the elements. The mean aggregation function computes the values of the function for each 4bit element. Because all functions in each respective group have the same value, the mean just picks out that value because the average of a list of constants is that constant.

Now, we are in position to ask how big the training set should be to achieve a level of performance. For example, on average, how many in-samples do we need for a given error rate? For this problem, we can ask how large (on average) must the training set be in order to capture all of the possibilities and achieve perfect out-ofsample error rates? For this problem, this turns out to be sixty-three. 2 Let’s start over and retrain with these many in-samples.

>>> train=df.loc[get\_sample(63),'f']

>>> del df['fhat']

>>> df['fhat']=df.loc[train.index.unique(),'f']

>>> df.fhat.fillna(0,inplace=True) #final specification of fhat

>>> test= df.loc[get\_sample(50),'f']

>>> # error rate

>>> (df.loc[test.index,'fhat'] != df.loc[test.index,'f']).mean()

0.0

Notice that this bigger training set has a better error rate because it is able to identify the best element from the hypothesis set because the training set captured more of the complexity of the unknown f . This example shows the trade-offs between the size of the training set, the complexity of the target function, the probability structure of the data, and the size of the hypothesis set. Note that upon exposure to the data, the so-called learning method did nothing besides memorize the data and give any unknown, newly encountered data the zero output. This means that the hypothesis set contains the single hypothesis function that memorizes and defaults to zero output.

2 This is a slight generalization of the classic coupon collector problem. 4.3 Theory of Learning

249

If the method attempted to change the default zero output based on the particular data, then we could say that meaningful learning took place. What we lack here is generalization, which is the topic of the next section.

4.3.2 Theory of Generalization

What we really want to know is how our method will perform once deployed. It would be nice to have some kind of performance guarantee. In other words, we worked hard to minimize the errors in the training set, but what errors can we expect at deployment? In training, we minimized the in-sample error, E in ( f ˆ ), but that’s not good enough. We want guarantees about the out-of-sample error, E out ( f ˆ ). This is what generalization means in machine learning. The mathematical statement of this is the following:

P ( f ˆ ) − E in ( f ˆ ) | > Eout < δ ( | )

for and δ. Informally, this says that the probability of the respective errors differing by more than a given is less than some quantity, δ. This means that whatever the performance on the training set, it should probably be pretty close to the corresponding performance once deployed. Note that this does not say that the in-sample errors (E in ) are any good in an absolute sense. It just says that we would not expect much different after deployment. Thus, good generalization means no surprises after deployment, not necessarily good performance. There are two main ways to get at this: cross-validation and probability inequalities. Let’s consider the latter ﬁrst. There are two entangled issues: the complexity of the hypothesis set and the probability of the data. It turns out we can separate these two by deriving a separate notion of complexity free from any particular data probability.

VC Dimension. We ﬁrst need a way to quantify model complexity. Following Wasserman [1], let A be a class of sets and F = { x 1 , x 2 , . . . , x n } , a set of n data points. Then, we deﬁne

N A (F) = # { F ∩ A : A ∈ A }

This counts the number of subsets of F that can be extracted by the sets of A. The number of items in the set (i.e., cardinality) is noted by the # symbol. For example, suppose F = { 1 } and A = { (x ≤ a) } . In other words, A consists of all intervals closed on the right and parameterized by a. In this case we have N A (F) = 1 because all elements can be extracted from F using A. Speciﬁcally, any a > 1 means that A contains F.

The shatter coefﬁcient is deﬁned as,

s(A, n) = max N A (F) F∈ Fn 250

4

Machine Learning

where F consists of all ﬁnite sets of size n. Note that this sweeps over all ﬁnite sets so we don’t need to worry about any particular dataset of ﬁnitely many points. The deﬁnition is concerned with A and how its sets can pick off elements from the dataset. A set F is shattered by A if it can pick out every element in it. This provides a sense of how the complexity in A consumes data. In our last example, the set of half-closed intervals shattered every singleton set { x 1 } .

Now, we come to the main deﬁnition of the Vapnik–Chervonenkis [2] dimension d VC which deﬁned as the largest k for which s(A, n) = 2 k , except in the case where s(A, n) = 2 n for which it is deﬁned as inﬁnity. For our example where F = { x 1 } , we already saw that A shatters F. How about when F = { x 1 , x 2 } ? Now, we have two points and we have to consider whether all subsets can be extracted by A. In this case, there are four subsets, { ∅, { x 1 } , { x 2 } , { x 1 , x 2 }} . Note that ∅ denotes the empty set. The empty set is easily extracted—pick a so that it is smaller than both x 1 and x 2 . Assuming that x 1 < x 2 , we can get the next set by choosing x 1 < a < x 2 . The last set is likewise doable by choosing x 2 < a. The problem is that we cannot capture the third set, { x 2 } , without capturing x 1 as well. This means that we cannot shatter any ﬁnite set with n = 2 using A. Thus, d VC = 1.

Here is the climatic result

8 4((2n) d VC + 1) E out ( f ˆ ) ≤ E in ( f ˆ ) + ln √ n ( δ )

with probability at least 1 − δ. This basically says that the expected out-of-sample error can be no worse than the in-sample error plus a penalty due to the complexity of the hypothesis set. The expected in-sample error comes from the training set but the complexity penalty comes from just the hypothesis set, thus disentangling these two issues.

A general result like this, for which we do not worry about the probability of the data, is certain to be pretty generous, but nonetheless, it tells us how the complexity penalty enters into the out-of-sample error. In other words, the bound on E out ( f ˆ ) gets worse for a more complex hypothesis set. Thus, this generalization bound is a useful guideline but not very practical if we want to get a good estimate of E out ( f ˆ ).

4.3.3 Worked Example for Generalization/Approximation Complexity

The stylized curves in Fig.4.8 illustrate the idea that there is some optimal point of complexity that represents the best generalization given the training set.

To get a ﬁrm handle on these curves, let’s develop a simple one-dimensional machine learning method and go through the steps to create this graph. Let’s suppose we have a training set consisting of x-y pairs { (x i , y i ) } . Our method groups the x-data into intervals and then averages the y-data in those intervals. Predict- 4.3 Theory of Learning

251

Fig. 4.8 In the ideal situation, there is a best model that represents the optimal tradeoff between complexity and error. This is shown by the vertical line

ing for new x-data means simply identifying the interval containing the new data then reporting the corresponding value. In other words, we are building a simple one-dimensional, nearest neighbor classiﬁer. For example, suppose the training set x-data is the following:

>>> train=DataFrame(columns=['x','y'])

>>> train['x']=np.sort(np.random.choice(range(2\*\*10),size=90))

>>> train.x.head(10) # first ten elements 0 15 1 30 2 45 3 65 4 76 5 82 6 115 7 145 8 147 9 158 Name: x, dtype: int64

In this example, we took a random set of 10-bit integers. To group these into, say, ten intervals, we simply use Numpy reshape as in the following:

>>> train.x.values.reshape(10,-1) array([[ 15, 30, 45, 65, 76, 82, 115, 145, 147], [158, 165, 174, 175, 181, 209, 215, 217, 232], [233, 261, 271, 276, 284, 296, 318, 350, 376], [384, 407, 410, 413, 452, 464, 472, 511, 522], [525, 527, 531, 534, 544, 545, 548, 567, 567], [584, 588, 610, 610, 641, 645, 648, 659, 667], [676, 683, 684, 697, 701, 703, 733, 736, 750], [754, 755, 772, 776, 790, 794, 798, 804, 830], [831, 834, 861, 883, 910, 910, 911, 911, 937], [943, 946, 947, 955, 962, 962, 984, 989, 998]]) 252

4

Machine Learning

where every row is one of the groups. Note that the range of each group (i.e., length of the interval) is not preassigned, and is learned from the training data. For this example, the y-values correspond to the number of ones in the bit representation of the x-values. The following code deﬁnes this target function,

>>> f\_target=np.vectorize(lambda i:i.count('1'))

Programming Tip

The above function uses np.vectorize which is a convenience method in Numpy that converts plain Python functions into Numpy versions. This basically saves additional looping semantics and makes it easier to use with other Numpy arrays and functions.

Next, we create the bit representations of all of the x-data below and then complete training set y-values,

>>> train['xb']= train.x.map('{0:010b}'.format)

>>> train.y=train.xb.map(f\_target)

>>> train.head(5) x y xb 0 15 4 0000001111 1 30 4 0000011110 2 45 4 0000101101 3 65 2 0001000001 4 76 3 0001001100

To train on this data, we just group by the speciﬁed amount and then average the y-data over each group.

>>> train.y.values.reshape(10,-1).mean(axis=1) array([3.55555556, 4.88888889, 4.44444444, 4.88888889, 4.11111111, 4. , 6. , 5.11111111, 6.44444444, 6.66666667])

Note that the axis=1 keyword argument just means average across the columns. So far, this deﬁnes the training. To predict using this method, we have to extract the edges from each of the groups and then ﬁll in with the group-wise mean we just computed for y. The following code extracts the edges of each group.

>>> le,re=train.x.values.reshape(10,-1)[:,[0,-1]].T

>>> print (le) # left edge of group

[ 15 158 233 384 525 584 676 754 831 943]

>>> print (re) # right edge of group

[147 232 376 522 567 667 750 830 937 998] 4.3 Theory of Learning

253

Next, we compute the group-wise means and assign them to their respective edges.

>>> val = train.y.values.reshape(10,-1).mean(axis=1).round()

>>> func = pd.Series(index=range(1024))

>>> func[le]=val # assign value to left edge

>>> func[re]=val # assign value to right edge

>>> func.iloc[0]=0 # default 0 if no data

>>> func.iloc[-1]=0 # default 0 if no data

>>> func.head() 0 0.0 1 NaN 2 NaN 3 NaN 4 NaN dtype: float64

Note that the Pandas Series object automatically ﬁlls in unassigned values with NaN. We have thus far only ﬁlled in values at the edges of the groups. Now, we need to ﬁll in the intermediate values.

>>> fi=func.interpolate('nearest')

>>> fi.head() 0 0.0 1 0.0 2 0.0 3 0.0 4 0.0 dtype: float64

The interpolate method of the Series object can apply a wide variety of powerful interpolation methods, but we only need the simple nearest neighbor method to create our piece-wise approximant. Figure 4.9 shows how this looks for the training data we have created.

Now, with all that established, we can now draw the curves for this machine learning method. Instead of partitioning the training data for cross-validation (which we’ll discuss later), we can simulate test data using the same mechanism as for the training data, as shown next,

>>> test=pd.DataFrame(columns=['x','xb','y'])

>>> test['x']=np.random.choice(range(2\*\*10),size=500)

>>> test.xb= test.x.map('{0:010b}'.format)

>>> test.y=test.xb.map(f\_target)

>>> test.sort\_values('x',inplace=True)

The curves are the respective errors for the training data and the testing data. For our error measure, we use the mean squared error,

n 1 f ˆ (xi E out = n ∑ ( ) − y i )2 i=1 254

4

Machine Learning

Fig. 4.9 The vertical lines show the training data and the thick black line is the approximant we have learned from the training data

where { (x i , y i ) } i=1 n come from the test data. The in-sample error (E in ) is deﬁned the same except for the in-sample data. In this example, the size of each group is proportional to d VC , so the more groups we choose, the more complexity in the ﬁtting. Now, we have all the ingredients to understand the trade-offs of complexity versus error.

Figure 4.10 shows the curves for our one-dimensional clustering method. The dotted line shows the mean squared error on the training set and the other line shows the same for the test data. The shaded region is the complexity penalty of this method. Note that with enough complexity, the method can exactly memorize the testing data, but that only penalizes the testing error (E out ). This effect is exactly what the Vapnik–Chervonenkis theory expresses. The horizontal axis is proportional to the VC dimension. In this case, complexity boils down to the number of intervals used in the sectioning. At the far right, we have as many intervals as there are elements in the dataset, meaning that every element is wrapped in its own interval. The average value of the data in that interval is therefore just the corresponding y value because there are no other elements to average over.

Before we leave this problem, there is another way to visualize the performance of our learning method. This problem can be thought of as a multi-class identiﬁcation problem. Given a 10-bit integer, the number of ones in its binary representation is in one of the classes { 0, 1, . . . , 10 } . The output of the model tries to put each integer in its respective class. How well this was done can be visualized using a confusion matrix as shown in the next code block,

>>> from sklearn.metrics import confusion\_matrix

>>> cmx=confusion\_matrix(test.y.values,fi[test.x].values)

>>> print(cmx) 4.3 Theory of Learning

255

Fig. 4.10 The dotted line shows the mean squared error on the training set and the other line shows the same for the test data. The shaded region is the complexity penalty of this method. Note that as the complexity of the model increases, the training error decreases, and the method essentially memorizes the data. However, this improvement in training error comes at the cost of larger testing error

[[ 1 [ 1 [ 0 [ 1 [ 0 [ 0 [ 0 [ 1 [ 4 [ 2

0

0

0

0

0

0

0

0

0

0

0 0 0 0 0 0 1 0 1 1 0 0 3 9 7 4 0 0 3 23 19 6 6 0 1 26 27 14 27 2 3 15 31 28 30 8 1 8 18 20 25 23 1 10 5 13 7 19 1 2 0 2 2 7 0 0 0 1 0 0

0

0

0

2

2

1

2

3

4

0

0] 0] 5] 0] 0] 0] 2] 6] 3] 0]]

The rows of this 10 × 10 matrix show the true class and the columns indicate the class that the model predicted. The numbers in the matrix indicate the number of times that association was made. For example, the ﬁrst row shows that there was one entry in the test set with no ones in its binary representation (i.e, namely the number zero) and it was correctly classiﬁed (namely, it is in the ﬁrst row, ﬁrst column of the matrix). The second row shows there were four entries total in the test set with a binary representation containing exactly a single one. This was incorrectly classiﬁed as the 0-class (i.e, ﬁrst column) once, the 2-class (third column) once, the 4-class (ﬁfth column) once, and the 5-class (sixth column) once. It was never classiﬁed correctly because the second column is zero for this row. In other words, the diagonal entries show the number of times it was correctly classiﬁed. 256

4

Machine Learning

Using this matrix, we can easily estimate the true-detection probability that we covered earlier in our hypothesis testing section,

>>> print(cmx.diagonal()/cmx.sum(axis=1)) [1. 0. 0.10714286 0.38333333 0.27272727 0.24137931 0.25252525 0.29230769 0.16 0. ]

In other words, the ﬁrst element is the probability of detecting 0 when 0 is in force, the second element is the probability of detecting 1 when 1 is in force, and so on. We can likewise compute the false-alarm rate for each of the classes in the following:

>>> print((cmx.sum(axis=0)-cmx.diagonal())/(cmx.sum()-cmx.sum(axis=1))) [0.01803607 0. 0.02330508 0.15909091 0.20199501 0.15885417 0.17955112 0.09195402 0.02105263 0.03219316]

Programming Tip

The Numpy sum function can sum across a particular axis or, if the axis is unspeciﬁed, will sum all entries of the array.

Inthiscase,theﬁrstelementistheprobabilitythat 0isdeclaredwhenanothercategory is in force, the next element is the probability that 1 is declared when another category is in force, and so on. For a decent classiﬁer, we want a true-detection probability to be greater than the corresponding false-alarm rate, otherwise the classiﬁer is no better than a coin-ﬂip.

The missing feature of this problem, from the learning algorithm standpoint, is that we did not supply the bit representation of every element which was used to derive the target variable, y. Instead, we just used the integer value of each of the 10bit numbers, which essentially concealed the mechanism for creating the y values. In other words, there was an unknown transformation from the input space X to Y that the learning algorithm had to overcome, but that it could not overcome, at least not without memorizing the training data. This lack of knowledge is a key issue in all machine learning problems, although we have made it explicit here with this stylized example. This means that there may be one or more transformations from X → X ′ that can help the learning algorithm get traction on the so-transformed space while providing a better trade-off between generalization and approximation than could have been achieved otherwise. Finding such transformations is called feature engineering.

4.3.4 Cross-Validation

In the last section, we explored a stylized machine learning example to understand the issues of complexity in machine learning. However, to get an estimate of out-ofsample errors, we simply generated more synthetic data. In practice, this is not an 4.3 Theory of Learning

257

option, so we need to estimate these errors from the training set itself. This is what cross-validation does. The simplest form of cross-validation is k-fold validation. For example, if K = 3, then the training data is divided into three sections wherein each of the three sections is used for testing and the remaining two are used for training. This is implemented in Scikit-learn as in the following:

>>> import numpy as np

>>> from sklearn.model\_selection import KFold

>>> data =np.array(['a',]\*3+['b',]\*3+['c',]\*3) # example

>>> print (data)

['a' 'a' 'a' 'b' 'b' 'b' 'c' 'c' 'c']

>>> kf = KFold(3)

>>> for train\_idx,test\_idx in kf.split(data): ... print (train\_idx,test\_idx) ...

[3 4 5 6 7 8] [0 1 2] [0 1 2 6 7 8] [3 4 5] [0 1 2 3 4 5] [6 7 8]

Inthecodeabove,weconstructasampledataarrayandthenseehowKFoldsplitsitup into indices for training and testing, respectively. Notice that there are no duplicated elements in each row between training and testing indices. To examine the elements of the dataset in each category, we simply use each of the indices as in the following:

>>> for train\_idx,test\_idx in kf.split(data): ... print('training', data[ train\_idx ]) ... print('testing' , data[ test\_idx ]) ...

training ['b' 'b' 'b' 'c' 'c' 'c'] testing ['a' 'a' 'a'] training ['a' 'a' 'a' 'c' 'c' 'c'] testing ['b' 'b' 'b'] training ['a' 'a' 'a' 'b' 'b' 'b'] testing ['c' 'c' 'c']

This shows how each group is used in turn for training/testing. There is no random shufﬂing of the data unless the shuffle keyword argument is given. The error over the test set is the cross-validation error. The idea is to postulate models of differing complexity and then pick the one with the best cross-validation error. For example, suppose we had the following sine wave data,

>>> xi = np.linspace(0,1,30)

>>> yi = np.sin(2\*np.pi\*xi)

and we want to ﬁt this with polynomials of increasing order.

Figure 4.11 shows the individual folds in each panel. The circles represent the training data. The diagonal line is the ﬁtted polynomial. The gray shaded areas indicate the regions of errors between the ﬁtted polynomial and the held-out testing 258

4

Machine Learning

Fig. 4.11 This shows the folds and errors for the linear model. The shaded areas show the errors in each respective test set (i.e., cross-validation scores) for the linear model

data. The larger the gray area, the bigger the cross-validation errors, as are reported in the title of each frame.

After reviewing the last four ﬁgures and averaging the cross-validation errors, the one with the least average error is declared the winner. Thus, cross-validation provides a method of using a single dataset to make claims about unseen out-ofsample data insofar as the model with the best complexity can be determined. The entire process to generate the above ﬁgures can be captured using cross\_val\_score as shown for the linear regression (compare the output with the values in the titles in each panel of Fig.4.11),

>>> from sklearn.metrics import make\_scorer, mean\_squared\_error

>>> from sklearn.model\_selection import cross\_val\_score

>>> from sklearn.linear\_model import LinearRegression

>>> Xi = xi.reshape(-1,1) # refit column-wise

>>> Yi = yi.reshape(-1,1)

>>> lf = LinearRegression()

>>> scores = cross\_val\_score(lf,Xi,Yi,cv=4,

... scoring=make\_scorer(mean\_squared\_error)) >>> print(scores) [0.3554451 0.33131438 0.50454257 0.45905672] 4.3 Theory of Learning

259

Programming Tip

The make\_scorer function is a wrapper that enables cross\_val\_score to compute scores from the given estimator’s output.

The process can be further automated by using a pipeline as in the following:

>>> from sklearn.pipeline import Pipeline

>>> from sklearn.preprocessing import PolynomialFeatures

>>> polyfitter = Pipeline([('poly', PolynomialFeatures(degree=3)), ... ('linear', LinearRegression())])

>>> polyfitter.get\_params()

{'memory': None, 'steps': [('poly', PolynomialFeatures(degree=3, include\_bias=True, interaction\_only=False)), ('linear', LinearRegression(copy\_X=True, fit\_intercept=True, n\_jobs=None,

normalize=False))], 'poly': PolynomialFeatures(degree=3,

include\_bias=True, interaction\_only=False), 'linear': LinearRegression(copy\_X=True, fit\_intercept=True, n\_jobs=None, normalize=False), 'poly\_\_degree': 3, 'poly\_\_include\_bias': True, 'poly\_\_interaction\_only': False, 'linear\_\_copy\_X':

True, 'linear\_\_fit\_intercept': True, 'linear\_\_n\_jobs': None, 'linear\_\_normalize': False}

The Pipeline object is a way of stacking standard steps into one big estimator, while respecting the usual fit and predict interfaces. The output of the get\_params function contains the polynomial degrees we previously looped over to create Fig.4.11, etc. We will use these named parameters in the next code block. To do this automatically using this polyfitter estimator, we need the Grid Search Cross Validation object, GridSearchCV. The next step is to use this to create the grid of parameters we want to loop over as in the following:

>>> from sklearn.model\_selection import GridSearchCV ... cv=4,return\_train\_score=True) The gs object will loop over the polynomial degrees up to cubic using fourfold cross-validation cv=4, like we did manually earlier. The poly\_\_degree item comes from the previous get\_params call. Now, we just apply the usual fit method on the training data,

>>> gs=GridSearchCV(polyfitter,{'poly\_\_degree':[1,2,3]},

>>> \_=gs.fit(Xi,Yi)

>>> gs.cv\_results\_

{'mean\_fit\_time': array([0.00041956, 0.00041848, 0.00043315]), 'std\_fit\_time': array([3.02347168e-05, 5.91589236e-06, 6.70625100e-06]), 'mean\_score\_time': array([0.00027096, 0.00027257, 0.00032073]), 'std\_score\_time': array([9.02611933e-06, 1.20837301e-06, 5.49008608e-05]),

'param\_poly\_\_degree': masked\_array(data=[1, 2, 3],

mask=[False, False, False],

fill\_value='?',

dtype=object), 'params': 260

4

Machine Learning

[{'poly\_\_degree': 1}, {'poly\_\_degree': 2}, {'poly\_\_degree': 3}], 'split0\_test\_score': array([ -2.03118491, -68.54947351, -1.64899934]), 'split1\_test\_score': array([-1.38557769, -3.20386236, 0.81372823]), 'split2\_test\_score': array([ -7.82417707, -11.8740862 , 0.47246476]), 'split3\_test\_score': array([ -3.21714294, -60.70054797, 0.14328163]), 'mean\_test\_score': array([ -3.4874447 , -36.06830421, -0.07906481]), 'std\_test\_score': array([ 2.47972092, 29.1121604 , 0.975868 ]), 'rank\_test\_score': array([2, 3, 1], dtype=int32), 'split0\_train\_score': array([0.52652515, 0.93434227, 0.99177894]), 'split1\_train\_score': array([0.5494882 , 0.60357784, 0.99154288]), 'split2\_train\_score': array([0.54132528, 0.59737218, 0.99046089]), 'split3\_train\_score': array([0.57837263, 0.91061274, 0.99144127]), 'mean\_train\_score': array([0.54892781, 0.76147626, 0.99130599]), 'std\_train\_score': array([0.01888775, 0.16123462, 0.00050307])}

the scores shown correspond to the cross-validation scores for each of the parameters (e.g., polynomial degrees) using fourfold cross-validation. Note that the higher scores are better here and the cubic polynomial is best, as we observed earlier. The default R 2 metric is used for the scoring in this case as opposed to mean squared error. The validation results of this pipeline for the quadratic ﬁt are shown in Fig.4.12, and for the cubic ﬁt, in Fig.4.13. This can be changed by passing the scoring=make\_scorer(mean\_squared\_error) keyword argument to GridSearchCV. There is also RandomizedSearchCV that does not necessarily evaluate every point on the grid and instead randomly samples the grid according to an input probability distribution. This is very useful for a large number of hyperparameters.

4.3.5 Bias and Variance

Considering average error in terms of in-samples and out-samples depends on a particular training data set. What we want is a concept that captures the performance of the estimator for all possible training data. For example, our ultimate estimator, f ˆ is derived from a particular set of training data (D) and is thus denoted, f ˆ D . This makes the out-of-sample error explicitly, E out ( f ˆ D ). To eliminate the dependence on a particular set of training dataset, we have to compute the expectation across all training datasets,

E D E out ( f ˆ D ) = bias + var

where

bias(x) = ( f ˆ (x) − f (x))2

and

var(x) = E D ( f ˆ D (x) − f ˆ (x))2 4.3 Theory of Learning

261

and where f ˆ is the mean of all estimators for all datasets. There is nothing to say that such a mean is an estimator that could have arisen from any particular training data, however. It just implies that for any particular point x, the mean of the values of all the estimators is f ˆ (x). Therefore, bias captures the sense that, even if all possible data were presented to the learning method, it would still differ from the target function by this amount. On the other hand, var shows the variation in the ﬁnal hypothesis, depending on the training data set, notwithstanding the target function. Thus, the tension between approximation and generalization is captured by these two terms. For example, suppose there is only one hypothesis. Then, var = 0 because there can be no variation due to a particular set of training data because no matter what that training data is, the learning method always selects the one and only hypothesis. In this case, the bias could be very large, because there is no opportunity for the learning method to alter the hypothesis due to the training data, and the method can only ever pick the single hypothesis!

Let’sconstructanexampletomakethisconcrete.Supposewehaveahypothesisset consisting of all linear regressions without an intercept term, h(x) = ax. The training data consists of only two points { (x i , sin(πx i )) } i=1 2 where x i is drawn uniformly from the interval [ −1, 1 ] . From Sect.3.7 on linear regression, we know that the solution for a is the following:

xT y a= xT x

(4.3.5.1)

Fig. 4.12 This shows the folds and errors as in Figs.4.10 and 4.11. The shaded areas show the errors in each respective test set for the quadratic model 262

4

Machine Learning

Fig. 4.13 This shows the folds and errors. The shaded areas show the errors in each respective test set for the cubic model

where x = [ x 1 ] and, x 2 y = [ y 1 , y 2 ] . The f ˆ (x) represents the solution over all possible sets of training data for a ﬁxed x. The following code shows how to construct the training data,

>>> from scipy import stats

>>> def gen\_sindata(n=2):

... ... ... ... ...

x=stats.uniform(-1,2) # define random variable v = x.rvs((n,1)) # generate sample # use sample for sine return (v,y)

y

=

np.sin(np.pi\*v)

Again,usingScikit-learn’sLinearRegressionobject,wecancomputethea parameter. Note that we have to set fit\_intercept=False keyword to suppress the default automatic ﬁtting of the intercept.

>>> lr = LinearRegression(fit\_intercept=False)

>>> lr.fit(\*gen\_sindata(2))

LinearRegression(copy\_X=True, fit\_intercept=False, n\_jobs=None, normalize=False)

>>> lr.coef\_

array([[0.24974914]]) 4.3 Theory of Learning

263

Fig. 4.14 For a two-element training set consisting of the points shown, the line is the best ﬁt over the hypothesis set,

h(x) = ax

Programming Tip

Note that we designed gen\_sindata to return a tuple to use the automatic unpacking feature of Python functions in lr.fit(\*gen\_sindata()). In other words, using the asterisk notation means we don’t have to separately assign the outputs of gen\_sindata before using them for lr.fit.

In this case, f ˆ (x) = ax, where a the expected value of the parameter over all possible training datasets. Using our knowledge of probability, we can write this out explicitly as the following (Fig.4.14):

x 1 sin(πx 1 ) + x 2 sin(πx 2 ) a=E ( x 1 2 + x 2 2 )

where x = [ x 1 , x 2 ] and y = [ sin(πx 1 ), sin(πx 2 ) ] in Eq.(4.3.5.1). However, computing this expectation analytically is hard, but for this speciﬁc situation, a ≈ 1.43. To get this value using simulation, we just loop over the process, collect the outputs, and the average them as in the following:

>>> a\_out=[] # output container

>>> for i in range(100):

... \_=lr.fit(\*gen\_sindata(2)) ... a\_out.append(lr.coef\_[0,0]) ...

>>> np.mean(a\_out) # approx 1.43

1.5476180748170179 264

4

Machine Learning

Fig. 4.15 These curves decompose the mean squared error into its constituent bias and variance for this example

Note that you may have to loop over many more iterations to get close to the purported value. The var requires the variance of a,

var(x) = E((a − a)x) 2 = x 2 E(a − a) 2 ≈ 0.71x2

The bias is the following:

bias(x) = (sin(πx) − ax)2 Figure 4.15 shows the bias, var, and mean squared error for this problem. Notice that there is zero bias and zero variance when x = 0. This is because the learning method cannot help but get that correct because all the hypotheses happen to match the value of the target function at that point! Likewise, the var is zero because all possible pairs, which constitute the training data, are ﬁtted through zero because h(x) = ax has no choice but to go through zero. The errors are worse at the end points. As we discussed in our statistics chapter, those points have the most leverage against the hypothesized models and result in the worst errors. Notice that reducing the edge-errors depends on getting exactly those points near the edges as training data. The sensitivity to a particular dataset is reﬂected in this behavior.

What if we had more than two points in the training data? What would happen to var and bias? Certainly, the var would decrease because it would be harder and harder to generate training datasets that would be substantially different from each other. The bias would also decrease because more points in the training data means better approximation of the sine function over the interval. What would happen if we changed the hypothesis set to include more complex polynomials? As we have already seen with our polynomial regression earlier in this chapter, we would see the same overall effect as here, but with relatively smaller absolute errors and the same edge effects we noted earlier. 4.3 Theory of Learning

265

4.3.6 Learning Noise

We have thus far not considered the effect of noise in our analysis of learning. The following example should help resolve this. Let’s suppose we have the following scalar target function,

y(x) = w o T x + η

where η ∼ N(0, σ 2 ) is an additive noise term and w, x ∈ R d . Furthermore, we have n measurements of y. This means the training set consists of { (x i , y i ) } i=1 n . Stacking the measurements together into a vector format,

y = Xw o + η

with y, η ∈ R n , w o ∈ R d and X contains x i as columns. The hypothesis set consists of all linear models,

h(w, x) = w T x

We need to learn the correct w from the hypothesis set given the training data. So far, this is the usual setup for the problem, but how does the noise factor play to this? In our usual situation, the training set consists of randomly chosen elements from a larger space. In this case, that would be the same as getting random sets of x i vectors. That still happens in this case, but the problem is that even if the same x i appears twice, it will not be associated with the same y value due to the additive noise coming from η . To keep this simple, we assume that there is a ﬁxed set of xi vectors and that we get all of them in the training set. For every speciﬁc training set, we know how to solve for the MMSE from our earlier statistics work,

w = (X T X) −1 X T y

Given this setup, what is the in-sample mean squared error? Because this is the MMSE solution, we know from our study of the associated orthogonality of such systems that we have,

E in = ∥ y ∥ 2 − ∥ Xw ∥2

(4.3.6.1)

where our best hypothesis, h = Xw. Now, we want to compute the expectation of this over the distribution of η . For instance, for the ﬁrst term, we want to compute,

1 1 E | y | 2 = y) = ) E(yT Tr E(yyT n n 266

4

Machine Learning

where Tr is the matrix trace operator (i.e., sum of the diagonal elements). Because each η is independent, we have

Tr E(yy T ) = Tr Xw o w o T X T + σ 2 Tr I = Tr Xw o w o T X T + nσ2

(4.3.6.2)

where I is the n × n identity matrix. For the second term in Eq.(4.3.6.1), we have

2 | Xw | = Tr Xww T X T = Tr X(X T X) −1 X T yy T X(X T X) −1 XT

The expectation of this is the following:

E | Xw | 2 = Tr X(X T X) −1 X T E(yy T )X(X T X) −1 XT

(4.3.6.3)

which, after substituting in Eq.(4.3.6.2), yields,

E | Xw | 2 = Tr Xw o w o T X T + σ 2 d

(4.3.6.4)

Next, assembling Eq.(4.3.6.1) from this and Eq.(4.3.6.2) gives

1 d E(E in ) = = σ 2 1 Ein n ( n )

(4.3.6.5)

which provides an explicit relationship between the noise power, σ 2 , the complexity of the method (d) and the number of training samples (n). This is very illustrative because it reveals the ratio d/n, which is a statement of the trade-off between model complexity and in-sample data size. From our analysis of the VC dimension, we already know that there is a complicated bound that represents the penalty for complexity, but this problem is unusual in that we can actually derive an expression for this without resorting to bounding arguments. Furthermore, this result shows, that with a very large number of training examples (n → ∞), the expected in-sample error approaches σ 2 . Informally, this means that the learning method cannot generalize from noise and thus can only reduce the expected in-sample error by memorizing the data (i.e., d ≈ n). The corresponding analysis for the expected out-of-sample error is similar, but more complicated because we don’t have the orthogonality condition. Also, the outof-sample data has different noise from that used to derive the weights, w. This results in extra cross-terms,

E out = Tr Xw o w o T X T + ξξ T + Xww T X T − Xww o T XT (

−Xw o w T XT )

(4.3.6.6) 4.3 Theory of Learning

267

where we are using the ξ notation for the noise in the out-of-sample case, which is different from that in the in-sample case. Simplifying this leads to the following:

E(E out ) = Tr σ 2 I + σ 2 X(X T X) −1 XT

(4.3.6.7)

Then, assembling all of this gives

d E(E out ) = σ 2 1 + ( n )

(4.3.6.8)

which shows that even in the limit of large n, the expected out-of-sample error also approaches the noise power limit, σ 2 . This shows that memorizing the in-sample data (i.e., d/n ≈ 1) imposes a proportionate penalty on the out-of-sample performance (i.e., EE out ≈ 2σ 2 when EE in ≈ 0). The following code simulates this important example:

>>> def est\_errors(d=3,n=10,niter=100):

... assert n>d ... wo = np.matrix(arange(d)).T ... Ein = list() ... Eout = list() ... # choose any set of vectors ... X = np.matrix(np.random.rand(n,d)) ... for ni in range(niter):

...

y

=

X\*wo

+

np.random.randn(X.shape[0],1)

... # training weights ...

w

h

=

=

np.linalg.inv(X.T\*X)\*X.T\*y

...

X\*w

... Ein.append(np.linalg.norm(h-y)\*\*2) ... # out of sample error ...

yp

=

X\*wo

+

np.random.randn(X.shape[0],1)

... Eout.append(np.linalg.norm(h-yp)\*\*2) ... return (np.mean(Ein)/n,np.mean(Eout)/n) ...

Programming Tip

Python has an assert statement to make sure that certain entry conditions for the variables in the function are satisﬁed. It is a good practice to use reasonable assertions at entry and exit to improve the quality of code.

The following runs the simulation for the given value of d. 268

4

Machine Learning

Fig. 4.16 The dots show the learning curves estimated from the simulation and the solid lines show the corresponding terms for our analytical result. The horizontal line shows the variance of the additive noise (σ 2 = 1 in this case). Both the expected in-sample and out-of-sample errors asymptotically approach this line

>>> d=10

>>> xi = arange(d\*2,d\*10,d//2)

>>> ei,eo=np.array([est\_errors(d=d,n=n,niter=100) for n in xi]).T

which results in Fig.4.16. This ﬁgure shows the estimated expected in-sample and out-of-sample errors from our simulation compared with our corresponding analytical result. The heavy horizontal line shows the variance of the additive noise σ 2 = 1. Both these curves approach this asymptote because the noise is the ultimate learning limit for this problem. For a given dimension d, even with an inﬁnite amount of training data, the learning method cannot generalize beyond the limit of the noise power. Thus, the expected generalization error is E(E out ) − E(E in ) = 2σ n 2 d .

4.4 Decision Trees

Adecisiontreeistheeasiestclassiﬁertounderstand,interpret,andexplain.Adecision tree is constructed by recursively splitting the dataset into a sequence of subsets based on if-then questions. The training set consists of pairs (x, y) where x ∈ R d where d is the number of features available and where y is the corresponding label. The learning method splits the training set into groups based on x while attempting to keep the assignments in each group as uniform as possible. In order to do this, the learning method must pick a feature and an associated threshold for that feature upon which to divide the data. This is tricky to explain in words, but easy to see with an example. First, let’s set up the Scikit-learn classiﬁer, 4.4 Decision Trees

269

>>> from sklearn import tree

>>> clf = tree.DecisionTreeClassifier()

Let’s also create some example data,

>>> import numpy as np

>>> M=np.fromfunction(lambda i,j:j>=2,(4,4)).astype(int)

>>> print(M) [[0 0 1 1] [0 0 1 1] [0 0 1 1] [0 0 1 1]]

Programming Tip

The fromfunction creates Numpy arrays using the indices as inputs to a function whose value is the corresponding array entry.

We want to classify the elements of the matrix based on their respective positions in the matrix. By just looking at the matrix, the classiﬁcation is pretty simple—classify as 0 for any positions in the ﬁrst two columns of the matrix, and classify 1 otherwise. Let’s walk through this formally and see if this solution emerges from the decision tree. The values of the array are the labels for the training set and the indices of those values are the elements of x. Speciﬁcally, the training set has X = { (i, j) } and Y = { 0, 1 } Now, let’s extract those elements and construct the training set.

>>> i,j = np.where(M==0)

>>> x=np.vstack([i,j]).T # build nsamp by nfeatures

>>> y = j.reshape(-1,1)\*0 # 0 elements

>>> print(x) [[0 0] [0 1] [1 0] [1 1] [2 0] [2 1] [3 0] [3 1]]

>>> print(y) [[0]

[0]

[0]

[0]

[0]

[0] 270

4

Machine Learning

[0]

[0]]

Thus, the elements of x are the two-dimensional indices of the values of y. For example, M[x[0,0],x[0,1]]=y[0,0]. Likewise, to complete the training set, we just need to stack the rest of the data to cover all the cases,

>>> i,j = np.where(M==1)

>>> x=np.vstack([np.vstack([i,j]).T,x ]) # build nsamp x nfeatures

>>> y=np.vstack([j.reshape(-1,1)\*0+1,y]) # 1 elements

With all that established, all we have to do is train the classiﬁer:

>>> clf.fit(x,y) DecisionTreeClassifier(class\_weight=None, criterion='gini', max\_depth=None, max\_features=None, max\_leaf\_nodes=None, min\_impurity\_decrease=0.0, min\_impurity\_split=None,

min\_samples\_leaf=1, min\_samples\_split=2,

min\_weight\_fraction\_leaf=0.0, presort=False, random\_state=None, splitter='best')

To evaluate how the classiﬁer performed, we can report the score,

>>> clf.score(x,y) 1.0

For this classiﬁer, the score is the accuracy, which is deﬁned as the ratio of the sum of the true-positive (T P) and true-negatives (T N) divided by the sum of all the terms, including the false terms,

TP + TN accuracy = T P + T N + FN + FP

In this case, the classiﬁer gets every point correctly, so F N = F P = 0. On a related note, two other common names from information retrieval theory are recall (a.k.a. sensitivity) and precision (a.k.a. positive predictive value, T P/(T P + F P)). We can visualize this tree in Fig.4.17. The Gini coefﬁcients (a.k.a. categorical variance) in the ﬁgure are a measure of the purity of each so-determined class. This coefﬁcient is deﬁned as,

Gini m = ∑ p m,k (1 − p m,k ) k

where

1 p m,k = ∑ I (y i = k) Nm x i ∈R m

which is the proportion of observations labeled k in the m th node and I (·) is the usual indicator function. Note that the maximum value of the Gini coefﬁcient is 4.4 Decision Trees

271

Fig. 4.17 Example decision tree. The Gini coefﬁcient in each branch measures the purity of the partition in each node. The samples item in the box shows the number of items in the corresponding node in the decision tree

max Gini m = 1 − 1/m. For our simple example, half of the sixteen samples are in category 0 and the other half are in the 1 category. Using the notation above, the top box corresponds to the 0 th node, so p 0,0 = 1/2 = p 0,1 . Then, Gini 0 = 0.5. The next layer of nodes in Fig.4.17 is determined by whether or not the second dimension of the x data is greater than 1.5. The Gini coefﬁcients for each of these child nodes is zero because after the prior split, each subsequent category is pure. The value list in each of the nodes shows the distribution of elements in each category at each node. To make this example more interesting, we can contaminate the data slightly,

>>> M[1,0]=1 # put in different class

>>> print(M) # now contaminated [[0 0 1 1] [1 0 1 1] [0 0 1 1] [0 0 1 1]]

Now we have a 1 entry in the previously pure ﬁrst column’s second row. Let’s re-do the analysis as in the following:

>>> i,j = np.where(M==0)

>>> x=np.vstack([i,j]).T

>>> y = j.reshape(-1,1)\*0

>>> i,j = np.where(M==1)

>>> x=np.vstack([np.vstack([i,j]).T,x])

>>> y = np.vstack([j.reshape(-1,1)\*0+1,y])

>>> clf.fit(x,y) DecisionTreeClassifier(class\_weight=None, criterion='gini', max\_depth=None, max\_features=None, max\_leaf\_nodes=None, min\_impurity\_decrease=0.0, min\_impurity\_split=None,

min\_samples\_leaf=1, min\_samples\_split=2,

min\_weight\_fraction\_leaf=0.0, presort=False, random\_state=None, splitter='best')

The result is shown in Fig.4.18. Note the tree has grown signiﬁcantly due to this one change! The 0 th node has the following parameters, p 0,0 = 7/16 and p 0,1 = 9/16. This makes the Gini coefﬁcient for the 0 th node equal to 16 7 ( 1 − 16 7 ) + 16 9 (1− 16 9 ) = 0.492. As before, the root node splits on X [ 1 ] ≤ 1.5. Let’s see if we can reconstruct the succeeding layer of nodes manually, as in the following: 272

4

Machine Learning

>>> y[x[:,1]>1.5] # first node on the right array([[1],

[1],

[1],

[1],

[1],

[1],

[1],

[1]])

This obviously has a zero Gini coefﬁcient. Likewise, the node on the left contains the following:

>>> y[x[:,1]<=1.5] # first node on the left array([[1],

[0],

[0],

[0],

[0],

[0],

[0],

[0]])

The Gini coefﬁcient in this case is computed as (1/8)\*(1-1/8)+(7/8)\* (1-7/8)=0.21875. This node splits based on X[1]<0.5. The child node to the right derives from the following equivalent logic,

>>> np.logical\_and(x[:,1]<=1.5,x[:,1]>0.5) array([False, False, False, False, False, False, False, False, False, False, True, True, False, True, False, True])

with corresponding classes,

>>> y[np.logical\_and(x[:,1]<=1.5,x[:,1]>0.5)] array([[0],

[0],

[0],

[0]])

Programming Tip

The logical\_and in Numpy provides element-wise logical conjunction. It is not possible to accomplish this with something like 0.5< x[:,1] <=1.5 because of the way Python parses this syntax. 4.4 Decision Trees

273

Fig. 4.18 Decision tree for contaminated data. Note that just one change in the training data caused the tree to grow ﬁve times as large as before!

Notice that for this example as well as for the previous one, the decision tree was exactly able to memorize (overﬁt) the data with perfect accuracy. From our discussion of machine learning theory, this is an indication of potential problems in generalization.

The key step in building the decision tree is to come up with the initial split. There are a number of algorithms that can build decision trees based on different criteria, but the general idea is to control the information entropy as the tree is developed. In practical terms, this means that the algorithms attempt to build trees that are not excessively deep. It is well established that this is a very hard problem to solve completely and there are many approaches to it. This is because the algorithms must make global decisions at each node of the tree using the local data available up to that point.

For this example, the decision tree partitions the X space into different regions corresponding to different Y labels as shown in Fig.4.19. The root node at the top of Fig.4.18 splits the input data based on X [ 1 ] ≤ 1.5. This corresponds to the top left panel in Fig.4.19 (i.e., node 0) where the vertical line divides the training data shown into two regions, corresponding to the two subsequent child nodes. The next split happens with X [ 1 ] ≤ 0.5 as shown in the next panel of Fig.4.19 titled node 1. This continues until the last panel on the lower right, where the contaminated element we injected has been isolated into its own sub-region. Thus, the last panel is a representation of Fig.4.18, where the horizontal/vertical lines correspond to successive splits in the decision tree.

Figure 4.20 shows another example, but now using a simple triangular matrix. As shown by the number of vertical and horizontal partitioning lines, the decision 274

4

Machine Learning

Fig. 4.19 The decision tree divides the training set into regions by splitting successively along each dimension until each region is as pure as possible

Fig. 4.20 The decision tree ﬁtted to this triangular matrix is very complex, as shown by the number of horizontal and vertical partitions. Thus, even though the pattern in the training data is visually clear, the decision tree cannot automatically uncover it

tree that corresponds to this ﬁgure is tall and complex. Notice that if we apply a simple rotational transform to the training data, we can obtain Fig.4.21, which requires a trivial decision tree to ﬁt. Thus, there may be transformations of the training data that simplify the decision tree, but these are very difﬁcult to derive in general. Nonetheless, this highlights a key weakness of decision trees wherein they may be easy to understand, to train, and to deploy, but may be completely blind to such time-saving and complexity-saving transformations. Indeed, in higher dimensions, 4.4 Decision Trees

275

Fig. 4.21 Using a simple rotation on the training data in Fig.4.20, the decision tree can now easily ﬁt the training data with a single partition

it may be impossible to even visualize the potential of such latent transformations. Thus, the advantages of decision trees can be easily outmatched by other methods that we will study later that do have the ability to uncover useful transformations, but which will necessarily be harder to train. Another disadvantage is that because of how decision trees are built, even a single misplaced data point can cause the tree to grow very differently. This is a symptom of high variance.

In all of our examples, the decision tree was able to memorize the training data exactly, as we discussed earlier, this is a sign of potentially high generalization errors. There are pruning algorithms that strategically remove some of the deepest nodes. but these are not yet fully implemented in Scikit-learn, as of this writing. Alternatively, restricting the maximum depth of the decision tree can have a similar effect. The DecisionTreeClassifier and DecisionTreeRegressor in Scikit-learn both have keyword arguments that specify maximum depth.

4.4.1 Random Forests

It is possible to combine a set of decision trees into a larger composite tree that has better performance than its individual components by using ensemble learning. This is implemented in Scikit-learn as RandomForestClassifier. The composite tree helps mitigate the primary weakness of decision trees—high variance. Random forest classiﬁers help by averaging out the predictions of many constituent trees to minimize this variance by randomly selecting subsets of the training set to train the embedded trees. On the other hand, this randomization can increase bias because there may be a subset of the training set that yields an excellent decision tree, but the averaging effect over randomized training samples washes this out in the same averaging that reduces the variance. This is a key trade-off. The following code implements a simple random forest classiﬁer from our last example. 276

4

Machine Learning

Fig. 4.22 The constituent decision trees of the random forestandhowtheypartitioned the training set are shown in these four panels. The random forest classiﬁer uses the individual outputs of each of the constituent trees to produce a collaborative ﬁnal estimate

>>> from sklearn.ensemble import RandomForestClassifier

>>> rfc = RandomForestClassifier(n\_estimators=4,max\_depth=2)

>>> rfc.fit(X\_train,y\_train.flat)

RandomForestClassifier(bootstrap=True, class\_weight=None, criterion='gini', min\_impurity\_decrease=0.0, min\_impurity\_split=None,

max\_depth=2, max\_features='auto', max\_leaf\_nodes=None,

min\_samples\_leaf=1, min\_samples\_split=2,

min\_weight\_fraction\_leaf=0.0, n\_estimators=4, n\_jobs=None,

oob\_score=False, random\_state=None, verbose=0, warm\_start=False)

Note that we have constrained the maximum depth max\_depth=2 to help with generalization. To keep things simple we have only set up a forest with four individual classiﬁers. 3 Figure 4.22 shows the individual classiﬁers in the forest that have been trained above. Even though all the constituent decision trees share the same training data, the random forest algorithm randomly picks feature subsets (with replacement) upon which to train individual trees. This helps avoid the tendency of decision trees to become too deep and lopsided, which hurts both performance and generalization. At the prediction step, the individual outputs of each of the constituent decision trees are put to a majority vote for the ﬁnal classiﬁcation. To estimate generalization errors without using cross-validation, the training elements not used for a particular constituent tree can be used to test that tree and form a collaborative estimate of generalization errors. This is called the out-of-bag estimate.

The main advantage of random forest classiﬁers is that they require very little tuning and provide a way to trade-off bias and variance via averaging and random-

3 We have also set the random seed to a ﬁxed value to make the ﬁgures reproducible in the Jupyter Notebook corresponding to this section. 4.4 Decision Trees

277

ization. Furthermore, they are fast and easy to train in parallel (see the n\_jobs keyword argument) and fast to predict. On the downside, they are less interpretable than simple decision trees. There are many other powerful tree methods in Scikitlearn like ExtraTrees and Gradient Boosted Regression Trees GradientBoosting Regressor which are discussed in the online documentation.

4.4.2 Boosting Trees

To understand additive modeling using trees, recall the Gram–Schmidt orthogonalization procedure for vectors. The purpose of this orthogonalization procedure is to create an orthogonal set of vectors starting with a given vector u 1 . We have already discussed the projection operator in Sect.2.2. The Gram–Schmidt orthogonalization procedure starts with a vector v 1 , which we deﬁne as the following:

u1 =v1

with the corresponding projection operator proj u 1 . The next step in the procedure is to remove the residual of u 1 from v 2 , as in the following:

u 2 = v 2 − proj u 1 (v 2 )

This procedure continues for v 3 as in the following:

u 3 = v 3 − proj u 1 (v 3 ) − proj u 2 (v 3 )

and so on. The important aspect of this procedure is that new incoming vectors (i.e., v k ) are stripped of any preexisting components already present in the set of { u 1 , u 2 , . . . , u M } . Note that this procedure is sequential. That is, the order of the incoming v i matters. 4 Thus, any new vector can be expressed using the so-constructed { u 1 , u 2 , . . . , u M } basis set, as in the following:

x = ∑ α i ui

The idea behind additive trees is to reproduce this procedure for trees instead of vectors. There are many natural topological and algebraic properties that we lack for the general problem, however. For example, we already have well-established methods for measuring distances between vectors for the Gram–Schmidt procedure outlined above (namely, the L 2 distance), which we lack here. Thus, we need the concept of loss function, which is a way of measuring how well the process is working out at eachsequential step. This loss functionis parameterizedbythetrainingdataand

4 At least up to a rotation of the resulting orthonormal basis. 278

4

Machine Learning

by the classiﬁcation function under consideration: L y ( f (x)). For example, if we want a classiﬁer ( f ) that selects the label y i based upon the input data x i ( f : x i → y i ), then the squared error loss function would be the following:

L y ( f (x)) = ∑ (y i − f (x i ))2 i

We represent the classiﬁer in terms of a set of basis trees:

f (x) = ∑ α k u x (θ k ) k

The general algorithm for forward stage-wise additive modeling is the following:

• Initialize f (x) = 0

• For m = 1 to m = M, compute the following:

( β m , γ m ) = arg min ∑ L(y i , f m−1 (x i ) + β b(x i ; γ )) β , γ i

• Set f m (x) = f m−1 (x) + β m b(x ; γ m )

The key point is that the residuals from the prior step are used to ﬁt the basis function for the subsequent iteration. That is, the following equation is being sequentially approximated.

f m (x) − f m−1 (x) = β m b(x i ; γ m )

Let’s see how this works for decision trees and the exponential loss function.

L(x, f (x)) = exp(−y f (x))

Recall that for the classiﬁcation problem, y ∈ { −1, 1 } . For AdaBoost, the basis functions are the individual classiﬁers, G m (x) ↦ → { −1, 1 } The key step in the algorithm is the minimization step for the objective function

J( β , G) = ∑ exp(y i ( f m−1 (x i ) + β G(x i ))) i

( β m , G m ) = arg min ∑ exp(y i ( f m−1 (x i ) + β G(x i ))) β ,G i

Now, because of the exponential, we can factor out the following:

(m) w i = exp(y i f m−1 (x i )) 4.4 Decision Trees

279

as a weight on each data element and re-write the objective function as the following:

J( β , G) = ∑ w i (m) exp(y i β G(x i )) i

The important observation here is that y i G(x i ) ↦ → 1 if the tree classiﬁes x i correctly and y i G(x i ) ↦ → −1 otherwise. Thus, the above sum has terms like the following:

J( β , G) = ∑ w i (m) exp(− β ) + ∑ w i (m) exp( β ) y i ̸ =G(x i ) y i =G(x i )

For β > 0, this means that the best G(x) is the one that incorrectly classiﬁes for the largest weights. Thus, the minimizer is the following:

G m = arg min ∑ w i (m) I (y i = ̸ G(x i )) G i

where I is the indicator function (i.e., I (True) = 1, I (False) = 0). For β > 0, we can re-write the objective function as the following:

J = (exp( β ) − exp(− β )) ∑ w i (m) I (y i = ̸ G(x i )) + exp(− β ) ∑ wi (m) i i

and substitute θ = exp(− β ) so that

J 1 = + θ θ m ∑ i w i (m) ( θ )

(4.4.2.1)

where

w i (m) I (y i = ̸ G(x i )) ∑i m = ∑ i wi (m)

is the error rate of the classiﬁer with 0 ≤ m ≤ 1. Now, ﬁnding β is a straightforward calculus minimization exercise on the right side of Eq.(4.5.1.1), which gives the following:

1 1 − m = β m log 2 m

Importantly, β m canbecomenegativeif m < 1 2 ,whichwouldviolateourassumptions on β . This is captured in the requirement that the base learner be better than just random guessing, which would correspond to m > 2 1 . Practically speaking, this 280

4

Machine Learning

means that boosting cannot ﬁx a base learner that is no better than a random guess. Formally speaking, this is known as the empirical weak learning assumption [3].

Now we can move to the iterative weight update. Recall that

w i (m + 1) = exp(y i f m (x i )) = w i (m) exp(y i β m G m (x i ))

which we can re-write as the following:

w i (m + 1) = w i (m) exp( β m ) exp(−2 β m I (G m (x i ) = y i ))

This means that the data elements that are incorrectly classiﬁed have their corresponding weights increased by exp( β m ) and those that are correctly classiﬁed have their corresponding weights reduced by exp(− β m ). The reason for the choice of the exponential loss function comes from the following:

1 P(Y = 1 | x) f ∗ (x) = arg min E Y | x (exp(−Y f (x))) = log f (x) 2 P(Y = –1 | x)

This means that boosting is approximating a f (x) that is actually half the log-odds of the conditional class probabilities. This can be rearranged as the following

1 P(Y = 1 | x) = 1 + exp(–2 f ∗ (x))

The important beneﬁt of this general formulation for boosting, as a sequence of additive approximations, is that it opens the door to other choices of loss function, especially loss functions that are based on robust statistics that can account for errors in the training data (c.f. Hastie).

Gradient Boosting. Given a differentiable loss function, the optimization process can be formulated using numerical gradients. The fundamental idea is to treat the f (x i ) as a scalar parameter to be optimized over. Generally speaking, we can think of the following loss function,

N L( f ) = ∑ L(y i , f (x i )) i=1

as a vectorized quantity

f = { f (x 1 ), f (x 2 ), . . . , f (x N ) }

so that the optimization is over this vector

f ˆ = arg min L(f)

f 4.4 Decision Trees

281

With this general formulation we can use numerical optimization methods to solve for the optimal f as a sum of component vectors as in the following:

M f M = ∑ hm m=0

Note that this leaves aside the prior assumption that f is parameterized as a sum of individual decision trees.

, f (x i )) ∂L(yi gi,m = [ ∂ f (x i ) ] f (x i )= f m−1 (x i )

4.5 Boosting Trees

4.5.1 Boosting Trees

To understand additive modeling using trees, recall the Gram–Schmidt orthogonalization procedure for vectors. The purpose of this orthogonalization procedure is to create an orthogonal set of vectors starting with a given vector u 1 . We have already discussed the projection operator in Sect.2.2. The Gram–Schmidt orthogonalization procedure starts with a vector v 1 , which we deﬁne as the following:

u1 =v1

with the corresponding projection operator proj u 1 . The next step in the procedure is to remove the residual of u 1 from v 2 , as in the following:

u 2 = v 2 − proj u 1 (v 2 )

This procedure continues for v 3 as in the following:

u 3 = v 3 − proj u 1 (v 3 ) − proj u 2 (v 3 )

and so on. The important aspect of this procedure is that new incoming vectors (i.e., v k ) are stripped of any preexisting components already present in the set of { u 1 , u 2 , . . . , u M } . Note that this procedure is sequential. That is, the order of the incoming v i matters. 5 Thus, any new vector can be expressed using the so-constructed { u 1 , u 2 , . . . , u M } basis set, as in the following:

5 At least up to a rotation of the resulting orthonormal basis. 282

4

Machine Learning

x = ∑ α i ui

The idea behind additive trees is to reproduce this procedure for trees instead of vectors. There are many natural topological and algebraic properties that we lack for the general problem, however. For example, we already have well-established methods for measuring distances between vectors for the Gram–Schmidt procedure outlined above (namely, the L 2 distance), which we lack here. Thus, we need the concept of loss function, which is a way of measuring how well the process is working out at eachsequential step. This loss functionis parameterizedbythetrainingdataand by the classiﬁcation function under consideration: L y ( f (x)). For example, if we want a classiﬁer ( f ) that selects the label y i based upon the input data x i ( f : x i → y i ), then the squared error loss function would be the following:

L y ( f (x)) = ∑ (y i − f (x i ))2 i

We represent the classiﬁer in terms of a set of basis trees:

f (x) = ∑ α k u x (θ k ) k

The general algorithm for forward stage-wise additive modeling is the following:

• Initialize f (x) = 0

• For m = 1 to m = M, compute the following:

( β m , γ m ) = arg min ∑ L(y i , f m−1 (x i ) + β b(x i ; γ )) β , γ i

• Set f m (x) = f m−1 (x) + β m b(x ; γ m )

The key point is that the residuals from the prior step are used to ﬁt the basis function for the subsequent iteration. That is, the following equation is being sequentially approximated.

f m (x) − f m−1 (x) = β m b(x i ; γ m )

Let’s see how this works for decision trees and the exponential loss function.

L(x, f (x)) = exp(−y f (x))

Recall that for the classiﬁcation problem, y ∈ { −1, 1 } . For AdaBoost, the basis functions are the individual classiﬁers, G m (x) ↦ → { −1, 1 } The key step in the algorithm is the minimization step for the objective function 4.5 Boosting Trees

283

J( β , G) = ∑ exp(y i ( f m−1 (x i ) + β G(x i ))) i

( β m , G m ) = arg min ∑ exp(y i ( f m−1 (x i ) + β G(x i ))) β ,G i

Now, because of the exponential, we can factor out the following:

w i (m) = exp(y i f m−1 (x i ))

as a weight on each data element and re-write the objective function as the following:

J( β , G) = ∑ w i (m) exp(y i β G(x i )) i

The important observation here is that y i G(x i ) ↦ → 1 if the tree classiﬁes x i correctly and y i G(x i ) ↦ → −1 otherwise. Thus, the above sum has terms like the following:

J( β , G) = ∑ w i (m) exp(− β ) + ∑ w i (m) exp( β ) y i ̸ =G(x i ) y i =G(x i )

For β > 0, this means that the best G(x) is the one that incorrectly classiﬁes for the largest weights. Thus, the minimizer is the following:

G m = arg min ∑ w i (m) I (y i = ̸ G(x i )) G i

where I is the indicator function (i.e., I (True) = 1, I (False) = 0).

For β > 0, we can re-write the objective function as the following:

J = (exp( β ) − exp(− β )) ∑ w i (m) I (y i = ̸ G(x i )) + exp(− β ) ∑ wi (m) i i

and substitute θ = exp(− β ) so that

J 1 = + θ θ m ∑ i w i (m) ( θ )

(4.5.1.1)

where

w i (m) I (y i = ̸ G(x i )) ∑i m = ∑ i wi (m) 284

4

Machine Learning

is the error rate of the classiﬁer with 0 ≤ m ≤ 1. Now, ﬁnding β is a straightforward calculus minimization exercise on the right side of Eq.(4.5.1.1), which gives the following:

1 1 − m = β m log 2 m

Importantly, β m canbecomenegativeif m < 1 2 ,whichwouldviolateourassumptions on β . This is captured in the requirement that the base learner be better than just random guessing, which would correspond to m > 2 1 . Practically speaking, this means that boosting cannot ﬁx a base learner that is no better than a random guess. Formally speaking, this is known as the empirical weak learning assumption [3].

Now we can move to the iterative weight update. Recall that

w i (m + 1) = exp(y i f m (x i )) = w i (m) exp(y i β m G m (x i ))

which we can re-write as the following:

w i (m + 1) = w i (m) exp( β m ) exp(−2 β m I (G m (x i ) = y i ))

This means that the data elements that are incorrectly classiﬁed have their corresponding weights increased by exp( β m ) and those that are correctly classiﬁed have their corresponding weights reduced by exp(− β m ). The reason for the choice of the exponential loss function comes from the following:

1 P(Y = 1 | x) f ∗ (x) = arg min E Y | x (exp(−Y f (x))) = log f (x) 2 P(Y = –1 | x)

This means that boosting is approximating a f (x) that is actually half the log-odds of the conditional class probabilities. This can be rearranged as the following

1 P(Y = 1 | x) = 1 + exp(–2 f ∗ (x))

The important beneﬁt of this general formulation for boosting, as a sequence of additive approximations, is that it opens the door to other choices of loss function, especially loss functions that are based on robust statistics that can account for errors in the training data (c.f. Hastie).

Gradient Boosting. Given a differentiable loss function, the optimization process can be formulated using numerical gradients. The fundamental idea is to treat the f (x i ) as a scalar parameter to be optimized over. Generally speaking, we can think of the following loss function,

N L( f ) = ∑ L(y i , f (x i )) i=1 4.5 Boosting Trees

285

as a vectorized quantity

f = { f (x 1 ), f (x 2 ), . . . , f (x N ) }

so that the optimization is over this vector

f ˆ = arg min L(f)

f

With this general formulation we can use numerical optimization methods to solve for the optimal f as a sum of component vectors as in the following:

M f M = ∑ hm m=0

Note that this leaves aside the prior assumption that f is parameterized as a sum of individual decision trees.

∂L(y i , f (x i )) gi,m = [ ∂ f (x i ) ] f (x i )= f m−1 (x i )

4.6 Logistic Regression

The Bernoulli distribution we studied earlier answers the question of which of two outcomes (Y ∈ { 0, 1 } ) would be selected with probability, p.

P(Y) = p Y (1 − p)1−Y

We also know how to solve the corresponding likelihood function for the maximum likelihood estimate of p given observations of the output, { Y i } i=1 n . However, now we want to include other factors in our estimate of p. For example, suppose we observe not just the outcomes, but a corresponding continuous variable, x. That is, the observed data is now { (x i , Y i ) } i=1 n How can we incorporate x into our estimation of p?

The most straightforward idea is to model p = ax + b where a, b are parameters of a ﬁtted line. However, because p is a probability with value bounded between zero and one, we need to wrap this estimate in another function that can map the entire real line into the [ 0, 1 ] interval. The logistic (a.k.a. sigmoid) function has this property,

es θ(s) = 1 + es 286

4

Machine Learning

Thus, the new parameterized estimate for p is the following:

eax + b pˆ = θ(ax + b) = 1 + eax + b

(4.6.0.1)

The logit function is deﬁned as the following:

t logit(t) = log 1−t

It has the important property of extracting the regression components from the probability estimator,

logit(p) = b + ax

More continuous variables can be accommodated easily as

logit(p) = b + ∑ a k xk k

This can be further extended beyond the binary case to multiple target labels. The maximum likelihood estimate of this uses numerical optimization methods that are implemented in Scikit-learn.

Let’s construct some data to see how this works. In the following: we assign class labels to a set of randomly scattered points in the two-dimensional plane,

>>> import numpy as np

>>> from matplotlib.pylab import subplots

>>> v = 0.9

>>> @np.vectorize ... def gen\_y(x):

... if x<5: return np.random.choice([0,1],p=[v,1-v]) ... else: return np.random.choice([0,1],p=[1-v,v]) ...

>>> xi = np.sort(np.random.rand(500)\*10)

>>> yi = gen\_y(xi)

Programming Tip

The np.vectorize decorator used in the code above makes it easy to avoid looping in code that uses Numpy arrays by embedding the looping semantics inside of the so-decorated function. Note, however, that this does not necessarily accelerate the wrapped function. It’s mainly for convenience. 4.6 Logistic Regression

287

Fig. 4.23 This scatterplot shows the binary Y variables and the corresponding x data for each category

Figure 4.23 shows a scatter plot of the data we constructed in the above code, { (x i , Y i ) } . As constructed, it is more likely that large values of x correspond to Y = 1. On the other hand, values of x ∈ [ 4, 6 ] of either category are heavily overlapped. This means that x is not a particularly strong indicator of Y in this region. Figure 4.24 shows the ﬁtted logistic regression curve against the same data. The points along the curve are the probabilities that each point lies in either of the two categories. For large values of x the curve is near one, meaning that the probability that the associated Y value is equal to one. On the other extreme, small values of x mean that this probability is close to zero. Because there are only two possible categories, this means that the probability of Y = 0 is thereby higher. The region in the middle corresponding to the middle probabilities reﬂect the ambiguity between the two categories because of the overlap in the data for this region. Thus, logistic regression cannot make a strong case for one category here. The following code ﬁts the logistic regression model,

>>> from sklearn.linear\_model import LogisticRegression

>>> lr = LogisticRegression()

>>> lr.fit(np.c\_[xi],yi)

LogisticRegression(C=1.0, class\_weight=None, dual=False, fit\_intercept=True,

intercept\_scaling=1, max\_iter=100, multi\_class='warn',

n\_jobs=None, penalty='l2', random\_state=None, solver='warn',

tol=0.0001, verbose=0, warm\_start=False)

For a deeper understanding of logistic regression, we need to alter our notation slightly and once again use our projection methods. More generally we can re-write Eq.(4.6.0.1) as the following:

1 p(x) = 1 + exp(− β T x)

(4.6.0.2) 288

4

Machine Learning

Fig. 4.24 This shows the ﬁtted logistic regression on the data shown in Fig.4.23. The points along the curve are the probabilities that each point lies in either of the two categories

where β , x ∈ R n . From our prior work on projection we know that the signed perpendicular distance between x and the linear boundary described by β is β T x/ ∥β∥ . This means that the probability that is assigned to any point in R n is a function of how close that point is to the linear boundary described by the following equation,

β T x=0

But there is something subtle hiding here. Note that for any α ∈ R,

α β T x = 0

describes the same hyperplane. This means that we can multiply β by an arbitrary scalar and still get the same geometry. However, because of exp(−α β T x) in Eq.(4.6.0.2), this scaling determines the intensity of the probability attributed to x. This is illustrated in Fig.4.25. The panel on the left shows two categories (squares/circles) split by the dotted line that is determined by β T x = 0. The background colors show the probabilities assigned to points in the plane. The right panel shows that by scaling with α, we can increase the probabilities of class membership for the given points, given the exact same geometry. The points near the boundary have lower probabilities because they could easily be on the opposite side. However, by scaling by α, we can raise those probabilities to any desired level at the cost of driving the points further from the boundary closer to one. Why is this a problem? By driving the probabilities arbitrarily using α, we can overemphasize the training set at the cost of out-of-sample data. That is, we may wind up insisting on emphatic class membership of yet unseen points that are close to the boundary that otherwise would have more equivocal probabilities (say, near 1/2). Once again, this is another manifestation of bias/variance trade-off.

Regularization is a method that controls this effect by penalizing the size of β as part of its solution. Algorithmically, logistic regression works by iteratively solving 4.6 Logistic Regression

289

Fig. 4.25 Scaling can arbitrarily increase the probabilities of points near the decision boundary

a sequence of weighted least-squares problems. Regression adds a ∥β∥ /C term to the least-squares error. To see this in action, let’s create some data from a logistic regression and see if we can recover it using Scikit-learn. Let’s start with a scatter of points in the two-dimensional plane,

>>> x0,x1=np.random.rand(2,20)\*6-3

>>> X = np.c\_[x0,x1,x1\*0+1] # stack as columns

Note that X has a third column of all ones. This is a trick to allow the corresponding line to be offset from the origin in the two-dimensional plane. Next, we create a linear boundary and assign the class probabilities according to proximity to the boundary.

>>> beta = np.array([1,-1,1]) # last coordinate for affine offset

>>> prd = X.dot(beta)

>>> probs = 1/(1+np.exp(-prd/np.linalg.norm(beta)))

>>> c = (prd>0) # boolean array class labels

This establishes the training data. The next block creates the logistic regression object and ﬁts the data.

>>> lr = LogisticRegression()

>>> \_=lr.fit(X[:,:-1],c)

Note that we have to omit the third dimension because of how Scikit-learn internally breaks down the components of the boundary. The resulting code extracts the corresponding β from the LogisticRegression object.

>>> betah = np.r\_[lr.coef\_.flat,lr.intercept\_] 290

4

Machine Learning

Fig. 4.26 The left panel shows the resulting boundary (dashed line) with C = 1 as the regularization parameter. The right panel is for C = 1000. The gray line is the boundary used to assign the class membership for the synthetic data. The dark circle is the point that logistic regression categorizes incorrectly

Programming Tip

The Numpy np.r\_ object provides a quick way to stack Numpy arrays horizontally instead of using np.hstack.

The resulting boundary is shown in the left panel in Fig.4.26. The crosses and triangles represent the two classes we created above, along with the separating gray line. The logistic regression ﬁt produces the dotted black line. The dark circle is the point that logistic regression categorizes incorrectly. The regularization parameter is C = 1 by default. Next, we can change the strength of the regularization parameter as in the following:

>>> lr = LogisticRegression(C=1000)

and the re-ﬁt the data to produce the right panel in Fig.4.26. By increasing the regularization parameter, we essentially nudged the ﬁtting algorithm to believe the data more than the general model. That is, by doing this we accepted more variance in exchange for better bias.

Maximum Likelihood Estimation for Logistic Regression. Let us again consider the binary classiﬁcation problem. We deﬁne y k = P(C 1 | x k ), the conditional probability of the data as a member of given class. Our construction of this problem provides

y k = θ( [ w, w 0 ] · [ x k , 1 ] ) 4.6 Logistic Regression

291

where θ is the logistic function. Recall that there are only two classes for this problem. The dataset looks like the following:

{ (x 0 ,r 0 ), . . . , (x k ,r k ), . . . , (x n−1 ,r n−1 ) }

where r k ∈ { 0, 1 } . For example, we could have the following sequence of observed classes,

{ C 0 , C 1 , C 1 , C 0 , C 1 }

For this case the likelihood is then the following:

= P(C 0 | x 0 )P(C 1 | x 1 )P(C 1 | x 1 )P(C 0 | x 0 )P(C 1 | x 1 )

which we can re-write as the following:

(w, w 0 ) = (1 − y 0 )y 1 y 2 (1 − y 3 )y4

Recall that there are two mutually exhaustive classes. More generally, this can be written as the following:

n (w | X) = ∏ y k r k (1 − y k )1−r k k

Naturally, we want to compute the logarithm of this as the cross-entropy,

E = − ∑ r k log(y k ) + (1 − r k ) log(1 − y k ) k

and then minimize this to ﬁnd w and w 0 . This is difﬁcult to do with calculus because the derivatives have nonlinear terms in them that are hard to solve for.

Multi-class Logistic Regression Using Softmax. The logistic regression problem provides a solution for the probability between exactly two alternative classes. To extend to the multi-class problem, we need the softmax function. Consider the likelihood ratio between the i th class and the reference class, C k ,

p(x ) | Ci log x = wi T p(x | C k )

Taking the exponential of this and normalizing across all the classes gives the softmax function,

exp ( w i T x ) y i = p(C i | x) = ∑ k exp ( w k T x ) 292

4

Machine Learning

Note that ∑ i y i = 1. If the w i T x term is larger than the others, after the exponentiation and normalization, it automatically suppresses the other y j ∀ j = ̸ i, which acts like the maximum function, except this function is differentiable, hence soft, as in softmax. While that is all straightforward, the trick is deriving the w i vectors from the training data { x i , y i } .

Once again, the launching point is the likelihood function. As with the two-class logistic regression problem, we have the likelihood as the following:

= ∏ ∏ (y i k )r i k k i

The log-likelihood of this is the same as the cross-entropy,

E = − ∑ ∑ r i k log yi k k i

This is the error function we want to minimize. The computation works as before with logistic regression, except there are more derivatives to keep track of in this case.

Understanding Logistic Regression. To generalize this technique beyond logistic regression, we need to re-think the problem more abstractly as the dataset { x i , y i } . We have the y i ∈ { 0, 1 } data modeled as Bernoulli random variables. We also have the x i data associated with each y i , but it is not clear how to exploit this association. What we would like is to construct E(Y | X) which we already know (see Sect.2.1) is the best MSE estimator. For this problem, we have

E(Y | X) = P(Y | X)

because only Y = 1 is nonzero in the summation. Regardless, we don’t have the conditional probabilities anyway. One way to look at logistic regression is as a way to build in the functional relationship between y i and x i . The simplest thing we could do is approximate,

E(Y | X) ≈ β 0 + β 1 x := η (x)

If this is the model, then the target would be the y i data. We can force the output of this linear regression into the interval [ 0, 1 ] by composing it with a sigmoidal function,

1 θ(x) = 1 + exp(−x)

Then we have a new function θ( η (x)) to match against y i using

J( β 0 , β 1 ) = ∑ (θ( η (x i )) − y i )2 i 4.6 Logistic Regression

293

This is a nice setup for an optimization problem. We could certainly solve this numerically using scipy.optimize. Unfortunately, this would take us into the black box of the optimization algorithm where we would lose all of our intuitions and experience with linear regression. We can take the opposite approach. Instead of trying to squash the output of the linear estimator into the desired domain, we can map the y i data into the unbounded space of the linear estimator. Thus, we deﬁne the inverse of the above θ function as the link function.

y g(y) = log ( 1 − y )

This means that our approximation to the unknown conditional expectation is the following:

g(E(Y | X)) ≈ β 0 + β 1 x := η (x)

We cannot apply this directly to the y i , so we compute the Taylor series expansion centered on E(Y | X), up to the linear term, to obtain the following:

g(Y)

≈ ≈

g(E(Y

|

X))

+

(Y

−

E(Y

|

X))g

′

(E(Y

|

X))

η (x) + (Y − θ( η (x)))g ′ (θ( η (x))) := z

Because we do not know the conditional expectation, we replaced these terms with our earlier θ( η (x)) function. This new approximation deﬁnes our transformed data that we will use to feed the linear model. Note that the β parameters are embedded in this transformation. The (Y − θ( η (x))) term acts as the usual additive noise term. Also,

1 g ′ (x) = x(1 − x)

The following code applies this transformation to the xi,yi data

>>> b0, b1 = -2,0.5

>>> g = lambda x: np.log(x/(1-x))

>>> theta = lambda x: 1/(1+np.exp(-x))

>>> eta = lambda x: b0 + b1\*x

>>> theta\_ = theta(eta(xi))

>>> z=eta(xi)+(yi-theta\_)/(theta\_\*(1-theta\_))

Note the two vertical scales shown in Fig.4.27. The red scale on the right is the { 0, 1 } domain of the y i data (red dots) and the left scale is transformed z i data (black dots). Note that the transformed data is more linear where the original data is less equivocal at the extremes. Also, this transformation used a speciﬁc pair of 294

4

Machine Learning

Fig. 4.27 The transformation underlying logistic regression

β i parameters. The idea is to iterate over this transformation and derive new βi parameters. With this approach, we have

V(Z | X) = (g ′ ) 2 V(Y | X)

Recall that, for this binary variable, we have

P(Y | X) = θ( η (x)))

Thus,

V(Y | X) = θ( η (x))(1 − θ( η (x)))

from which we obtain

V(Z | X) = [θ( η (x))(1 − θ( η (x)))]−1

The important fact here is the variance is a function of the X (i.e., heteroskedastic). As we discussed with Gauss–Markov, the appropriate linear regression is weighted least-squares where the weights at each data point are inversely proportional to the variance. This ensures that the regression process accounts for this heteroskedasticity. Numpy has a weighted least-squares implemented in the polyfit function,

>>> w=(theta\_\*(1-theta\_))

>>> p=np.polyfit(xi,z,1,w=np.sqrt(w))

The output of this ﬁt is shown in Fig.4.28, along with the raw data and V(Z | X) for this particular ﬁtted β i . Iterating a few more times reﬁnes the estimated line but it does not take many such iterations to converge. As indicated by the variance line, the ﬁtted line favors the data at either extreme. 4.7 Generalized Linear Models

295

4.7 Generalized Linear Models

Logistic regression is one example of a wider class of Generalized Linear Models (GLMs). These GLMs have the following three key features

• A target Y variable distributed according to one of the exponential family of distributions (e.g., Normal, binomial, Poisson)

• An equation that links the expected value of Y with a linear combination of the observed variables (i.e., { x 1 , x 2 , . . . , x n } ).

• A smooth invertible link function g(x) such that g(E(Y)) = ∑ k β k xk

Exponential Family. Here is the one-parameter exponential family,

f (y ; λ) = e λy−γ(λ)

The natural parameter is λ and y is the sufﬁcient statistic. For example, for logistic p regression, we have γ(λ) = − log(1 + e λ ) and λ = log 1−p .

An important property of this exponential family is that

dγ(λ) E λ (y) = = γ ′ (λ) dλ

(4.7.0.1)

To see this, we compute the following:

1 = f (y ; λ)dy = e λy−γ(λ) dy ∫ ∫

d f (y ; λ) 0 = dy = e λy−γ(λ) ( y − γ ′ (λ) ) dy ∫ dλ ∫

ye λy−γ(λ) dy = E λ (y) = γ ′ (λ) ∫

Using the same technique, we also have,

V λ (Y) = γ ′′ (λ)

which explains the usefulness of this generalized notation for the exponential family.

Deviance. The scaled Kullback–Leibler divergence is called the deviance as deﬁned below,

f1 (y) D( f 1 , f 2 ) = 2 f 1 (y) log dy ∫ f 2 (y) 296

4

Machine Learning

Hoeffding’s Lemma

Using our exponential family notation, we can write out the deviance as the following:

1 ) f (y λ1 ; D( f (y ), f (y ; λ 2 )) = f (y ; λ 1 ) log dy λ1 ; 2 ∫ f (y ; λ 2 )

= f (y ; λ 1 )((λ 1 − λ 2 )y − ( γ (λ 1 ) − γ (λ 2 )))dy ∫

= E λ 1 [ (λ 1 − λ 2 )y − ( γ (λ 1 ) − γ (λ 2 )) ]

= (λ 1 − λ 2 )E λ 1 (y) − ( γ (λ 1 ) − γ (λ 2 ))

= (λ 1 − λ 2 )μ 1 − ( γ (λ 1 ) − γ (λ 2 ))

whereμ 1 := E λ 1 (y).Forthemaximumlikelihoodestimate Plugging this into the above equation gives the following:

ˆ λ1

,wehaveμ 1 = y.

1 ˆ ˆ ˆ D( f (y λ 1 ), f (y ; λ 2 )) = ( λ 1 − λ 2 )y − ( γ ( λ 1 ) − γ (λ 2 )) ; 2

= log f (y ; λ 1 ˆ ) − log f (y ; λ 2 )

ˆ f (y λ 1 ) ; = log f (y ; λ 2 )

Taking the negative exponential of both sides gives

ˆ ˆ f (y ; λ 2 ) = f (y ; λ 1 )e− 1 2 D( f (y ; λ 1 ), f (y ; λ 2 ))

Because D is always nonnegative, the likelihood is maximized when the deviance is zero. In particular, for the scalar case, it means that y itself is ˆ the best maximum likelihood estimate for the mean. Also, f (y ; λ 1 ) is called the saturated model. We write Hoeffding’s Lemma as the following:

f (y ; μ) = f (y ; y)e− 1 2 D( f (y ; y), f (y ; μ))

(4.7.0.2)

to emphasize that f (y ; y) is the likelihood function when the mean is replaced by the sample itself and f (y ; μ) is the likelihood function when the mean is replaced by μ.

Vectorizing Equation (4.7.0.2) using mutual independence gives the following:

f (y ; μ) = e − ∑ i D(y i ,μ i ) ∏ f (y i ; y i )

The idea now is to minimize the deviance by deriving, 4.7 Generalized Linear Models

297

Fig. 4.28 The output of the weighted least-squares ﬁt is shown, along with the raw data and V(Z | X)

μ(β) = g −1 (M T β)

This means the MLE β ˆ is the best p × 1 vector β that minimizes the total deviance where g is the link function and M is the p × n structure matrix. This is the key step with GLM estimation because it reduces the number of parameters from n to p. The structure matrix is where the associated x i data enters into the problem. Thus, GLM maximum likelihood ﬁtting minimizes the total deviance like plain linear regression minimizes the sum of squares.

With the following:

λ =MT β

with 2×n-dimensional M. The corresponding joint density function is the following:

f (y ; β) = e β T ξ − ψ (β) f 0 (y)

where

ξ = My

and

ψ (β) = ∑ γ(m i T β)

where now the sufﬁcient statistic is ξ and the parameter vector is β, which ﬁts into our exponential family format, and m i is the i th column of M. 298

4

Machine Learning

Given this joint density, we can compute the log-likelihood as the following:

= β T ξ − ψ (β)

To maximize this likelihood, we take the derivative of this with respect to β to obtain the following:

d = My Mμ(MT β) dβ

since γ ′ (m i T β) = m i T μ i (β) and (c.f. Eq.(4.7.0.1)), γ ′ = μ λ . Setting this derivative equal to zero gives the conditions for the maximum likelihood solution,

M(y − μ(M T β)) = 0

(4.7.0.3)

where μ is the element-wise inverse of the link function. This leads us to exactly the same place we started: trying to regress y against μ(M T β).

Example. The structure matrix M is where the x i data associated with the corresponding y i enters the problem. If we choose

M T = [ 1, x ]

where 1 is an n-length vector and

β = [ β 0 , β 1 ]T

with μ(x) = 1/(1 + e −x ), we have the original logistic regression problem.

Generally,μ(β)isanonlinearfunctionandthusweregressagainstourtransformed variable z

z = M T β + diag(g ′ (μ))(y − μ(M T β))

This ﬁts the format of the Gauss Markov (see Sect.3.11) problem and has the following solution, (4.7.0.4)

β ˆ = (MR z −1 M T ) −1 MR z −1 z

where

R z := V(z) = diag(g ′ (μ)) 2 R = v(μ) diag(g ′ (μ)) 2 I

where g is the link function and v is the variance function on the designated distriˆ bution of the y i . Thus, β has the following covariance matrix, 4.7 Generalized Linear Models

299

Fig. 4.29 Some data for Poisson example

V( ˆ β) = (MR z −1 M T )−1

These results allow inferences about the estimated parameters Eq.(4.7.0.4) as an iteration as follow,

ˆ β. We can easily write

β k + 1 ˆ = (MR z k −1 M T ) −1 MR z k −1 zk

Example. Consider the data shown in Fig.4.29. Note that the variance of the data increases for each x and the data increases as a power of x along x. This makes this data a good candidate for a Poisson GLM with g(μ) = log(μ).

We can use our iterative matrix-based approach. The following code initializes the iteration.

>>> M = np.c\_[x\*0+1,x].T

>>> gi = np.exp # inverse g link function

>>> bk = np.array([.9,0.5]) # initial point

>>> muk = gi(M.T @ bk).flatten()

>>> Rz = np.diag(1/muk)

>>> zk = M.T @ bk + Rz @ (y-muk)

and this next block establishes the main iteration

>>> while abs(sum(M @ (y-muk))) > .01: # orthogonality condition as threshold ... Rzi = np.linalg.inv(Rz) ... bk = (np.linalg.inv(M @ Rzi @ M.T)) @ M @ Rzi @ zk ... muk = gi(M.T @ bk).flatten() ... Rz =np.diag(1/muk) ... zk = M.T @ bk + Rz @ (y-muk) ...

with corresponding ﬁnal β computed as the following: 300

4

Machine Learning

>>> print(bk) [0.71264653 0.48934384]

ˆ with corresponding estimated V( β) as

>>> print(np.linalg.inv(M @ Rzi @ M.T)) [[ 0.01867659 -0.00359408] [-0.00359408 0.00073501]]

The orthogonality condition Eq.(4.7.0.3) is the following:

>>> print(M @ (y-muk)) [-5.88442660e-05 -3.12199976e-04]

For comparison, the statsmodels module provides the Poisson GLM object. Note that the reported standard error is the square root of the diagonal elements of ˆ V( β). A plot of the data and the ﬁtted model is shown below in Fig.4.30.

>>> pm=sm.GLM(y, sm.tools.add\_constant(x),

... family=sm.families.Poisson())

>>> pm\_results=pm.fit()

>>> pm\_results.summary()

<class 'statsmodels.iolib.summary.Summary'> """ Generalized Linear Model Regression Results ============================================================================== Dep. Variable: y No. Observations: 50 Model: GLM Df Residuals: 48 Model Family: Poisson Df Model: 1 Link Function: log Scale: 1.0000 Method: IRLS Log-Likelihood: -134.00 Date: Tue, 12 Mar 2019 Deviance: 44.230 Time: 06:54:16 Pearson chi2: 43.1 No. Iterations: 5 Covariance Type: nonrobust ============================================================================== coef std err z P>|z| [0.025 0.975]

-----------------------------------------------------------------------------const 0.7126 0.137 5.214 0.000 0.445 0.981 x1 0.4893 0.027 18.047 0.000 0.436 0.542 ============================================================================== """

4.8 Regularization

We have referred to regularization in Sect.4.6, but we want to develop this important idea more fully. Regularization is the mechanism by which we navigate the bias/variance trade-off. To get started, let’s consider a classic constrained least-squares problem, 4.8 Regularization

301

Fig. 4.30 Fitted using the Poisson GLM

Fig. 4.31 The solution of the constrained L 2 minimization problem is at the point where the constraint (dark line) intersects the L 2 ball (gray circle) centered at the origin. The point of intersection is indicated by the dark circle. The two neighboring squares indicate points on the line that are close to the solution

minimize

x

x ∥2 2

∥

subject to: x 0 + 2x 1 = 1

where ∥ x ∥ 2 = x 0 2 + x 1 2 is the L 2 norm. Without the constraint, it would be easy to √ minimize the objective function—just take x = 0. Otherwise, suppose we somehow know that ∥ x ∥ 2 < c, then the locus of points deﬁned by this inequality is the circle in Fig.4.31. The constraint is the line in the same ﬁgure. Because every value of c deﬁnes a circle, the constraint is satisﬁed when the circle touches the line. The circle can touch the line at many different points, but we are only interested in the smallest such circle because this is a minimization problem. Intuitively, this means that we inﬂate a L 2 ball at the origin and stop when it just touches the constraint. The point of contact is our L 2 minimization solution. 302

4

Machine Learning

We can obtain the same result using the method of Lagrange multipliers. We can re-write the entire L 2 minimization problem as one objective function using the Lagrange multiplier, λ,

J(x 0 , x 1 , λ) = x 0 2 + x 1 2 + λ(1 − x 0 − x 1 )

and solve this as an ordinary function using calculus. Let’s do this using Sympy.

>>> import sympy as S

>>> S.var('x:2 l',real=True) (x0, x1, l)

>>> J=S.Matrix([x0,x1]).norm()\*\*2 + l\*(1-x0-2\*x1)

>>> sol=S.solve(map(J.diff,[x0,x1,l]))

>>> print(sol)

{l: 2/5, x0: 1/5, x1: 2/5}

Programming Tip

Using the Matrix object is overkill for this problem but it does demonstrate how Sympy’s matrix machinery works. In this case, we are using the norm method to compute the L 2 norm of the given elements. Using S.var deﬁnes Sympy variables and injects them into the global namespace. It is more Pythonic to do something like x0 = S.symbols(’x0’,real=True) instead but the other way is quicker, especially for variables with many dimensions.

The solution deﬁnes the exact point where the line is tangent to the circle in Fig.4.31. The Lagrange multiplier has incorporated the constraint into the objective function.

There is something subtle and very important about the nature of the solution, however. Notice that there are other points very close to the solution on the circle, indicated by the squares in Fig.4.31. This closeness could be a good thing, in case it helps us actually ﬁnd a solution in the ﬁrst place, but it may be unhelpful in so far as it creates ambiguity. Let’s hold that thought and try the same problem using the L1 norm instead of the L 2 norm. Recall that

d ∥ x ∥ 1 = ∑ | x i | i=1

where d is the dimension of the vector x. Thus, we can reformulate the same problem in the L 1 norm as in the following:

minimize

x

x ∥1

∥

subject to: x 1 + 2x 2 = 1 4.8 Regularization

303

It turns out that this problem is somewhat harder to solve using Sympy, but we have convex optimization modules in Python that can help.

>>> from cvxpy import Variable, Problem, Minimize, norm1, norm

>>> x=Variable((2,1),name='x')

>>> constr=[np.matrix([[1,2]])\*x==1]

>>> obj=Minimize(norm1(x))

>>> p= Problem(obj,constr)

>>> p.solve()

0.49999999996804073

>>> print(x.value) [[6.2034426e-10] [5.0000000e-01]]

Programming Tip

The cvxy module provides a uniﬁed and accessible interface to the powerful cvxopt convex optimization package, as well as other open-source solver packages.

As shown in Fig.4.32, the constant-norm contour in the L 1 norm is shaped like a diamond instead of a circle. Furthermore, the solutions found in each case are different. Geometrically, this is because inﬂating the circular L 2 reaches out in all directions whereas the L 1 ball creeps out along the principal axes. This effect is much morepronouncedinhigherdimensionalspaceswhere L 1 -ballsgetmorespikey. 6 Like the L 2 case, there are also neighboring points on the constraint line, but notice that these are not close to the boundary of the corresponding L 1 ball, as they were in the L 2 case. This means that these would be harder to confuse with the optimal solution because they correspond to a substantially different L 1 ball.

To double-check our earlier L 2 result, we can also use the cvxpy module to ﬁnd the L 2 solution as in the following code,

>>> constr=[np.matrix([[1,2]])\*x==1]

>>> obj=Minimize(norm(x,2)) #L2 norm

>>> p= Problem(obj,constr)

>>> p.solve()

0.4473666974719267

>>> print(x.value) [[0.1999737 ] [0.40004849]]

The only change to the code is the L 2 norm and we get the same solution as before.

Let’s see what happens in higher dimensions for both L 2 and L 1 as we move from two dimensions to four dimensions.

6 We discussedthe geometryofhigh-dimensional space whenwe coveredthe curse ofdimensionality in the statistics chapter. 304

4

Machine Learning

>>> x=Variable((4,1),name='x')

>>> constr=[np.matrix([[1,2,3,4]])\*x==1]

>>> obj=Minimize(norm1(x))

>>> p= Problem(obj,constr)

>>> p.solve()

0.2499999991355072

>>> print(x.value) [[3.88487210e-10] [8.33295420e-10] [7.97158511e-10] [2.49999999e-01]]

And also in the L 2 case with the following code,

>>> constr=[np.matrix([[1,2,3,4]])\*x==1]

>>> obj=Minimize(norm(x,2))

>>> p= Problem(obj,constr)

>>> p.solve()

0.1824824789618193

>>> print(x.value) [[0.03332451] [0.0666562 ] [0.09999604] [0.13333046]]

Note that the L 1 solution has selected out only one dimension for the solution, as the other components are effectively zero. This is not so with the L 2 solution, which has meaningful elements in multiple coordinates. This is because the L 1 problem has many pointy corners in the four-dimensional space that poke at the hyperplane that is deﬁned by the constraint. This essentially means the subsets (namely, the points at the corners) are found as solutions because these touch the hyperplane. This effect becomes more pronounced in higher dimensions, which is the main beneﬁt of using the L 1 norm as we will see in the next section.

4.8.1 Ridge Regression

Now that we have a sense of the geometry of the situation, let’s revisit our classic linear regression problem. To recap, we want to solve the following problem,

min ∥ y − Xβ ∥ β∈R p

where X = [ x 1 , x 2 , . . . , x p ] and x i ∈ R n . Furthermore, we assume that the p column vectors are linearly independent (i.e., rank(X) = p). Linear regression produces the 4.8 Regularization

305

Fig. 4.32 The diamond is the L 1 ball in two dimensions and the line is the constraint. The point of intersection is the solution to the optimization problem. Note that for L 1 optimization, the two nearby points on the constraint (squares) do not touch the L 1 ball. Compare this with Fig.4.31

β that minimizes the mean squared error above. In the case where p = n, there is a unique solution to this problem. However, when p < n, then there are inﬁnitely many solutions.

To make this concrete, let’s work this out using Sympy. First, let’s deﬁne an example X and y matrix,

>>> import sympy as S

>>> from sympy import Matrix

>>> X = Matrix([[1,2,3],

... [3,4,5]]) >>> y = Matrix([[1,2]]).T

Now, we can deﬁne our coefﬁcient vector β using the following code,

>>> b0,b1,b2=S.symbols('b:3',real=True)

>>> beta = Matrix([[b0,b1,b2]]).T # transpose

Next, we deﬁne the objective function we are trying to minimize

>>> obj=(X\*beta -y).norm(ord=2)\*\*2

Programming Tip

The Sympy Matrix class has useful methods like the norm function used above to deﬁne the objective function. The ord=2 means we want to use the L 2 norm. The expression in parenthesis evaluates to a Matrix object.

Note that it is helpful to deﬁne real variables using the keyword argument whenever applicable because it relieves Sympy’s internal machinery of dealing with complex 306

4

Machine Learning

numbers. Finally, we can use calculus to solve this by setting the derivatives of the objective function to zero.

>>> sol=S.solve([obj.diff(i) for i in beta])

>>> beta.subs(sol) Matrix([ [ b2], [-2\*b2 + 1/2], [ b2]])

Notice that the solution does not uniquely specify all the components of the beta variable. This is a consequence of the p < n nature of this problem where p = 2 and n = 3. While the existence of this ambiguity does not alter the solution,

>>> obj.subs(sol) 0

But it does change the length of the solution vector beta,

>>> beta.subs(sol).norm(2) sqrt(2\*b2\*\*2 + (2\*b2 - 1/2)\*\*2)

If we want to minimize this length we can easily use the same calculus as before,

>>> S.solve((beta.subs(sol).norm()\*\*2).diff()) [1/6]

This provides the solution of minimum length in the L 2 sense,

>>> betaL2=beta.subs(sol).subs(b2,S.Rational(1,6))

>>> betaL2 Matrix([ [1/6], [1/6], [1/6]])

But what is so special about solutions of minimum length? For machine learning, driving the objective function to zero is symptomatic of overﬁtting the data. Usually, at the zero bound, the machine learning method has essentially memorized the training data, which is bad for generalization. Thus, we can effectively stall this problem by deﬁning a region for the solution that is away from the zero bound.

minimize ∥ y − Xβ ∥2 2 β

subject to: ∥ β ∥ 2 < c

where c is the tuning parameter. Using the same process as before, we can re-write this as the following: 4.8 Regularization

307

min ∥ y − Xβ ∥ 2 2 + α ∥ β ∥2 2 β∈R p

where α is the tuning parameter. These are the penalized or Lagrange forms of these problems derived from the constrained versions. The objective function is penalized by the ∥ β ∥ 2 term. For L 2 penalization, this is called ridge regression. This is implemented in Scikit-learn as Ridge. The following code sets this up for our example,

>>> from sklearn.linear\_model import Ridge

>>> clf = Ridge(alpha=100.0,fit\_intercept=False)

>>> clf.fit(np.array(X).astype(float),np.array(y).astype(float)) Ridge(alpha=100.0, copy\_X=True, fit\_intercept=False, max\_iter=None, normalize=False, random\_state=None, solver='auto', tol=0.001)

Note that the alpha scales of the penalty for the ∥ β ∥ 2 . We set the fit\_intercept=False argument to omit the extra offset term from our example. The corresponding solution is the following:

>>> print(clf.coef\_) [[0.0428641 0.06113005 0.07939601]]

To double-check the solution, we can use some optimization tools from Scipy and our previous Sympy analysis, as in the following:

>>> from scipy.optimize import minimize

>>> f = S.lambdify((b0,b1,b2),obj+beta.norm()\*\*2\*100.)

>>> g

=

lambda

x:f(x[0],x[1],x[2])

>>> out = minimize(g,[.1,.2,.3]) # initial guess

>>> out.x

array([0.0428641 , 0.06113005, 0.07939601])

Programming Tip

We had to deﬁne the additional g function from the lambda function we created from the Sympy expression in f because the minimize function expects a single object vector as input instead of a three separate arguments.

which produces the same answer as the Ridge object. To better understand the meaning of this result, we can re-compute the mean squared error solution to this problem in one step using matrix algebra instead of calculus,

>>> betaLS=X.T\*(X\*X.T).inv()\*y

>>> betaLS Matrix([ [1/6], [1/6], [1/6]]) 308

4

Machine Learning

Notice that this solves the posited problem exactly,

>>> X\*betaLS-y Matrix([

[0],

[0]])

This means that the ﬁrst term in the objective function goes to zero,

∥ y − Xβ LS ∥ = 0

But, let’s examine the L 2 length of this solution versus the ridge regression solution,

>>> print(betaLS.norm().evalf(), np.linalg.norm(clf.coef\_)) 0.288675134594813 0.10898596412575512

Thus, the ridge regression solution is shorter in the L 2 sense, but the ﬁrst term in the objective function is not zero for ridge regression,

>>> print((y-X\*clf.coef\_.T).norm()\*\*2) 1.86870864136429

Ridge regression solution trades ﬁtting error ( ∥ y −Xβ ∥ 2 ) for solution length ( ∥ β ∥ 2 ).

Let’s see this in action with a familiar example from Sect.3.12.4. Consider Fig.4.33. For this example, we created our usual chirp signal and attempted to ﬁt it with a high-dimensional polynomial, as we did in Sect.4.3.4. The lower panel is the same except with ridge regression. The shaded gray area is the space between the true signal and the approximant in both cases. The horizontal hash marks indicate the subset of x i values that each regressor was trained on. Thus, the training set represents a nonuniform sample of the underlying chirp waveform. The top panel shows the usual polynomial regression. Note that the regressor ﬁts the given points extremely well, but fails at the endpoint. The ridge regressor misses many of the points in the middle, as indicated by the gray area, but does not overshoot at the ends as much as the plain polynomial regression. This is the basic trade-off for ridge regression. The Jupyter notebook corresponding to this section has the code for this graph, but the main steps are shown in the following:

# create chirp signal # sample chirp randomly xin= np.sort(np.random.choice(xi.flatten(),20,replace=False))[:,None] # create sampled waveform y = np.cos(2\*pi\*(xin+xin\*\*2)) # create full waveform for reference yi = np.cos(2\*pi\*(xi+xi\*\*2)) # create polynomial features from sklearn.preprocessing import PolynomialFeatures

xi = np.linspace(0,1,100)[:,None]

qfit = PolynomialFeatures(degree=8) # quadratic 4.8 Regularization

309

Fig. 4.33 The top ﬁgure shows polynomial regression and the lower panel shows polynomial ridge regression. The ridge regression does not match as well throughout most of the domain, but it does not ﬂare as violently at the ends. This is because the ridge constraint holds the coefﬁcient vector down at the expense of poorer performance along the middle of the domain

Xq = qfit.fit\_transform(xin) # reformat input as polynomial Xiq = qfit.fit\_transform(xi)

from sklearn.linear\_model import LinearRegression lr=LinearRegression() # create linear model lr.fit(Xq,y) # fit linear model # create ridge regression model and fit clf = Ridge(alpha=1e-9,fit\_intercept=False) clf.fit(Xq,y)

4.8.2 Lasso Regression

Lasso regression follows the same basic pattern as ridge regression, except with the L 1 norm in the objective function.

min ∥ y − Xβ ∥ 2 + α ∥ β ∥1 β∈R p

The interface in Scikit-learn is likewise the same. The following is the same problem as before using lasso instead of ridge regression, 310

4

Machine Learning

>>> X = np.matrix([[1,2,3],

... [3,4,5]])

>>> y = np.matrix([[1,2]]).T

>>> from sklearn.linear\_model import Lasso

>>> lr = Lasso(alpha=1.0,fit\_intercept=False)

>>> \_=lr.fit(X,y)

>>> print(lr.coef\_)

[0. 0. 0.32352941]

As before, we can use the optimization tools in Scipy to solve this also,

>>> from scipy.optimize import fmin

>>> obj = 1/4.\*(X\*beta-y).norm(2)\*\*2 + beta.norm(1)\*l

>>> f = S.lambdify((b0,b1,b2),obj.subs(l,1.0))

>>> g = lambda x:f(x[0],x[1],x[2])

>>> fmin(g,[0.1,0.2,0.3]) Optimization terminated successfully.

Current function value: 0.360297 Iterations: 121 Function evaluations: 221 array([2.27469304e-06, 4.02831864e-06, 3.23134859e-01])

Programming Tip

The fmin function from Scipy’s optimization module uses an algorithm that does not depend upon derivatives. This is useful because, unlike the L 2 norm, the L 1 norm has sharp corners that make it harder to estimate derivatives.

This result matches the previous one from the Scikit-learn Lasso object. Solving it using Scipy is motivating and provides a good sanity check, but specialized algorithms are required in practice. The following code block re-runs the lasso with varying α and plots the coefﬁcients in Fig.4.34. Notice that as α increases, all but one of the coefﬁcients is driven to zero. Increasing α makes the trade-off between ﬁtting the data in the L 1 sense and wanting to reduce the number of nonzero coefﬁcients (equivalently, the number of features used) in the model. For a given problem, it may be more practical to focus on reducing the number of features in the model (i.e., large α) than the quality of the data ﬁt in the training data. The lasso provides a clean way to navigate this trade-off.

The following code loops over a set of α values and collects the corresponding lasso coefﬁcients to be plotted in Fig.4.34

>>> o=[]

>>> alphas= np.logspace(-3,0,10)

>>> for a in alphas: 4.8 Regularization

311

Fig. 4.34 As α increases, more of the model coefﬁcients are driven to zero for lasso regression

... ... ... ...

clf = Lasso(alpha=a,fit\_intercept=False) \_=clf.fit(X,y) o.append(clf.coef\_)

4.9 Support Vector Machines

Support vector machines (SVM) originated from the statistical learning theory developed by Vapnik–Chervonenkis. As such, it represents a deep application of statistical theory that incorporates the VC dimension concepts we discussed in the ﬁrst section. Let’s start by looking at some pictures. Consider the two-dimensional classiﬁcation problem shown in Fig.4.35. Figure 4.35 shows two classes (gray and white circles) that can be separated by any of the lines shown. Speciﬁcally, any such separating line can be written as the locus of points (x) in the two-dimensional plane that satisfy the following:

β 0 + β T x =0

To classify an arbitrary x using this line, we just compute the sign of β 0 + β T x and assign one class to the positive sign and the other class to the negative sign. To uniquely specify such a separating line (or, hyperplane in a higher dimensional space) we need additional criteria.

Figure 4.36 shows the data with two bordering parallel lines that form a margin around the central separating line. The maximal margin algorithm ﬁnds the widest 312

4

Machine Learning

Fig. 4.35 In the two-dimensional plane, the two classes (gray and white circles) are easily separated by any one of the lines shown

Fig. 4.36 The maximal margin algorithm ﬁnds the separating line that maximizes the margin shown. The elements that touch the margins are the support elements. The dotted elements are not relevant to the solution

margin and the unique separating line. As a consequence, the algorithm uncovers the elements in the data that touch the margins. These are the support elements. The other elements away from the border are not relevant to the solution. This reduces model variance because the solution is insensitive to the removal of elements other than these supporting elements (usually a small minority).

To see how this works for linearly separable classes, consider a training set consisting of { (x, y) } where y ∈ { −1, 1 } . For any point x i , we compute the functional margin as γ i ˆ = y i ( β 0 + β T x i ). Thus, γ i ˆ > 0 when x i is correctly classiﬁed. The geometrical margin is γ = γ ˆ / ∥β∥ . When x i is correctly classiﬁed, the geometrical margin is equal to the perpendicular distance from x i to the line. Let’s look see how the maximal margin algorithm works.

Let M be the width of the margin. The maximal margin algorithm is can be formulated as a quadratic programming problem. We want to simultaneously maximize the margin M while ensuring that all of the data points are correctly classiﬁed.

maximize M β 0 , β , ∥β∥ =1

subject to: y i ( β 0 + β T x i ) ≥ M, i = 1, . . . , N. 4.9 Support Vector Machines

313

The ﬁrst line says we want to generate a maximum value for M by adjusting β 0 and β while keeping ∥β∥ = 1. The functional margins for each i th data element are the constraints to the problem and must be satisﬁed for every proposed solution. In words, the constraints enforce that the elements have to be correctly classiﬁed and outside of the margin around the separating line. With some reformulation, it turns out that M = 1/ ∥β∥ and this can be put into the following standard format,

minimize ∥β∥ β 0 , β

subject to: y i ( β 0 + β T x i ) ≥ 1, i = 1, . . . , N.

This is a convex optimization problem and can be solved using powerful methods in that area.

The situation becomes more complex when the two classes are not separable and we have to allow some unavoidable mixing between the two classes in the solution. This means that the constraints have to be modiﬁed as in the following:

y i ( β 0 + β T x i ) ≥ M(1 − ξ i )

where ξ i are the slack variables and represent the proportional amount that the prediction is on the wrong side of the margin. Thus, elements are misclassiﬁed when ξ i > 1. With these additional variables, we have a more general formulation of the convex optimization problem,

minimize ∥β∥ β 0 , β

subject to: y i ( β 0 + β T x i ) ≥ 1 − ξ i ,

ξ i ≥ 0, ∑ ξ i ≤ constant, i = 1, . . . , N.

which can be rewritten in the following equivalent form,

1 minimize ∑ ξi C + ∥β∥ β 0 , β 2

(4.9.0.1)

subject to: y i ( β 0 + β T x i ) ≥ 1 − ξ i , ξ i ≥ 0 i = 1, . . . , N.

Because the ξ i terms are all positive, the objective is to maximize the margin (i.e., minimize ∥β∥ ) while minimizing the proportional drift of the predictions to the wrong side of the margin (i.e., C ∑ ξ i ). Thus, large values of C shunt algorithmic focus toward the correctly classiﬁed points near the decision boundary and small values focus on further data. The value C is a hyper-parameter for the SVM.

The good news is that all of these complicated pieces are handled neatly inside of Scikit-learn. The following sets up the linear kernel for the SVM (more on kernels soon), 314

4

Machine Learning

Fig. 4.37 The two class shown (white and gray circles) are linearly separable. The maximal margin solution is shown by the dark black line in the middle. The dotted lines show the extent of the margin. The large circles indicate the support vectors for the maximal margin solution

>>> from sklearn.datasets import make\_blobs

>>> from sklearn.svm import SVC

>>> sv = SVC(kernel='linear')

We can create some synthetic data using make\_blobs and then ﬁt it to the SVM,

>>> X,y=make\_blobs(n\_samples=200, centers=2, n\_features=2,

... random\_state=0,cluster\_std=.5) >>> sv.fit(X,y) SVC(C=1.0, cache\_size=200, class\_weight=None, coef0=0.0,

decision\_function\_shape='ovr', degree=3, gamma='auto\_deprecated',

kernel='linear', max\_iter=-1, probability=False, random\_state=None,

shrinking=True, tol=0.001, verbose=False)

After ﬁtting, the SVM now has the estimated support vectors and the coefﬁcients of the β in the sv.support\_vectors\_ and sv.coef\_ attributes, respectively. Figure 4.37 shows the two sample classes (white and gray circles) and the line separating them that was found by the maximal margin algorithm. The two parallel dotted lines show the margin. The large circles enclose the support vectors, which are the data elements that are relevant to the solution. Notice that only these elements can touch the edges of the margins.

Figure 4.38 shows what happens when the value of C changes. Increasing this value emphasizes the ξ part of the objective function in Eq.(4.9.0.1). As shown in the top left panel, a small value for C means that the algorithm is willing to accept many support vectors at the expense of maximizing the margin. That is, the proportional amount that predictions are on the wrong side of the margin is more acceptable with smaller C. As the value of C increases, there are fewer support vectors because the optimization process prefers to eliminate support vectors that are far away from the margins and accept fewer of these that encroach into the margin. Note that as the value of C progresses through this ﬁgure, the separating line tilts slightly. 4.9 Support Vector Machines

315

Fig. 4.38 The maximal margin algorithm ﬁnds the separating line that maximizes the margin shown. The elements that touch the margins are the support elements. The dotted elements are not relevant to the solution

4.9.1 Kernel Tricks

Support Vector Machines provide a powerful method to deal with linear separations, but they can also apply to nonlinear boundaries by exploiting the so-called kernel trick. The convex optimization formulation of the SVM includes a dual formulation that leads to a solution that requires only the inner products of the features. The kernel trick is to substitute inner products by nonlinear kernel functions. This can be thought of as mapping the original features onto a possibly inﬁnite-dimensional space of new features. That is, if the data are not linearly separable in two-dimensional space (for example) maybe they are separable in three-dimensional space (or higher)?

To make this concrete, suppose the original input space is R n and we want to use a nonlinear mapping ψ : x ↦ → F where F is an inner product space of higher dimension. The kernel trick is to calculate the inner product in F using a kernel function, K(x i , x j ) = 〈ψ (x i ), ψ (x j ) 〉 . The long way to compute this is to ﬁrst compute ψ (x) and then do the inner product. The kernel-trick way to do it is to use the kernel function and avoid computing ψ . In other words, the kernel function returns what the inner product in F would have returned if ψ had been applied. For example, to achieve an n th polynomial mapping of the input space, we can use κ(x i , x j ) = (x i T x j + θ) n . For example, suppose the input space is R 2 and F = R 4 and we have the following mapping,

ψ (x) : (x 0 , x 1 ) ↦ → (x 0 2 , x 1 2 , x 0 x 1 , x 1 x 0 )

The inner product in F is then,

〈ψ (x), ψ (y) 〉 = 〈 x, y 〉2 316

4

Machine Learning

In other words, the kernel is the square of the inner product in input space. The advantage of using the kernel instead of simply enlarging the feature space is computational because you only need to compute the kernel on all distinct pairs of the input space. The following example should help make this concrete. First we create some Sympy variables,

>>> import sympy as S

>>> x0,x1=S.symbols('x:2',real=True)

>>> y0,y1=S.symbols('y:2',real=True)

Next, we create the ψ function that maps into R 4 and the corresponding kernel function,

>>> psi = lambda x,y: (x\*\*2,y\*\*2,x\*y,x\*y)

>>> kern = lambda x,y: S.Matrix(x).dot(y)\*\*2

Notice that the inner product in R 4 is equal to the kernel function, which only uses the R 2 variables.

>>> print(S.Matrix(psi(x0,x1)).dot(psi(y0,y1))) x0\*\*2\*y0\*\*2 + 2\*x0\*x1\*y0\*y1 + x1\*\*2\*y1\*\*2

>>> print(S.expand(kern((x0,x1),(y0,y1)))) # same as above x0\*\*2\*y0\*\*2 + 2\*x0\*x1\*y0\*y1 + x1\*\*2\*y1\*\*2

Polynomial Regression Using Kernels. Recall our favorite linear regression problem from the regularization chapter,

min ∥ y − X β∥2 β

where X is a n ×m matrix with m > n. As we discussed, there are multiple solutions to this problem. The least-squares solution is the following:

β LS = X T (XX T ) −1 y

Given a new feature vector x, the corresponding estimator for y is the following:

yˆ = x T β LS = x T X T (XX T ) −1 y

Using the kernel trick, the solution can be written more generally as the following:

yˆ = k(x) T K −1 y

where the n × n kernel matrix K replaces XX T and where k(x) is a n-vector of components k(x) = [ κ(x i , x) ] and where K i, j = κ(x i , x j ) for the kernel function κ. With this more general setup, we can substitute κ(x i , x j ) = (x i T x j + θ) n for n th order polynomial regression [4]. Note that ridge regression can also be incorporated 4.9 Support Vector Machines

317

by inverting (K + αI), which can help stabilize poorly conditioned K matrices with a tunable α hyper-parameter [4].

For some kernels, the enlarged F space is inﬁnite-dimensional. Mercer’s conditions provide technical restrictions on the kernel functions. Powerful and well-studied kernels have been implemented in Scikit-learn. The advantage of kernel functions may evaporate for when n → m in which case using the ψ functions instead can be more practicable.

4.10 Dimensionality Reduction

Thefeaturesfromaparticulardatasetthatwillultimatelyproveimportantformachine learning can be difﬁcult to know ahead of time. This is especially true for problems that do not have a strong physical underpinning. The row dimension of the input matrix (X) for ﬁtting data in Scikit-learn is the number of samples and the column dimension is the number of features. There may be a large number of column dimensions in this matrix, and the purpose of dimensionality reduction is to somehow reduce these to only those columns that are important for the machine learning task.

Fortunately, Scikit-learn provides some powerful tools to help uncover the most relevant features. Principal component analysis (PCA) consists of taking the input X matrix and (1) subtracting the mean, (2) computing the covariance matrix, and (3) computing the eigenvalue decomposition of the covariance matrix. For example, if X has more columns than is practicable for a particular learning method, then PCA can reduce the number of columns to a more manageable number. PCA is widely used in statistics and other areas beyond machine learning, so it is worth examining what it does in some detail. First, we need the decomposition module from Scikit-learn.

>>> from sklearn import decomposition

>>> import numpy as np

>>> pca = decomposition.PCA()

Let’s create some very simple data and apply PCA.

>>> x = np.linspace(-1,1,30)

>>> X = np.c\_[x,x+1,x+2] # stack as columns

>>> pca.fit(X)

PCA(copy=True, iterated\_power='auto', n\_components=None, random\_state=None,

svd\_solver='auto', tol=0.0, whiten=False)

>>> print(pca.explained\_variance\_ratio\_) [1.00000000e+00 2.73605815e-32 8.35833807e-34]

Programming Tip

The np.c\_ is a shortcut method for creating stacked column-wise arrays. 318

4

Machine Learning

Fig. 4.39 The top panel shows the columns of the feature matrix and the bottom panel shows the dominant component that PCA has extracted

In this example, the columns are just constant offsets of the ﬁrst column. The explained variance ratio is the percentage of the variance attributable to the transformed columns of X. You can think of this as the information that is relatively concentrated in each column of the transformed matrix X. Figure 4.39 shows the graph of this dominant transformed column in the bottom panel. Note that a constant offset in each of the columns does not change its respective variance and thus, as far as PCA is concerned, the three columns are identical from an information standpoint.

To make this more interesting, let’s change the slope of each of the columns as in the following:

>>> X = np.c\_[x,2\*x+1,3\*x+2,x] # change slopes of columns

>>> pca.fit(X)

PCA(copy=True, iterated\_power='auto', n\_components=None, random\_state=None,

svd\_solver='auto', tol=0.0, whiten=False)

>>> print(pca.explained\_variance\_ratio\_) [1.00000000e+00 3.26962032e-33 3.78960782e-34 2.55413064e-35]

However, changing the slope did not impact the explained variance ratio. Again, there is still only one dominant column. This means that PCA is invariant to both constant offsets and scale changes. This works for functions as well as simple lines,

>>> x = np.linspace(-1,1,30)

>>> X = np.c\_[np.sin(2\*np.pi\*x),

... 2\*np.sin(2\*np.pi\*x)+1, ... 3\*np.sin(2\*np.pi\*x)+2] >>> pca.fit(X) PCA(copy=True, iterated\_power='auto', n\_components=None, random\_state=None,

svd\_solver='auto', tol=0.0, whiten=False)

>>> print(pca.explained\_variance\_ratio\_) [1.00000000e+00 3.70493694e-32 2.51542007e-33]

Once again, there is only one dominant column, which is shown in the bottom panel of Fig.4.40. The top panel shows the individual columns of the feature matrix. 4.10 Dimensionality Reduction

319

Fig. 4.40 The top panel shows the columns of the feature matrix and the bottom panel shows the dominant component that PCA has computed

To sum up, PCA is able to identify and eliminate features that are merely linear transformations of existing features. This also works when there is additive noise in the features, although more samples are needed to separate the uncorrelated noise from between features.

To see how PCA can simplify machine learning tasks, consider Fig.4.41 wherein the two classes are separated along the diagonal. After PCA, the transformed data lie alongasingleaxiswherethetwoclassescanbesplitusingaone-dimensionalinterval, which greatly simpliﬁes the classiﬁcation task. The class identities are preserved under PCA because the principal component is along the same direction that the classesareseparated.Ontheotherhand,iftheclassesareseparatedalongthedirection orthogonal to the principal component, then the two classes become mixed under PCA and the classiﬁcation task becomes much harder. Note that in both cases, the explained\_variance\_ratio\_ is the same because the explained variance ratio does not account for class membership.

PCA works by decomposing the covariance matrix of the data using the Singular Value Decomposition (SVD). This decomposition exists for all matrices and returns the following factorization for an arbitrary matrix A (Fig.4.42),

A = USVT

Because of the symmetry of the covariance matrix, U = V. The elements of the diagonal matrix S are the singular values of A whose squares are the eigenvalues of A T A. The eigenvector matrix U is orthogonal: U T U = I. The singular values are in decreasing order so that the ﬁrst column of U is the axis corresponding to the largest singularvalue.ThisistheﬁrstdominantcolumnthatPCAidentiﬁes.Theentriesofthe covariancematrixareoftheformE(x i x j )where x i and x j aredifferentfeatures. 7 This meansthatthecovariancematrixisﬁlledwithentriesthatattempttouncovermutually

7 Note that these entries are constructed from the data using an estimator of the covariance matrix because we do not have the full probability densities at hand. 320

4

Machine Learning

Fig. 4.41 The left panel shows the original two-dimensional data space of two easily distinguishable classes and the right panel shows the reduced the data space transformed using PCA. Because the two classes are separated along the principal component discovered by PCA, the classes are preserved under the transformation

Fig. 4.42 As compared with Fig.4.41, the two classes differ along the coordinate direction that is orthogonal to the principal component. As a result, the two classes are no longer distinguishable after transformation

correlated relationships between all pairs of columns of the feature matrix. Once these have been tabulated in the covariance matrix, the SVD ﬁnds optimal orthogonal transformationstoalignthecomponentsalongthedirectionsmoststronglyassociated with these correlated relationships. Simultaneously, because orthogonal matrices have columns of unit-length, the SVD collects the absolute squared lengths of these components into the S matrix. In our example above in Fig.4.41, the two feature vectors were obviously correlated along the diagonal, meaning that PCA selected that diagonal direction as the principal component.

We have seen that PCA is a powerful dimension reduction method that is invariant to linear transformations of the original feature space. However, this method performs poorly with transformations that are nonlinear. In that case, there are a wide range of 4.10 Dimensionality Reduction

321

extensionstoPCA,suchasKernelPCA,thatareavailableinScikit-learn,whichallow for embedding parameterized nonlinearities into the PCA at the risk of overﬁtting.

4.10.1 Independent Component Analysis

Independent Component Analysis (ICA) via the FastICA algorithm is also available in Scikit-learn. This method is fundamentally different from PCA in that it is the small differences between components that are emphasized, not the large principal components. This method is adopted from signal processing. Consider a matrix of signals (X) where the rows are the samples and the columns are the different signals. For example, these could be EKG signals from multiple leads on a single patient. The analysis starts with the following model,

X = SAT

(4.10.1.1)

In other words, the observed signal matrix is an unknown mixture (A) of some set of conformable, independent random sources S,

S = [s 1 (t), s 2 (t), . . . , s n (t)]

The distribution on the random sources is otherwise unknown, except there can be at most one Gaussian source, otherwise, the mixing matrix A cannot be identiﬁed because of technical reasons. The problem in ICA is to ﬁnd A in Eq.(4.10.1.1) and thereby un-mix the s i (t) signals, but this cannot be solved without a strategy to reduce the inherent arbitrariness in this formulation.

To make this concrete, let us simulate the situation with the following code,

>>> from numpy import matrix, c\_, sin, cos, pi

>>> t = np.linspace(0,1,250)

>>> s1 = sin(2\*pi\*t\*6)

>>> s2 =np.maximum(cos(2\*pi\*t\*3),0.3)

>>> s2 = s2 - s2.mean()

>>> s3 = np.random.randn(len(t))\*.1

>>> # normalize columns

>>> s1=s1/np.linalg.norm(s1)

>>> s2=s2/np.linalg.norm(s2)

>>> s3=s3/np.linalg.norm(s3)

>>> S =c\_[s1,s2,s3] # stack as columns

>>> # mixing matrix

>>> A = matrix([[ 1, 1,1], ... [0.5, -1,3], ... [0.1, -2,8]])

>>> X= S\*A.T # do mixing 322

4

Machine Learning

Fig. 4.43 The left column shows the original signals and the right column shows the mixed signals. The object of ICA is to recover the left column from the right

The individual signals (s i (t)) and their mixtures (X i (t)) are shown in Fig.4.43. To recover the individual signals using ICA, we use the FastICA object and ﬁt the parameters on the X matrix,

>>> from sklearn.decomposition import FastICA

>>> ica = FastICA()

>>> # estimate unknown S matrix

>>> S\_=ica.fit\_transform(X)

The results of this estimation are shown in Fig.4.44, showing that ICA is able to recover the original signals from the observed mixture. Note that ICA is unable to distinguish the signs of the recovered signals or preserve the order of the input signals.

To develop some intuition as to how ICA accomplishes this feat, consider the following two-dimensional situation with two uniformly distributed independent variables, u x , u y ∼ U [ 0, 1 ] . Suppose we apply the following orthogonal rotation matrix to these variables,

u x ′ cos( φ ) − sin( φ ) ux = [ u y ′ ] [ sin( φ ) cos( φ ) ] [ u y ] 4.10 Dimensionality Reduction

323

Fig. 4.44 The left column shows the original signals and the right column shows the signals that ICA was able to recover. They match exactly, outside of a possible sign change

Fig. 4.45 The left panel shows two classes labeled on the u x , u y uniformly independent random variables. The right panel shows these random variables after a rotation, which removes their mutual independence and makes it hard to separate the two classes along the coordinate directions

The so-rotated variables u x ′ , u y ′ are no longer independent, as shown in Fig.4.45. Thus, one way to think about ICA is as a search through orthogonal matrices so that the independence is restored. This is where the prohibition against Gaussian 324

4

Machine Learning

distributions arises. The two-dimensional Gaussian distribution of independent variables is proportional the following:

1 f (x) ∝ exp(− x) xT 2

Now, if we similarly rotated the x vector as,

y = Qx

the resulting density for y is obtained by plugging in the following:

x=QT y

because the inverse of an orthogonal matrix is its transpose, we obtain

1 1 f (y) ∝ exp(− QQ T y) = exp(− y) yT yT 2 2

In other words, the transformation is lost on the y variable. This means that ICA cannot search over orthogonal transformations if it is blind to them, which explains the restriction of Gaussian random variables. Thus, ICA is a method that seeks to maximize the non-Gaussian-ness of the transformed random variables. There are many methods to doing this, some of which involve cumulants and others that use the negentropy,

J (Y) = H(Z) − H(Y)

where H(Z) is the information entropy of the Gaussian random variable Z that has the same variance as Y. Further details would take us beyond our scope, but that is the outline of how the FastICA algorithm works.

The implementation of this method in Scikit-learn includes two different ways of extracting more than one independent source component. The deﬂation method iteratively extracts one component at a time using a incremental normalization step. The parallel method also uses the single-component method but carries out normalization of all the components simultaneously, instead of for just the newly computed component. Because ICA extracts independent components, a whitening step is used beforehand to balance the correlated components from the data matrix. Whereas PCA returns uncorrelated components along dimensions optimal for Gaussian random variables, ICA returns components that are as far from the Gaussian density as possible.

The left panel on Fig.4.45 shows the original uniform random sources. The white and black colors distinguish between two classes. The right panel shows the mixture of these sources, which is what we observe as input features. The top row of Fig.4.46 shows the PCA (left) and ICA (right) transformed data spaces. Notice that ICA is able to un-mix the two random sources whereas PCA transforms along the dominant 4.10 Dimensionality Reduction

325

Fig. 4.46 The panel on the top left shows two classes in a plane after a rotation. The bottom left panel shows the result of dimensionality reduction using PCA, which causes mixing between the two classes. The top right panel shows the ICA transformed output and the lower right panel shows that, because ICA was able to un-rotate the data, the lower dimensional data maintains the separation between the classes

diagonal. Because ICA is able to preserve the class membership, the data space can be reduced to two nonoverlapping sections, as shown. However, PCA cannot achieve a similar separation because the classes are mixed along the dominant diagonal that PCA favors as the main component in the decomposition.

For a good principal component analysis treatment, see [5–8]. Independent Component Analysis is discussed in more detail in [9].

4.11 Clustering

Clustering is the simplest member of a family of machine learning methods that do not require supervision to learn from data. Unsupervised methods have training sets that do not have a target variable. These unsupervised learning methods rely upon a meaningful metric to group data into clusters. This makes it an excellent exploratory 326

4

Machine Learning

Fig. 4.47 The four clusters are pretty easy to see in this example and we want clustering methods to determine the extent and number of such clusters automatically

data analysis method because there are very few assumptions built into the method itself. In this section, we focus on the popular K-means clustering method that is available in Scikit-learn.

Let’s manufacture some data to get going with make\_blobs from Scikit-learn. Figure 4.47 shows some example clusters in two dimensions. Clustering methods work by minimizing the following objective function,

J = ∑ ∑ ∥ x i − μ k ∥2 k i

The distortion for the k th cluster is the summand,

∑ ∥ x i − μ k ∥2 i

Thus, clustering algorithms work to minimize this by adjusting the centers of the individual clusters, μ k . Intuitively, each μ k is the center of mass of the points in the cloud. The Euclidean distance is the typical metric used for this,

∥ x ∥ 2 = ∑ xi 2

There are many clever algorithms that can solve this problem for the best μ k clustercenters. The K-means algorithm starts with a user-speciﬁed number of K clusters to optimize over. This is implemented in Scikit-learn with the KMeans object that follows the usual ﬁtting conventions in Scikit-learn,

>>> from sklearn.cluster import KMeans

>>> kmeans = KMeans(n\_clusters=4)

>>> kmeans.fit(X)

KMeans(algorithm='auto', copy\_x=True, init='k-means++', max\_iter=300, 4.11 Clustering

327

n\_clusters=4, n\_init=10, n\_jobs=None, precompute\_distances='auto',

random\_state=None, tol=0.0001, verbose=0)

where we have chosen K = 4. How do we choose the value of K? This is the eternal question of generalization versus approximation—too many clusters provide great approximation but bad generalization. One way to approach this problem is to compute the mean distortion for increasingly larger values of K until it no longer makes sense. To do this, we want to take every data point and compare it to the centers of all the clusters. Then, take the smallest value of this across all clusters and average those. This gives us an idea of the overall mean performance for the K clusters. The following code computes this explicitly.

Programming Tip

The cdist function from Scipy computes all the pairwise differences between the two input collections according to the speciﬁed metric.

>>> from scipy.spatial.distance import cdist

>>> m\_distortions=[]

>>> for k in range(1,7):

...

kmeans

=

KMeans(n\_clusters=k)

... \_=kmeans.fit(X) ... tmp=cdist(X,kmeans.cluster\_centers\_,'euclidean') ... m\_distortions.append(sum(np.min(tmp,axis=1))/X.shape[0]) ...

Note that the code above uses the cluster\_centers\_, which are estimated from K-means algorithm. The resulting Fig.4.48 shows the point of diminishing returns for added additional clusters.

Another ﬁgure-of-merit is the silhouette coefﬁcient, which measures how compact and separated the individual clusters are. To compute the silhouette coefﬁcient, we need to compute the mean intra-cluster distance for each sample (a i ) and the mean

Fig. 4.48 The mean distortion shows that there is a diminishing value in using more clusters 328

4

Machine Learning

Fig. 4.49 The shows how the silhouette coefﬁcient varies as the clusters move closer and become more compact

distance to the next nearest cluster (b i ). Then, the silhouette coefﬁcient for the ith sample is (Fig.4.49)

b i −ai sc i = max(a i , b i )

The mean silhouette coefﬁcient is just the mean of all these values over all the samples. The best value is one and the worst is negative one, with values near zero indicating overlapping clusters and negative values showing that samples have been incorrectly assigned to the wrong cluster. This ﬁgure-of-merit is implemented in Scikit-learn as in the following:

>>> from sklearn.metrics import silhouette\_score

Figure 4.50 shows how the silhouette coefﬁcient varies as the clusters become more dispersed and/or closer together.

K-means is easy to understand and to implement, but can be sensitive to the initial choice of cluster-centers. The default initialization method in Scikit-learn uses a very effective and clever randomization to come up with the initial clustercenters. Nonetheless, to see why initialization can cause instability with K-means, consider the following Fig.4.50. In Fig.4.50, there are two large clusters on the left and a very sparse cluster on the far right. The large circles at the centers are the cluster-centers that K-means found. Given K = 2, how should the cluster-centers be chosen? Intuitively, the ﬁrst two clusters should have their own cluster-center somewhere between them and the sparse cluster on the right should have its own cluster-center. 8 Why isn’t this happening?

8 Note that we are using the init=random keyword argument for this example in order to illustrate this. 4.11 Clustering

329

Fig. 4.50 The large circles indicate the clustercenters found by the K-means algorithm

TheproblemisthattheobjectivefunctionforK-meansistradingthedistanceofthe far-off sparse cluster with its small size. If we keep increasing the number of samples in the sparse cluster on the right, then K-means will move the cluster-centers out to meet them, as shown in Fig.4.50. That is, if one of the initial cluster-centers was right in the middle of the sparse cluster, the algorithm would have immediately captured it and then moved the next cluster-center to the middle of the other two clusters (bottom panel of Fig.4.50). Without some thoughtful initialization, this may not happen and the sparse cluster would have been merged into the middle cluster (top panel of Fig.4.50). Furthermore, such problems are hard to visualize with high-dimensional clusters. Nonetheless, K-means is generally very fast, easy to interpret, and easy to understand. It is straightforward to parallelize using the n\_jobs keyword argument so that many initial cluster-centers can be easily evaluated. Many extensions of Kmeans use different metrics beyond Euclidean and incorporate adaptive weighting of features. This enables the clusters to have ellipsoidal instead of spherical shapes.

4.12 Ensemble Methods

With the exception of the random forest, we have so far considered machine learning models as stand-alone entities. Combinations of models that jointly produce a classiﬁcation are known as ensembles. There are two main methodologies that create ensembles: bagging and boosting.

4.12.1 Bagging

Bagging refers to bootstrap aggregating, where bootstrap here is the same as we discussed in Sect.3.10. Basically, we resample the data with replacement and then train a classiﬁer on the newly sampled data. Then, we combine the outputs of each of the individual classiﬁers using a 9 (for discrete outputs) or a weighted average (for continuous outputs). This combination is particularly effective for models that 330

4

Machine Learning

Fig. 4.51 Two regions in the plane are separated by a nonlinear boundary. The training data is sampled from this plane. The objective is to correctly classify the sosampled data

are easily inﬂuenced by a single data element. The resampling process means that these elements cannot appear in every bootstrapped training set so that some of the models will not suffer these effects. This makes the so-computed combination of outputs less volatile. Thus, bagging helps reduce the collective variance of individual high-variance models.

To get a sense of bagging, let’s suppose we have a two-dimensional plane that is partitioned into two regions with the following boundary: y = −x + x 2 . Pairs of (x i , y i ) points above this boundary are labeled one and points below are labeled zero. Figure 4.51 shows the two regions with the nonlinear separating boundary as the black curved line.

The problem is to take samples from each of these regions and classify them correctly using a perceptron (see Sect.4.13). A perceptron is the simplest possible linear classiﬁer that ﬁnds a line in the plane to separate two purported categories. Because the separating boundary is nonlinear, there is no way that the perceptron can completely solve this problem. The following code sets up the perceptron available in Scikit-learn.

>>> from sklearn.linear\_model import Perceptron

>>> p=Perceptron()

>>> p

Perceptron(alpha=0.0001, class\_weight=None, early\_stopping=False, eta0=1.0, fit\_intercept=True, max\_iter=None, n\_iter=None, n\_iter\_no\_change=5, n\_jobs=None, penalty=None, random\_state=0, shuffle=True, tol=None,

validation\_fraction=0.1, verbose=0, warm\_start=False)

The training data and the resulting perceptron separating boundary are shown in Fig.4.52. The circles and crosses are the sampled training data and the gray separating line is the perceptron’s separating boundary between the two categories. The black squares are those elements in the training data that the perceptron misclassiﬁed. Because the perceptron can only produce linear separating boundaries, and the boundary in this case is nonlinear, the perceptron makes mistakes near where the boundary curves. The next step is to see how bagging can improve upon this by using multiple perceptrons. 4.12 Ensemble Methods

331

Fig. 4.52 The perceptron ﬁnds the best linear boundary between the two classes

The following code sets up the bagging classiﬁer in Scikit-learn. Here we select only three perceptrons. Figure 4.53 shows each of the three individual classiﬁers and the ﬁnal bagged classiﬁer in the panel on the bottom right. As before, the black circles indicate misclassiﬁcations in the training data. Joint classiﬁcations are determined by majority voting.

>>> from sklearn.ensemble import BaggingClassifier

>>> bp = BaggingClassifier(Perceptron(),max\_samples=0.50,n\_estimators=3)

>>> bp

BaggingClassifier(base\_estimator=Perceptron(alpha=0.0001, class\_weight=None, early\_stopping=False, eta0=1.0,

fit\_intercept=True, max\_iter=None, n\_iter=None, n\_iter\_no\_change=5, n\_jobs=None, penalty=None, random\_state=0, shuffle=True, tol=None,

validation\_fraction=0.1, verbose=0, warm\_start=False),

bootstrap=True, bootstrap\_features=False, max\_features=1.0,

max\_samples=0.5, n\_estimators=3, n\_jobs=None, oob\_score=False,

random\_state=None, verbose=0, warm\_start=False)

The BaggingClassifier can estimate its own out-of-sample error if passed the oob\_score=True ﬂag upon construction. This keeps track of which samples were used for training and which were not, and then estimates the out-of-sample error using those samples that were unused in training. The max\_samples keyword argument speciﬁes the number of items from the training set to use for the base classiﬁer. The smaller the max\_samples used in the bagging classiﬁer, the better the out-of-sample error estimate, but at the cost of worse in-sample performance. Of course, this depends on the overall number of samples and the degrees-of-freedom in each individual classiﬁer. The VC dimension surfaces again!

4.12.2 Boosting

As we discussed, bagging is particularly effective for individual high-variance classiﬁers because the ﬁnal majority-vote tends to smooth out the individual classiﬁers and 332

4

Machine Learning

Fig. 4.53 Each panel with the single gray line is one of the perceptrons used for the ensemble bagging classiﬁer on the lower right

produce a more stable collaborative solution. On the other hand, boosting is particularly effective for high-bias classiﬁers that are slow to adjust to new data. On the one hand, boosting is similar to bagging in that it uses a majority-voting (or averaging for numeric prediction) process at the end; and it also combines individual classiﬁers of the same type. On the other hand, boosting is serially iterative, whereas the individual classiﬁers in bagging can be trained in parallel. Boosting uses the misclassiﬁcations of prior iterations to inﬂuence the training of the next iterative classiﬁer by weighting those misclassiﬁcations more heavily in subsequent steps. This means that, at every step, boosting focuses more and more on speciﬁc misclassiﬁcations up to that point, letting the prior classiﬁcations be carried by earlier iterations.

The primary implementation for boosting in Scikit-learn is the Adaptive Boosting (AdaBoost) algorithm, which does classiﬁcation (AdaBoostClassifier) and regression (AdaBoostRegressor). The ﬁrst step in the basic AdaBoost algorithm is to initialize the weights over each of the training set indices, D 0 (i) = 1/n where there are n elements in the training set. Note that this creates a discrete uniform distribution over the indices, not over the training data { (x i , y i ) } itself. In other words, if there are repeated elements in the training data, then each gets its own weight. The next step is to train the base classiﬁer h k and record the classiﬁcation error at the kth iteration, k . Two factors can next be calculated using k , 4.12 Ensemble Methods

333

1 1 − k log α k = 2 k

and the normalization factor,

Z k = 2 √ k (1 − k )

For the next step, the weights over the training data are updated as in the following:

1 D k + 1 (i) = D k (i) exp (−α k y i h k (x i )) Zk

The ﬁnal classiﬁcation result is assembled using the α k factors, g = sgn( ∑ k α k h k ).

To re-do the problem above using boosting with perceptrons, we set up the AdaBoost classiﬁer in the following:

>>> from sklearn.ensemble import AdaBoostClassifier

>>> clf=AdaBoostClassifier(Perceptron(),n\_estimators=3,

... algorithm='SAMME', ... learning\_rate=0.5) >>> clf AdaBoostClassifier(algorithm='SAMME', base\_estimator=Perceptron(alpha=0.0001, class\_weight=None, early\_stopping=False, eta0=1.0,

fit\_intercept=True, max\_iter=None, n\_iter=None, n\_iter\_no\_change=5,

n\_jobs=None, penalty=None, random\_state=0, shuffle=True, tol=None,

validation\_fraction=0.1, verbose=0, warm\_start=False),

learning\_rate=0.5, n\_estimators=3, random\_state=None)

The learning\_rate above controls how aggressively the weights are updated. The resulting classiﬁcation boundaries for the embedded perceptrons are shown in Fig.4.54. Compare this to the lower right panel in Fig.4.53. The performance for both cases is about the same.

Fig. 4.54 The individual perceptron classiﬁers embedded in the AdaBoost classiﬁer are shown along with the misclassiﬁed points (in black). Compare this to the lower right panel of Fig.4.53 334

4

Machine Learning

4.13 Deep Learning

Neural networks have a long history going back to the 1960s, but the recent availability of large-scale, high-quality data and new parallel computing infrastructures have reinvigorated neural networks in terms of size and complexity. This new reinvigoration, with many new and complex topologies, is called deep learning. There have been exciting developments in image and video processing, speech recognition, and automated video captioning based on deep learning systems. However, this is still a very active area of research. Fortunately, big companies with major investments in this area have made much of their research software open source (e.g., Tensorﬂow, PyTorch), with corresponding Python bindings. To build up our understanding of neural networks, we begin with Rosenblatt’s 1960 Perceptron.

Perceptron Learning. The perceptron is the primary ancestor of the most popular deep learning technologies (i.e., multilayer perceptron) and it is the best place to start as it will reveal the basic mechanics and themes of more complicated neural networks. The job of the perceptron is to create a linear classiﬁer that can separate points in R n between two classes. The basic idea is that given a set of associations:

{ (x 0 , y 0 ), . . . , (x m , y m ) }

where each x ∈ R n−1 is augmented with a unit-entry to account for an offset term, and a set of weights w ∈ R n , compute the following as an estimate of the label

y ∈ { −1, 1 } .

yˆ = w T x

Concisely, this means that we want w such that

C 2 wT x i ≷ 0

C 1

where x i is in class C 2 if x i T w > 0 and class C 1 otherwise. To determine these weights, we apply the following learning rule:

w (k + 1) = w (k) − (y − yˆ )xi

The output of the perceptron can be summarized as

yˆ = sgn(x i T w)

The sign is the activation function of the perceptron. With this setup, we can write out the perceptron’s output as the following: 4.13 Deep Learning

335

>>> import numpy as np

>>> def yhat(x,w):

... return np.sign(np.dot(x,w)) ...

Let us create some fake data to play with:

>>> npts = 100

>>> X=np.random.rand(npts,2)\*6-3 # random scatter in 2-d plane

>>> labels=np.ones(X.shape[0],dtype=np.int) # labels are 0 or 1

>>> labels[(X[:,1]<X[:,0])]=-1

>>> X = np.c\_[X,np.ones(X.shape[0])] # augment with offset term

Note that we added a column of ones to account for the offset term. Certainly, by our construction, this problem is linearly separable, so let us see if the perceptron can ﬁnd the boundary between the two classes. Let us start by initializing the weights,

>>> winit = np.random.randn(3)

and then apply the learning rule,

>>> w= winit

>>> for i,j in zip(X,labels): ... w = w - (yhat(i,w)-j)\*i ...

Note that we are taking a single ordered pass through the data. In practice, we would have randomly shufﬂed the input data to ensure that there is no incidental structure in the order of the data that would inﬂuence training. Now, let us examine the accuracy of the perceptron,

>>> from sklearn.metrics import accuracy\_score

>>> print(accuracy\_score(labels,[yhat(i,w) for i in X]))

0.96

We can re-run the training rule over the data to try to improve the accuracy. A pass through the data is called an epoch.

>>> for i,j in zip(X,labels): ... w = w - (yhat(i,w)-j)\*i ...

>>> print(accuracy\_score(labels,[yhat(i,w) for i in X]))

0.98

Note that our initial weight for this epoch is the last weight from the previous pass. It is common to randomly shufﬂe the data between epochs. More epochs will result in better accuracy in this case. 336

4

Machine Learning

Fig. 4.55 The softsign function is a smooth approximation to the sign function. This makes it easier to differentiate for backpropagation

We can re-do this entire example with keras. First, we deﬁne the model,

>>> from keras.models import Sequential

>>> from keras.layers import Dense

>>> from keras.optimizers import SGD

>>> model = Sequential()

>>> model.add(Dense(1, input\_shape=(2,), activation='softsign'))

>>> model.compile(SGD(), 'hinge')

Note that we use the softsign activation instead of the sgn that we used earlier because we need a differentiable activation function. Given the form of the weight update in perceptron learning, it is equivalent to the hinge loss function. Stochastic gradient descent (SGD) is chosen for updating the weights. The softsign function is deﬁned as the following:

x s(t) = 1 + | x |

We can pull it out from the tensorflow backend that keras uses as in the following: plotted in Fig.4.55

>>> import tensorflow as tf

>>> x = tf.placeholder('float')

>>> xi = np.linspace(-10,10,100)

>>> with tf.Session() as s:

... y\_=(s.run(tf.nn.softsign(x),feed\_dict={x:xi})) ...

Next, all we have to do is fit the model on data,

>>> h=model.fit(X[:,:2], labels, epochs=300, verbose=0)

The h variable is the history that contains the internal metrics and parameters involved in the fit training phase. We can extract the trajectory of the loss function from this history and draw the loss in Fig.4.56. 4.13 Deep Learning

337

Fig. 4.56 Trajectory of the loss function

Fig. 4.57 The basic multilayer perceptron has a single hidden layer between input and output. Each of the arrows has a multiplicative weight associated with it

Multilayer Perceptron. The multilayer perceptron (MLP) generalizes the perceptron by stacking them as fully connected individual layers. The basic topology is shown in Fig.4.57. In the previous section we saw that the basic perceptron could generate a linear boundary for data that is linearly separable. The MLP can create more complex nonlinear boundaries. Let us examine the moons dataset from scikit-learn,

>>> from sklearn.datasets import make\_moons

>>> X, y = make\_moons(n\_samples=1000, noise=0.1, random\_state=1234)

The purpose of the noise term is to make data for each of the categories harder to disambiguate. These data are shown in Fig.4.58.

The challenge for the MLP is to derive a nonlinear boundary between these two classes. We construct our MLP using keras,

>>> from keras.optimizers import Adam

>>> model = Sequential()

>>> model.add(Dense(4,input\_shape=(2,),activation='sigmoid'))

>>> model.add(Dense(2,activation='sigmoid'))

>>> model.add(Dense(1,activation='sigmoid'))

>>> model.compile(Adam(lr=0.05), 'binary\_crossentropy') 338

4

Machine Learning

Fig. 4.58 Data from make\_moons

This MLP has three layers. The input layer has four units and the next layer has two units and the output layer has one unit to distinguish between the two available classes. Instead of plain stochastic gradient descent, we use the more advanced Adam optimizer. A quick summary of the model elements and parameters comes from the model.summary() method,

>>> model.summary() \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Layer (type) Output Shape Param # ================================================================= dense\_2 (Dense) (None, 4) 12 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ dense\_3 (Dense) (None, 2) 10 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ dense\_4 (Dense) (None, 1) 3 ================================================================= Total params: 25 Trainable params: 25 Non-trainable params: 0 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

As usual, we split the input data into train and test sets,

>>> from sklearn.model\_selection import train\_test\_split

>>> X\_train,X\_test,y\_train,y\_test=train\_test\_split(X,y,

... test\_size=0.3, ... random\_state=1234)

Thus, we reserve 30% of the data for testing. Next, we train the MLP,

>>> h=model.fit(X\_train, y\_train, epochs=100, verbose=0)

To compute the accuracy metric using the test set, we need to compute the model prediction on the this set.

>>> y\_train\_ = model.predict\_classes(X\_train,verbose=0)

>>> y\_test\_ = model.predict\_classes(X\_test,verbose=0) 4.13 Deep Learning

339

Fig. 4.59 The derived boundary separates the two classes

>>> print(accuracy\_score(y\_train,y\_train\_))

1.0

>>> print(accuracy\_score(y\_test,y\_test\_))

0.9966666666666667

Tovisualizetheso-derivedboundarybetweenthesetwoclasses,weusethecontourf function from matplotlib which generates a ﬁlled contour plot shown in Fig.4.59.

Instead of computing the accuracy separately, we can assign it as a metric for keras to track by supplying it on the compile step, as in the following:

>>> model.compile(Adam(lr=0.05), ... 'binary\_crossentropy', ... metrics=['accuracy'])

Then, we can train again,

>>> h=model.fit(X\_train, y\_train, epochs=100, verbose=0)

Now, we can evaluate the model on the test data,

>>> loss,acc=model.evaluate(X\_test,y\_test,verbose=0)

>>> print(acc)

0.9966666666666667

where loss is the loss function and acc is the corresponding accuracy. The documentation has other metrics that can be speciﬁed during the compile step.

Backpropagation. We have seen that the MLP can generate complicated nonlinear boundaries for classiﬁcation problems. The key algorithm underpinning MLP is backpropagation. The idea is that when we stack layers into the MLP, we are applying function composition, which basically means we take the output of one function and then feed it into the input of another. 340

4

Machine Learning

h = ( f ◦ g)(x) = f (g(x))

For example, for the simple perceptron, we have g(x) = w T x and f (x) = sgn(x). The key property of this composition is that derivatives use the chain rule from calculus.

h ′ (x) = f ′ (g(x))g ′ (x)

Notice this has turned the differentiation operation into a multiplication operation. Explaining backpropagation in general is a notational nightmare, so let us see if we can get the main idea from a speciﬁc example. Consider the following two-layer MLP with one input and one output.

There is only one input (x 1 ). The output of the ﬁrst layer is

z 1 = f (x 1 w 1 + b 1 ) = f (p 1 )

where f is the sigmoid function and b 1 is the bias term. The output of the second layer is

z 2 = f (z 1 w 2 + b 2 ) = f (p 2 )

To keep it simple, let us suppose that the loss function for this MLP is the squared error,

1 J = − y)2 (z2 2

where y is the target label. Backpropagation has two phases. The forward phase computes the MLP loss function given the values of the inputs and corresponding weights. The backward phase applies the incremental weight updates to each weight based on the forward phase. To implement gradient descent, we have to calculate the derivative of the loss function with respect to each of the weights.

∂J ∂z 2 ∂ p2 ∂J = ∂w 2 ∂z 2 ∂ p 2 ∂w2

The ﬁrst term is the following:

∂J =z2 −y ∂z2

The second term is the following:

∂z2 = f ′ (p 2 ) = f (p 2 )(1 − f (p 2 )) ∂ p2 4.13 Deep Learning

341

Note that by property of the sigmoid function, we have f ′ (x) = (1 − f (x)) f (x). The third term is the following:

∂ p2 =z1 ∂w2

Thus, the update for w 2 is the following:

Δw 2 ∝ (z 2 − y)z 1 (1 − z 2 )z2

The corresponding analysis fo b 2 gives the following:

Δb 2 = (z 2 − y)z 2 (1 − z 2 )

Let’s keep going backward to w 1 ,

∂J ∂z 2 ∂ p 2 ∂z 1 ∂ p1 ∂J = ∂w 1 ∂z 2 ∂ p 2 ∂z 1 ∂ p 1 ∂w1

The ﬁrst new term is the following:

∂ p2 =w2 ∂z1

and then the next two terms,

∂z1 = f (p 1 )(1 − f (p 1 )) = z 1 (1 − z 1 ) ∂ p1

∂ p1 =x1 ∂w1

This makes the update for w 1 ,

Δw 1 ∝ (z 2 − y)z 2 (1 − z 2 )w 2 z 1 (1 − z 1 )x1

To understand why this is called backpropagation, we can deﬁne

δ 2 := (z 2 − y)z 2 (1 − z 2 )

This makes the weight update for w 2 ,

Δw 2 ∝ δ 2 z1 342

4

Machine Learning

This means that the weight update for w 2 is proportional to the output of the prior layer (z 1 ) and a factor that accounts steepness of the activation function. Likewise, the weight update for w 1 can be written as the following:

Δw 1 ∝ δ 1 x1

where

δ 1 := δ 2 w 2 z 1 (1 − z 1 )

Note that this weight update is proportional to the input (prior layer’s output) just as the weight update for w 2 was proportional to the prior layer output z 1 . Also, the δ factors propagate recursively backward to the input layer. These characteristics permit efﬁcient numerical implementations for large networks because the subsequent computations are based on prior calculations. This also means that each individual unit’s calculations are localized upon the output of the prior layer. This helps segregate the individual processing behavior of each unit within each layer.

Functional Deep Learning. Keras has an alternative API that makes it possible to understand the performance of neural networks using the composition of functions ideas we discussed. The key objects for this functional interpretation are the Input object and the Model object.

>>> from keras.layers import Input

>>> from keras.models import Model

>>> import keras.backend as K

We can re-create the data from our earlier classiﬁcation example

>>> from sklearn.datasets import make\_moons

>>> X, y = make\_moons(n\_samples=1000, noise=0.1, random\_state=1234)

The ﬁrst step is to construct a placeholder for the input using the Input object,

>>> inputs = Input(shape=(2,))

Next, we can stack the Dense layers as before but now tie their inputs to the previous layer’s outputs by calling Dense as a function.

>>> l1=Dense(3,input\_shape=(2,),activation='sigmoid')(inputs)

>>> l2=Dense(2,input\_shape=(3,),activation='sigmoid')(l1)

>>> outputs=Dense(1,input\_shape=(3,),activation='sigmoid')(l1)

This means that output = ( 2 ◦ 1 )(input) where 1 and 2 are the middle layers. With that established, we collect the individual pieces in the Model object and then fit and train as usual.

>>> model = Model(inputs=inputs,outputs=outputs)

>>> model.compile(Adam(lr=0.05), 4.13 Deep Learning

343

Fig. 4.60 The embedded representation of the input just before the ﬁnal output that shows the internal divergence of the two target classes

... ...

'binary\_crossentropy', metrics=['accuracy'])

>>> h=model.fit(X\_train, y\_train, epochs=500, verbose=0)

This gives the same result as before. The advantage of the functional perspective is that now we can think of the individual layers as mappings between multidimensional R n spaces. For example, 1 : R 2 → ↦ R 3 and 2 : R 3 → ↦ R 2 . Now, we can investigate the performance of the network from the inputs just up until the ﬁnal mapping to R at the output by deﬁning the functional mapping ( 2 ◦ 1 )(inputs) : R 2 → ↦ R 2 , as shown in Fig.4.60.

To get this result, we have to deﬁne a keras function using the inputs.

>>> l2\_function = K.function([inputs], [l2])

>>> # functional mapping just before output layer

>>> l2o=l2\_function([X\_train])

the l2o list contains the output of the l2 layer that is shown in Fig.4.60.

4.13.1 Introduction to Tensorﬂow

Tensorﬂow is the leading deep learning framework. It is written in C++ with Python bindings. Although we will primarily use the brilliant Keras abstraction layer to compose our neural networks with Tensorﬂow providing the backed computing, it is helpful to see how Tensorﬂow itself works and how to interact with it, especially for later debugging. To get started, import Tensorﬂow using the recommended convention.

>>> import tensorflow as tf 344

4

Machine Learning

Fig. 4.61 Flow diagram for adder

Tensorﬂow is graph-based. We have to assemble a computational graph. To get started, let’s deﬁne some constants,

>>> # declare constants

>>> a = tf.constant(2)

>>> b = tf.constant(3)

The context manager (i.e., the with statement) is the recommended way to create a session variable, which is a realization of the computational graph that is composed of operations and tensor data objects. In this context, a tensor is another word for a multidimensional matrix.

>>> # default graph using the context manager

>>> with tf.Session() as sess:

... print('a= ',a.eval()) ... print('b= ',b.eval()) ... print("a+b",sess.run(a+b)) ...

a= 2 b= 3 a+b 5

Thus, we can do some basic arithmetic on the declared variables. We can abstract the graph using placeholders. For example, to implement the computational graph shown in Fig.4.61, we can deﬁne the following:

>>> a = tf.placeholder(tf.int16)

>>> b = tf.placeholder(tf.int16)

Next, we deﬁne the addition operation in the graph,

>>> # declare operation

>>> adder = tf.add(a,b)

Then, we compose and execute the graph using the context manager,

>>> # default graph using context manager

>>> with tf.Session() as sess:

... print (sess.run(adder, feed\_dict={a: 2, b: 3})) ...

5 4.13 Deep Learning Fig. 4.62 Flow diagram for multiplier

This works with matrices also, with few changes (Fig.4.62)

>>> import numpy as np

>>> a = tf.placeholder('float',[3,5])

>>> b = tf.placeholder('float',[3,5])

>>> adder = tf.add(a,b)

>>> with tf.Session() as sess:

... b\_ = np.arange(15).reshape((3,5)) ... print(sess.run(adder,feed\_dict={a:np.ones((3,5)), ... b:b\_})) ...

[[ 1. 2. 3. 4. 5.]

[ 6. 7. 8. 9. 10.] [11. 12. 13. 14. 15.]]

Matrix operations like multiplication are also implemented,

>>> # the None dimension leaves it variable

>>> b = tf.placeholder('float',[5,None])

>>> multiplier = tf.matmul(a,b)

>>> with tf.Session() as sess:

... b\_ = np.arange(20).reshape((5,4)) ... print(sess.run(multiplier,feed\_dict={a:np.ones((3,5)), ... b:b\_})) ...

[[40. 45. 50. 55.]

[40. 45. 50. 55.] [40. 45. 50. 55.]]

The individual computational graphs can be stacked as shown in Fig.4.63.

>>> b = tf.placeholder('float',[3,5])

>>> c = tf.placeholder('float',[5,None])

>>> adder = tf.add(a,b)

>>> multiplier = tf.matmul(adder,c)

>>> with tf.Session() as sess:

345

... b\_ = np.arange(15).reshape((3,-1)) ... c\_ = np.arange(20).reshape((5,4)) ... print(sess.run(multiplier,feed\_dict={a:np.ones((3,5)), ... b:b\_, 346

4

Machine Learning

Fig. 4.63 Flow diagram for adder and multiplier

...

...

[[160. 175. 190. 205.]

[360. 400. 440. 480.] [560. 625. 690. 755.]]

c:c\_}))

Optimizers. To compute the parameters of complicated neural networks, a wide variety of optimization algorithms are also implemented in Tensorﬂow. Consider the classic least-squares problem: Find x that minimizes

min ∥ Ax − b ∥2 x

First, we have to deﬁne a variable that we want the optimizer to solve for,

>>> x = tf.Variable(tf.zeros((3,1)))

Next, we create sample matrices A and b,

>>> A = tf.constant([6,6,4,

... 3,4,0, ... 7,2,2, ... 0,2,1, ... 1,6,3],'float',shape=(5,3)) >>> b = tf.constant([1,2,3,4,5],'float',shape=(5,1))

In neural network terminology, the output of the model (Ax) is called the activation,

>>> activation = tf.matmul(A,x)

The job of the optimizer is to minimize the squared distance between the activation and the b vector. Tensorﬂow implements primitives like reduce\_sum to compute the square difference as a cost variable.

>>> cost = tf.reduce\_sum(tf.pow(activation-b,2))

With all that deﬁned, we can construct the speciﬁc Tensorﬂow optimizer we want,

>>> learning\_rate = 0.001

>>> optimizer=tf.train.GradientDescentOptimizer(learning\_rate).minimize(cost) 4.13 Deep Learning

347

Fig. 4.64 Iterative costs as gradient descent algorithm computes solution. Note that we are showing only a slice of all the values computed

The learning\_rate is an embedded parameter for the GradientDescent Optimizer gradient descent algorithm. Next, we have to initialize all the variables (Fig.4.64),

>>> init=tf.global\_variables\_initializer()

and create the session, without the context manager, just to show that the context manager is not a requirement,

>>> sess = tf.Session()

>>> sess.run(init)

>>> costs=[]

>>> for i in range(500):

... costs.append(sess.run(cost)) ... sess.run(optimizer) ...

Note that we have to iterate over the optimizer to get it to step-wise work through the gradient descent algorithm. As an illustration, we can plot the change in the cost function as it iterates.

The ﬁnal answer after all the iterations is the following:

>>> print (x.eval(session=sess)) [[-0.08000698] [ 0.6133011 ] [ 0.09500197]]

Because this is a classic problem, we know how to solve it analytically as in the following: 348

4

Machine Learning

>>> # least squares solution

>>> A\_=np.matrix(A.eval(session=sess))

>>> print (np.linalg.inv(A\_.T\*A\_)\*(A\_.T)\*b.eval(session=sess)) [[-0.07974136] [ 0.6141343 ] [ 0.09303147]]

which is pretty close to what we found by iterating.

Logistic Regression with Tensorﬂow. As an example, let us revisit the logistic regression problem using Tensorﬂow.

>>> import numpy as np

>>> from matplotlib.pylab import subplots

>>> v = 0.9

>>> @np.vectorize ... def gen\_y(x):

... if x<5: return np.random.choice([0,1],p=[v,1-v]) ... else: return np.random.choice([0,1],p=[1-v,v]) ...

>>> xi = np.sort(np.random.rand(500)\*10)

>>> yi = gen\_y(xi)

The simplest multilayer perceptron has a single hidden layer. Given the training set { x i , y i } The input vector x i is component-wise multiplied by the weight vector, w and then fed into the nonlinear sigmoidal function. The output of the sigmoidal function is then compared to the training output, y i , corresponding to the weight vector, to form the error. The key step after error-formation is the backpropagation step. This applies the chain rule from calculus to transmit the differential error back to the weight vector.

Let’s see if we can reproduce the logistic regression solution shown in Fig.4.24 using Tensorﬂow. The ﬁrst step is to import the Tensorﬂow module,

>>> import tensorflow as tf

We need to reformat the training set slightly,

>>> yi[yi==0]=-1 # use 1/-1 mapping

Then, we create the computational graph by creating variables and placeholders for the individual terms,

>>> w = tf.Variable([0.1])

>>> b = tf.Variable([0.1])

>>> # the training set items fill these

>>> x = tf.placeholder("float", [None])

>>> y = tf.placeholder("float", [None]) 4.13 Deep Learning

349

The output of the neural network is sometimes called the activation,

>>> activation = tf.exp(w\*x + b)/(1+tf.exp(w\*x + b))

The optimization problem is to reduce the following objective function, which includes the one-dimensional regularization term w 2 ,

>>> # objective

>>> obj=tf.reduce\_sum(tf.log(1+tf.exp(-y\*(b+w\*x))))+tf.pow(w,2)

Given the objective function, we choose the GradientDescentOptimizer as the optimization algorithm with the embedded learning rate,

>>> optimizer = tf.train.GradientDescentOptimizer(0.001/5.).minimize(obj)

Now, we are just about ready to start the session. But, ﬁrst we need to initialize all the variables,

>>> init=tf.global\_variables\_initializer()

We’ll use an interactive session for convenience and then step through the optimization algorithm in the following loop,

>>> s = tf.InteractiveSession()

>>> s.run(init)

>>> for i in range(1000):

... s.run(optimizer,feed\_dict={x:xi,y:yi}) ...

The result of this is shown in Fig.4.65 which says that logistic regression and this simple single-layer perceptron both come up with the same answer.

Fig. 4.65 This shows the result from logistic regression as compared to the corresponding result from simple single-layer perceptron 350

4

Machine Learning

4.13.2 Understanding Gradient Descent

Consider a smooth function f over R n suppose we want to ﬁnd the minimum value of f (x) over this domain, as in the following:

x ∗ = arg min f (x)

x

The idea with gradient descent is to choose an initial point x (0) ∈ Rn

x (k + 1) = x (k) − α∇ f (x (k) )

where α is the step size (learning rate). The intuition here is that ∇ f is the direction of increase and so that moving in the opposite direction scaled by α moves toward a lower function value. This approach turns out to be very fast for well-conditioned, strongly convex f but in general there are practical issues.

Figure 4.66 shows the function f (x) = 2 − 3x 3 + x 4 and its ﬁrst-order Taylor series approximation at selected points along the curve for a given width parameter. That is, the Taylor approximation approximates the function at a speciﬁc point with a corresponding interval around that point for which the approximation is assumed valid. The size of this width is determined by the α step parameter. Crucially, the quality of the approximation varies along the curve. In particular, there are sections where two nearby approximations overlap given the width, as indicated by the dark shaded regions. This is key because gradient descent works by using such ﬁrst-order approximations to estimate the next step in the minimization. That is, the gradient descent algorithm never actually sees f (x), but rather only the given ﬁrst-order approximant. It judges the direction of the next iterative step by sliding down the slope of the approximant to the edge of a region (determined by α) and then using that next point for the next calculated approximant. As shown by the shaded regions, it is possible that the algorithm will overshoot the minimum because the step size (α) is too large. This can cause oscillations as shown in Fig.4.67.

Let us consider the following Python implementation of gradient descent, using Sympy.

>>> x = sm.var('x')

>>> fx = 2 - 3\*x\*\*3 + x\*\*4

>>> df = fx.diff(x) # compute derivative

>>> x0 =.1 # initial guess

>>> xlist = [(x0,fx.subs(x,x0))]

>>> alpha = 0.1 # step size

>>> for i in range(20):

... x0 = x0 - alpha\*df.subs(x,x0) ... xlist.append((x0,fx.subs(x,x0))) ... 4.13 Deep Learning

351

Fig. 4.66 The piece-wise linear approximant to f (x)

Fig. 4.67 The step size may cause oscillations

Figure 4.67 shows the sequential steps. Note that the algorithm oscillates at the end because the step size is too large. Practically speaking, it is not possible to know the optimal step size for general functions without strong assumptions on f (x).

Figure 4.68 shows how the algorithm moves along the function as well as the approximant ( f ˆ (x)) that the algorithm sees along the way. Note that initial steps are crowded around the initial point because the corresponding gradient is small there. Toward the middle, the algorithm makes a big jump because the gradient is steep, before ﬁnally oscillating toward the end. Sections of the underlying function that are relatively ﬂat can cause the algorithm to converge very slowly. Furthermore, if there are multiple local minima, then the algorithm cannot guarantee ﬁnding the global minimum. 352

4

Machine Learning

Fig. 4.68 Gradient descent produces a sequence of approximants

As we have seen, the step size is key to both performance and convergence. Indeed, a step size that is too big can cause divergence and one that is too small can take a very long time to converge.

Newton’s Method. Consider the following second-order Taylor series expansion

1 J(x) = f (x 0 ) + ∇ f (x 0 ) T (x − x 0 ) + ) T ∇ 2 f (x 0 )(x − x 0 ) (x x0 2

where H(x) := ∇ 2 f (x) is the Hessian matrix of second derivatives. The (i, j)th entry of this matrix is the following:

∂ 2 f (x)

∂x i ∂xj

We can use basic matrix calculus to ﬁnd the minimum by computing:

∇ x J(x) = ∇ f (x 0 ) + H(x)(x − x 0 ) = 0

Solving this for x gives the following:

x = x 0 − H(x) −1 ∇ f (x 0 )

Thus, after renaming some terms, the descent algorithm works by the following update equation:

x (k + 1) = x (k) − H(x (k) ) −1 ∇ f (x (k) )

There are a number of practical problems with this update equation. First, it requires computing the Hessian matrix at every step. For a signiﬁcant problem, this means managing a potentially very large matrix. For example, given 1000 dimensions the 4.13 Deep Learning

353

corresponding Hessian has 1000 × 1000 elements. Some other issues are that the Hessian may not be numerically stable enough to invert, the functional form of the partial derivatives may have to be separately approximated, and the initial guess has to be in a region where the convexity of the function matches the derived assumption. Otherwise, just based on these equations, the algorithm will converge on the local maximum and not the local minimum. Consider a slight change of the previous code to implement Newton’s method:

>>> x0 =2. # init guess is near to solution

>>> xlist = [(x0,fx.subs(x,x0))]

>>> df2 = fx.diff(x,2) # 2nd derivative

>>> for i in range(5):

... x0 = x0 - df.subs(x,x0)/df2.subs(x,x0) ... xlist.append((x0,fx.subs(x,x0))) ...

>>> xlist = np.array(xlist).astype(float)

>>> print (xlist)

[[ 2. -6. ]

[ 2.33333333 -6.4691358 ] [ 2.25555556 -6.54265522] [ 2.25002723 -6.54296874] [ 2.25 -6.54296875] [ 2.25 -6.54296875]]

Note that it took very few iterations to get to the minimum (as compared to our prior method), but if the initial guess is too far away from the actual minimum, the algorithm may not ﬁnd the local minimum at all and instead ﬁnd the local maximum. Naturally, there are many extensions to this method to account for these effects, but the main thrust of this section is to illustrate how higher order derivatives (when available) in a computationally feasible context can greatly accelerate convergence of descent algorithms.

Managing Step Size. The problem of determining a good step size (learning rate) can be approached with an exact line search. That is, along the ray that extends along x + q∇ f (x), ﬁnd

q min = arg min f (x + q∇ f (x))

q ≥ 0

In words, this means that given a direction from a point x along the direction ∇ f (x), ﬁnd the minimum for this one-dimensional problem. Thus, the minimization procedure alternates at each iteration between moving to a new x position in R n and ﬁnding a new step size by solving the one-dimensional minimization.

While this is conceptually clean, the problem is that solving the one-dimensional line search at every step means evaluating the objective function f (x) at many points 354

4

Machine Learning

along the one-dimensional slice. This can be very time consuming for an objective function that is computationally expensive to evaluate. With Newton’s method, we have seen that higher order derivatives can accelerate convergence and we can apply those ideas to the one-dimensional line search, as with the backtracking algorithm.

• Fix parameters β ∈ [ 0, 1) an α > 0.

• If f (x − α∇ f (x)) > f (x) − α ∥ ∇ f (x) ∥ 2 2 then reduce α → β α. Otherwise, do the usual gradient descent update: x (k + 1) = x (k) − α∇ f (x (k) ).

To gain some intuition about how this works, return to our second-order Taylor series expansion of the function f about x 0 ,

1 f (x 0 ) + ∇ f (x 0 ) T (x − x 0 ) + ) T ∇ 2 f (x 0 )(x − x 0 ) (x x0 2

We have already discussed the numerical issues with the Hessian term, so one approach is to simply replace that term with an n × n identity matrix I to obtain the following:

1 h α (x) = f (x 0 ) + ∇ f (x 0 ) T (x − x 0 ) + ∥2 x x0 ∥ 2α

This is our more tractable surrogate function. But what is the relationship between this surrogate and what we are actually trying to minimize? The key difference is that the curvature information that is contained in the Hessian term has now been reduced to a single 1/α factor. Intuitively, this means that local complicated curvature of f about a given point x 0 has been replaced with a uniform bowl-shaped structure, the steepness of which is determined by scaling 1/α. Given a speciﬁc α, we already know how to step directly to the minimum of h α (x); namely, using the following gradient descent update equation:

x (k + 1) = x (k) − α∇ f (x (k) )

That is the immediate solution to the surrogate problem, but it does not directly supply the next iteration for the function we really want: f . Let us suppose that our minimization of the surrogate has taken us to a new point x (k) that satisﬁes the following inequality,

f (x (k + 1) ) ≤ h α (x (k + 1) )

or, more explicitly,

1 f (x (k + 1) ) ≤ f (x (k) ) + ∇ f (x (k) ) T (x (k + 1) − x (k) ) + − x (k) ∥2 ∥ x(k + 1) 2α 4.13 Deep Learning

355

We can substitute the update equation into this and simplify as,

α f (x (k + 1) ) ≤ f (x (k) ) − α∇ f (x (k) ) T (∇ f (x (k) )) + ) ∥2 ∥ ∇ f (x(k) 2

which ultimately boils down to the following:

α f (x (k + 1) ) ≤ f (x (k) ) − ) ∥2 ∥ ∇ f (x(k) 2

(4.13.2.1)

The important observation here is that if we have not reached the minimum of f , then the last term is always positive and we have moved downward,

f (x (k + 1) ) < f (x (k) )

which is what we were after. Conversely, if the inequality in Eq.(4.13.2.1) holds for some α > 0, then we know that h α > f . This is the key observation behind the backtracking algorithm. That is, we can test a sequence of values for α until we ﬁnd one that satisﬁes Eq.(4.13.2.1). For example, we can start with some initial α and then scale it up or down until the inequality is satisﬁed which means that we have found the correct step size and then can proceed with the descent step. This is what the backtracking algorithm is doing as shown in Fig.4.69. The dotted line is the h α (x) and the gray line is f (x). The algorithm hops to the quadratic minimum of the h α (x) function which is close to the actual minimum of f (x).

The basic implementation of backtracking is shown below:

>>> x0 = 1

>>> alpha = 0.5

>>> xnew = x0 - alpha\*df.subs(x,x0)

>>> while fx.subs(x,xnew)>(fx.subs(x,x0)-(alpha/2.)\*(fx.subs(x,x0))\*\*2): ... alpha = alpha \* 0.8 ... xnew = x0 - alpha\*df.subs(x,x0) ...

>>> print (alpha,xnew)

0.32000000000000006 2.60000000000000

Stochastic Gradient Descent. A common variation on gradient descent is to alter how the weights are updated. Speciﬁcally, suppose we want to minimize an objective function of the following form:

m min ∑ f i (x) x i=1

where i indexes the i th data element for an error function. Equivalently, each summand is parameterized by a data element.

For the usual gradient descent algorithm, we would compute the incremental weights, component-wise as in the following 356

4

Machine Learning

Fig. 4.69 The approximation h α (x) (dotted line) moves the next iteration from x = 1 to the indicated point that is near the minimum of f (x) by ﬁnding an appropriate step size (α)

m x (k + 1) = x (k) − α k ∑ ∂ f i (x (k) ) i=1

by summing over all of the data. The key idea for stochastic gradient descent is to not sum over all of the data but rather to update the weights for each randomized ith data element:

x (k + 1) = x (k) − α k ∂ f i (x (k) )

A compromise between batch and this jump-every-time stochastic gradient descent is mini-batch gradient descent in which a randomized subset (σ r , | σ r | = M b ) of the data is summed over at each step as in the following:

x (k + 1) = x (k) − α k ∑ ∂ f i (x (k) ) i∈σ r

Each step update for the standard gradient descent algorithm processes m data points for each of the p dimensions, O(mp), whereas for stochastic gradient descent, we have O(p). Mini-batch gradient descent is somewhere in-between these estimates. For very large, high-dimensional data, the computational costs of gradient descent can become prohibitive thus favoring stochastic gradient descent. Outside of the computational advantages, stochastic gradient descent has other favorable attributes. For example, the noisy jumping around helps the algorithm avoid getting stalled in local minima and this helps the algorithm when the starting point is far away from the actual minimum. The obverse is that stochastic gradient descent can struggle to clinch the minimum when it is close to it. Another advantage is robustness to a minority of bad data elements. Because only random subsets of the data are actually used in the update, the few individual outlier data points (perhaps due to poor data integrity) do not necessarily contaminate every step update (Fig.4.70). 4.13 Deep Learning

357

Fig. 4.70 The approximation h α (x) (dotted line) moves the next iteration from x = 1 to the indicated point that is near the minimum of f (x) by ﬁnding an appropriate step size (α)

Momentum. The gradient descent algorithm can be considered as a particle moving along a high-dimensional landscape in search of a minimum. Using a physical analogy, we can add the concept of momentum to the particle’s motion. Consider the position of the particle (x (k) ) at any time k under a net force proportional to −∇ J. This setup induces an estimated velocity term for the particle motion proportional to η (x (k + 1) − x (k) ). That is, the particle’s velocity is estimated proportional to the difference in two successive positions. The simplest version of stochastic gradient descent update that incorporates this momentum is the following:

x (k + 1) = x (k) − α∇ f (x (k) ) + η (x (k + 1) − x (k) )

Momentum is particularly useful when gradient descent sloshes up and down a steep ravine in the error surface instead of pursuing the descending ravine directly to the local minimum. This oscillatory behavior can cause slow convergence. There are many extensions to this basic idea such as Nesterov momentum.

Advanced Stochastic Gradient Descent. Methods that aggregate histories of the stepupdatescanprovidesuperiorperformancetothebasicstochasticgradientdescent algorithm. For example, Adam (Adaptive Moment Estimator) implements an adaptive step size for each parameter. It also keeps track of an exponentially decaying mean and variance of past gradients using the exponentially weighted moving average (EWMA). This smoothing technique computes the following recursion,

y n = ax n + (1 − a)yn−1

with y 0 = x 0 as the initial condition. The 0 < a < 1 factor controls the amount of mixing between the previous moving average and the new data point at n. For example, if a = 0.9, the EWMA favors the new data x n over the prior value yn−1 (1 − a = 0.1) of the EWMA. This calculation is common in a wide variety of 358

4

Machine Learning

Fig. 4.71 Different variations of gradient descent

time series applications (i.e., signal processing, quantitative ﬁnance). The impulse response of the EWMA (x = δ n ) is (1 − a) n . You can think of this as the weighted window function that is applied to x n . As opposed to the standard moving average that considers a ﬁxed window of data to average over, this exponential window retains prior memory of the entire sequence, albeit weighted by powers of (1 − a). To see this, we can generate the response to an impulse data series using pandas,

>>> import pandas as pd

>>> x = pd.Series([1]+[0]\*20)

>>> ma =x.rolling(window=3, center=False).mean()

>>> ewma = x.ewm(1).mean() As shown by Fig.4.71, the single nonzero data point thereafter inﬂuences the EWMA whereas for the ﬁxed-width moving window average, the effect terminates after the window passes. Note that mini-batch smoothes out data at each iteration by averaging over training data and EWMA smoothes out the descent motion across iterations of the algorithm.

Advanced stochastic gradient descent algorithms are themselves an area of intense interest and development. Each method has its strengths and weaknesses pursuant to the data at hand (i.e., sparse vs. dense data) and there is no clear favorite appropriate to all circumstances. As a practical matter, some variants have parallel implementations that accelerate performance (i.e., Nui’s Hogwild update scheme).

Python Example Using Sympy. Each of these methods will make more sense with some Python. We emphasize that this implementation is strictly expository and would not be suitable for a large-scale application. Let us reconsider the classiﬁcation problem in the section on logistic regression with the target y i ∈ { 0, 1 } . The logistic regression seeks to minimize the cross-entropy:

m J( β ) = ∑ log(1 + exp(x i T β )) − y i x i T β i 4.13 Deep Learning

359

with the corresponding gradient,

m 1 β ∇ β ∑ 1 + = exp(−x i T ) ) x i −J( y i xi i β

To get started let’s create some sample data for logistic regression

>>> import numpy as np

>>> import sympy as sm

>>> npts = 100

>>> X=np.random.rand(npts,2)\*6-3 # random scatter in 2-d plane

>>> labels=np.ones(X.shape[0],dtype=np.int) # labels are 0 or 1

>>> labels[(X[:,1]<X[:,0])]=0

This provides the data in the X Numpy array and the target labels in the labels array. Next, we want to develop the objective function with Sympy,

>>> x0,x1 = sm.symbols('x:2',real=True) # data placeholders

>>> b0,b1 = sm.symbols('b:2',real=True) # parameters

>>> bias = sm.symbols('bias',real=True) # bias term

>>> y = sm.symbols('y',real=True) # label placeholders

>>> summand = sm.log(1+sm.exp(x0\*b0+x1\*b1+bias))-y\*(x0\*b0+x1\*b1+bias) ... for (i,j),y\_i in zip(X,labels)])

>>> J = sum([summand.subs({x0:i,x1:j,y:y\_i})

We can use Sympy to compute the gradient as in the following:

>>> from sympy.tensor.array import derive\_by\_array

>>> grad = derive\_by\_array(summand,(b0,b1,bias))

Using the sm.latex function renders grad as the following:

x 0 e b 0 x 0 +b 1 x 1 +bias x 1 e b 0 x 0 +b 1 x 1 +bias e b 0 x 0 +b 1 x 1 +bias −x0 y + −x 1 y + −y + [ e b 0 x 0 +b 1 x 1 +bias + 1 e b 0 x 0 +b 1 x 1 +bias + 1 e b 0 x 0 +b 1 x 1 +bias + 1 ]

which matches our previous computation of the gradient. For standard gradient descent, the gradient is computed by summing over all of the data,

>>> grads=np.array([grad.subs({x0:i,x1:j,y:y\_i}) ... for (i,j),y\_i in zip(X,labels)]).sum(axis=0)

Now, to implement gradient descent, we set up the following loop:

>>> # convert expression into function

>>> Jf = sm.lambdify((b0,b1,bias),J)

>>> gradsf = sm.lambdify((b0,b1,bias),grads)

>>> niter = 200

>>> winit = np.random.randn(3)\*20

>>> alpha = 0.1 # learning rate (step-size) 360

4

Machine Learning

>>> WK = winit # initialize

>>> Jout=[] # container for output

>>> for i in range(niter):

... WK = WK - alpha \* np.array(gradsf(\*WK)) ... Jout.append(Jf(\*WK)) ...

For stochastic gradient descent, the above code changes to the following:

>>> import random

>>> sgdWK = winit # initialize

>>> Jout=[] # container for output

>>> # don't sum along all data as before

>>> grads=np.array([grad.subs({x0:i,x1:j,y:y\_i})

... for (i,j),y\_i in zip(X,labels)])

>>> for i in range(niter):

... gradsf = sm.lambdify((b0,b1,bias),random.choice(grads)) ... sgdWK = sgdWK - alpha \* np.array(gradsf(\*sgdWK)) ... Jout.append(Jf(\*sgdWK)) ...

The main difference here is that the gradient calculation no longer sums across all of the input data (i.e., grads list) and is instead randomly chosen by the random.choice function the above body of the loop. The extension to batch gradient descent from this code just requires averaging over a sub-selection of the data for the gradients in the batch variable.

>>> mbsgdWK = winit # initialize

>>> Jout=[] # container for output

>>> mb = 10 # number of elements in batch

>>> for i in range(niter):

... batch = np.vstack([random.choice(grads) ... for i in range(mb)]).mean(axis=0) ... gradsf = sm.lambdify((b0,b1,bias),batch) ... mbsgdWK = mbsgdWK-alpha\*np.array(gradsf(\*mbsgdWK)) ... Jout.append(Jf(\*mbsgdWK)) ...

It is straightforward to incorporate momentum into this loop using a Python deque, as in the following:

>>> from collections import deque

>>> momentum = deque([winit,winit],2)

>>> mbsgdWK = winit # initialize

>>> Jout=[] # container for output

>>> mb = 10 # number of elements in batch

>>> for i in range(niter):

... batch=np.vstack([random.choice(grads) ...

for

i

in

range(mb)]).mean(axis=0) 4.13 Deep Learning

361

... ... ... ...

gradsf=sm.lambdify((b0,b1,bias),batch) mbsgdWK=mbsgdWK-alpha\*np.array(gradsf(\*mbsgdWK))+0.5\*(momentum[1]-momentum[0]) Jout.append(Jf(\*mbsgdWK))

Figure 4.71 shows the three variants of the gradient descent algorithm. Notice that the stochastic gradient descent algorithm is the most erratic, as it is characterized by taking a new direction for every randomly selected data element. Mini-batch gradient descent smoothes these out by averaging across multiple data elements. The momentum variant is somewhere in-between the to as the effect of the momentum term is not pronounced in this example.

Python Example Using Theano. The code shown makes each step of the gradient descent algorithms explicit using Sympy, but this implementation is far too slow. The theano module provides thoughtful and powerful high-level abstractions for algorithm implementation that relies upon underlying C/C++ and GPU execution models.Thismeansthatcalculationsthatareprototypedwith theanocanbeexecuted downstream outside of the Python interpreter which makes them much faster. The downside of this approach is that calculations can become much harder to debug because of the multiple levels of abstraction. Nonetheless, theano is a powerful tool for algorithm development and execution.

To get started we need some basics from theano.

>>> import theano

>>> import theano.tensor as T

>>> from theano import function, shared

the next step is to deﬁne variables, which are essentially placeholders for values that will be computed downstream later. The next block deﬁnes two named variables as a double-sized ﬂoat matrix and vector. Note that we did not have to specify the dimensions of each at this point.

>>> x = T.dmatrix("x") # double matrix

>>> y = T.dvector("y") # double vector

The parameters of our implementation of gradient descent come next, as the following:

>>> w = shared(np.random.randn(2), name="w") # parameters to fit

>>> b = shared(0.0, name="b") # bias term

variables that are shared are ones whose values can be set separately via other computations or directly via the set\_value() method. These values can also be retrieved using the get\_value() method. Now, we need to deﬁne the probability of obtaining a 1 from the given data as p. The cross-entropy function and the T.dot function are already present (along with a wide range of other related functions) in theano. The conformability of the constituent arguments is the responsibility of the user. 362

4

Machine Learning

>>> p=1/(1+T.exp(-T.dot(x,w)-b)) # probability of 1

>>> error = T.nnet.binary\_crossentropy(p,y)

>>> loss = error.mean()

>>> gw, gb = T.grad(loss, [w, b])

The error variable is TensorVariable type which has many built-in methods such as mean. The so-derived loss function is therefore also a TensorVariable. The last T.grad line is the best part of Theano because it can compute these gradients automatically.

>>> train = function(inputs=[x,y],

... ... ...

outputs=[error], updates=((w, w - alpha \* gw), (b, b - alpha \* gb)))

The last step is to set up training by deﬁning the training function in theano. The user will supply the previously deﬁned and named input variables (x and y) and theano will return the previously deﬁned error variable. Recall that the w and b variables were deﬁned as shared variables. This means that the function train can update their values between calls using the update formula speciﬁed in the updates keyword variable. In this case, the update is just plain gradient descent with the previously deﬁned alpha step size variable.

We can execute the training plan using the train function in the following loop:

>>> training\_steps=1000

>>> for i in range(training\_steps): ... error = train(X, labels) ...

The train(X,labels) call is where the X and labels arrays we deﬁned earlier replace the placeholder variables. The update step refreshes all of the shared variables at each iterative step. At the end of the iteration, the so-computed parameters are in the w and b variables with values available via get\_value(). The implementation for stochastic gradient descent requires just a little modiﬁcation to this loop, as in the following:

>>> for i in range(training\_steps): ...

idx

=

error

np.random.randint(0,X.shape[0])

...

...

=

train([X[idx,:]],

[labels[idx]])

where the idx variable selects a random data element from the set and uses that for the update step at every iteration. Likewise, batch stochastic gradient descent follows with the following modiﬁcation,

>>> batch\_size = 50

>>> indices = np.arange(X.shape[0])

>>> for i in range(training\_steps): ...

idx

=

np.random.permutation(indices)[:batch\_size] 4.13 Deep Learning

363

... ...

error

=

train(X[idx,:],

labels[idx])

>>> print (w.get\_value()) [-4.84350587 5.013989 ]

>>> print (b.get\_value()) # bias term

0.5736726430208784

Here, we set up an indices variable that is used for randomly selecting subsets in the idx variable that are passed to the train function. All of these implementations parallel the corresponding previous implementations in Sympy, but these are many orders of magnitude faster due to theano.

4.13.3 Image Processing Using Convolutional Neural Networks

In this section, we develop the convolutional neural network (CNN) which is the fundamental deep learning image processing application. We deconstruct every layer of this network to develop insight into the purpose of the individual operations. CNNs take image as inputs and images can be represented as Numpy arrays, which makes them fast and easy to use with any of the scientiﬁc Python tools. The individual entries of the Numpy array are the pixels and the row/column dimensions are the height/width of the image, respectively. The array values are between 0 through 255 and correspond to the intensity of the pixel at that location. Three-dimensional images have a third depth-dimension as the color channel (e.g., red, green, blue). Two-dimensional image arrays are grayscale.

Programming Tip

Matplotlib makes it easy to draw images using the underlying Numpy arrays. For instance, we can draw Fig.4.72 using the following MNIST image from sklearn.datasets, which represents grayscale hand-drawn digits (the number zero in this case).

>>> from matplotlib.pylab import subplots, cm

>>> from sklearn import datasets

>>> mnist = datasets.load\_digits()

>>> fig, ax = subplots()

>>> ax.imshow(mnist.images[0],

... interpolation='nearest', ... cmap=cm.gray) <matplotlib.image.AxesImage object at 0x7f98d4212f98>

Thecmapkeywordargumentspeciﬁesthecolormapasgray.Theinterpolation keyword means that the resulting image from imshow does not try to visually smooth out the data, which can be confusing when working at the pixel level. The other hand-drawn digits are shown below in Fig.4.73. 364

4

Machine Learning

Fig. 4.72 Image of a hand-drawn number zero from the MNIST dataset

Fig. 4.73 Samples of the other hand-drawn digits from MNIST

Convolution. Convolution is an intensive calculation and it is the core of convolutional neural networks. The purpose of convolution is to create alternative representations of the input image that emphasize or deemphasize certain features represented by the kernel. The convolution operation consists of a kernel and an input matrix. The convolution operation is a way of aligning and comparing image data with the corresponding data in an image kernel. You can think of an image kernel as a template for a canonical feature that the convolution operation will uncover. To keep it simple suppose we have the following 3x3 kernel matrix,

>>> import numpy as np

>>> kern = np.eye(3,dtype=np.int)

>>> kern 4.13 Deep Learning

365

array([[1, 0, 0],

[0, 1, 0], [0, 0, 1]])

Using this kernel, we want to ﬁnd anything in an input image that looks like a diagonal line. Let’s suppose we have the following input Numpy image

>>> tmp = np.hstack([kern,kern\*0])

>>> x = np.vstack([tmp,tmp])

>>> x array([[1, 0, 0, 0, 0, 0], [0, 1, 0, 0, 0, 0], [0, 0, 1, 0, 0, 0], [1, 0, 0, 0, 0, 0], [0, 1, 0, 0, 0, 0], [0, 0, 1, 0, 0, 0]])

Note that this image is just the kernel stacked into a larger Numpy array. We want to see if the convolution can pull out the kernel that is embedded in the image. Of course, in a real application we would not know whether or not the kernel is present in the image, but this example helps us understand the convolution operation step-by-step. There is a convolution function available in the scipy module.

>>> from scipy.ndimage.filters import convolve

>>> res = convolve(x,kern,mode='constant',cval=0)

>>> res

array([[2, 0, 0, 0, 0, 0],

[0, 3, 0, 0, 0, 0], [0, 0, 2, 0, 0, 0], [2, 0, 0, 1, 0, 0], [0, 3, 0, 0, 0, 0], [0, 0, 2, 0, 0, 0]])

Each step of the convolution operation is represented in Fig.4.74. The kern matrix (light blue square) is overlaid upon the x matrix and the element-wise product is computed and summed. Thus, the 0,0 array output corresponds to this operation applied to the top left 3x3 slice of the input, which results in 3. The convolution operation is sensitive to boundary conditions. For this example, we have chosen mode=constant and cval=0 which means that the input image is bordered by zeros when the kernel sweeps outside of the input image boundary. This is the simplest option for managing the edge conditions and scipy.ndimage.filters.convolve provides other practical alternatives. It is also common to normalize the output of the convolution operation by dividing by the number of pixels in the kernel (i.e., 3 in this example). Another way to think about the convolution operation is as a matched ﬁlter that peaks when it ﬁnds a compatible sub-feature. The ﬁnal output of the convolution operation is shown in Fig.4.75. The values of the individual pixels are shown in color. Notice where the maximum values of the output image are located on the diagonals. 366

4

Machine Learning

Fig. 4.74 The convolution process that produces the res array. As shown in the sequence, the light blue kern array is slid around, overlaid, multiplied, and summed upon the x array to generate the values of shown in the title. The output of the convolution is shown in Fig.4.75

Fig. 4.75 The res array output of the convolution is shown in Fig.4.74. The values (in red) shown are the individual outputs of the convolution operation. The grayscale indicates the relative magnitude of the shown values (darker is greater) 4.13 Deep Learning

367

Fig. 4.76 The input array is a forward-slash diagonal. This sequence shows the step-by-step convolution operation. The output of this convolution is shown in Fig.4.77

However, the convolution operation is not a perfect detector and results in nonzero values for other cases. For example, suppose the input image is a forward-slash diagonal line. The step-by-step convolution with the kernel is shown in Fig.4.76 with corresponding output in Fig.4.77 that looks nothing like the kernel or the input image.

We can use multiple kernels to explore an input image. For example, suppose we have the input image shown on the left in Fig.4.78. The two kernels are shown in the upper row, with corresponding outputs on the bottom row. Each kernel is able to emphasize its particular feature but extraneous features appear in both outputs. We can have as many outputs as we have kernels but because each output image is as large as the input image, we need a way to reduce the size of this data. 368

4

Machine Learning

Fig. 4.77 The output of the convolution operation shown in Fig.4.76. Note that the output has nonzero elements where there is no match between the input image and the kernel

Fig. 4.78 Given two kernels (upper row) and the input image on the left, the output images are shown on the bottom row. Note that each kernel is able to emphasize its feature on the input composite image but other extraneous features appear in the outputs

Maximum Pooling. To reduce the size of the output images, we can apply maximum pooling to replace a tiled subset of the image with the maximum pixel value in that particular subset. The following Python code illustrates maximum pooling,

>>> def max\_pool(res,width=2,height=2):

... ... ... ... ...

m,n = res.shape xi = [slice(i,i+width) for i in range(0,m,width)] yi = [slice(i,i+height) for i in range(0,n,height)] out = np.zeros((len(xi),len(yi)),dtype=res.dtype) for ni,i in enumerate(xi): 4.13 Deep Learning

369

Fig. 4.79 The max\_pool function reduces the size of the output images (left column) to the images on the right column. Note that the pool size is 2x2 so that the resulting pooled images are half the size of the original images in each dimension

... ... ... ...

for nj,j in enumerate(yi):

out[ni,nj]= res[i,j].max() return out

Programming Tip

The slice object provides programmatic array slicing. For example, x[0,3]=x [slice(0,3)]. This means you can separate the slice from the array, which makes it easier to manage.

Pooling reduces the dimensionality of the output of the convolution and makes stacking convolutions together computationally feasible. Figure 4.79 shows the output of the max\_pool function on the indicated input images.

Rectiﬁed Linear Activation. Rectiﬁed Linear Activation Units (ReLUs) are neural network units that implement the following activation function,

x if x > 0 r(x) = { 0 otherwise 370

4

Machine Learning

Fig. 4.80 The training dataset for our convolutional neural network. The forward-slash images are labeled category 0 and the backward slash images are category 1

To use this activation properly, the kernels in the convolutional layer must be scaled to the { −1, 1 } range. We can implement our own rectiﬁed linear activation function using the following code,

>>> def relu(x):

... 'rectified linear activation function' ... out = np.zeros(x.shape,dtype=x.dtype) ...

idx

=

x>=0

... out[idx]=x[idx] ... return out ...

Now that we understand the basic building blocks, let us investigate how the operations ﬁt together. To create some training image data, we use the following function to create some random backward and forward slashed images as shown in Fig.4.80. As before, we have the scaled kernels shown in Fig.4.81. We are going to 4.13 Deep Learning

371

Fig. 4.81 The two scaled feature kernels for the convolutional neural network

apply the convolution, max-pooling, and rectiﬁed linear activation function sequence step-by-step and observe the outputs at each step.

>>> def gen\_rand\_slash(m=6,n=6,direction='back'):

... '''generate random forward/backslash images.

... Must have at least two pixels''' ... assert direction in ('back','forward') ...

assert

n>=2

and

m>=2

... import numpy as np ... import random ... out = -np.ones((m,n),dtype=float) ... i = random.randint(2,min(m,n)) ... j = random.randint(-i,max(m,n)-1) ... t = np.diag([1,]\*i,j) ... if direction == 'forward':

... t = np.flipud(t) ... try:

... assert t.sum().sum()>=2 ... out[np.where(t)]=1 ... return out ... except:

... return gen\_rand\_slash(m=m,n=n,direction=direction) ...

>>> # create slash-images training data with classification id 1 or 0

>>> training=[(gen\_rand\_slash(),1) for i in range(10)] + \

... [(gen\_rand\_slash(direction='forward'),0) for i in range(10)]

Figure 4.82 shows the output of convolving the training data in Fig.4.80 with kern1, as shown on the left panel of Fig.4.81. Note that the following code deﬁnes each of these kernels,

>>> kern1 = (np.eye(3,dtype=np.int)\*2-1)/9. # scale

>>> kern2 = np.flipud(kern1) 372

4

Machine Learning

Fig. 4.82 The output of convolving the training data in Fig.4.80 with kern1, as shown on the left panel of Fig.4.81

The next operation is the activation function for the rectiﬁed linear unit with output shown in Fig.4.83. Note that all of the negative terms have been replaced by zeros. The next step is the maximum pooling operation as shown in Fig.4.84. Notice that the number of total pixels in the training data has reduced from thirty-six per image to nine per image. With these processed images, we have the inputs we need for the ﬁnal classiﬁcation step.

Convolutional Neural Network Using Keras. Now that we have experimented with the individual operations using our own Python code, we can construct the convolutional neural network using Keras. In particular, we use the Keras functional interface to deﬁne this neural network because that makes it easy to unpack the operations at the individual layers.

>>> from keras import metrics

>>> from keras.models import Model

>>> from keras.layers.core import Dense, Activation, Flatten

>>> from keras.layers import Input

>>> from keras.layers.convolutional import Conv2D 4.13 Deep Learning

373

Fig. 4.83 The output of the rectiﬁed linear unit activation function with the input shown in Fig.4.82

>>> from keras.layers.pooling import MaxPooling2D

>>> from keras.optimizers import SGD

>>> from keras import backend as K

>>> from keras.utils import to\_categorical

Note that the names of the modules are consistent with their operations. We also need to tell Keras how to manage the input images,

>>> K.set\_image\_data\_format('channels\_first') # image data format

>>> inputs = Input(shape=(1,6,6)) # input data shape

Now we can build the individual convolutional layers. Note the speciﬁcation of the activations at each layer and placement of the inputs.

>>> clayer = Conv2D(2,(3,3),padding='same',

... ... ...

input\_shape=(1,6,6),name='conv', use\_bias=False, trainable=False)(inputs) 374

4

Machine Learning

Fig. 4.84 The output of maximum pooling operation with the input shown in Fig.4.83 for ﬁxed image kernel kern1

>>> relu\_layer= Activation('relu')(clayer)

>>> maxpooling = MaxPooling2D(pool\_size=(2,2),

... name='maxpool')(relu\_layer)

>>> flatten = Flatten()(maxpooling)

>>> softmax\_layer = Dense(2,

... activation='softmax', ... name='softmax')(flatten)

>>> model = Model(inputs=inputs, outputs=softmax\_layer)

>>> # inject fixed kernels into convolutional layer

>>> fixed\_kernels = [np.dstack([kern1,kern2]).reshape(3,3,1,2)]

>>> model.layers[1].set\_weights(fixed\_kernels)

Observe that the functional interface means that each layer is explicitly a function of the previous one. Note that trainable=False for the convolutional layer because we want to inject our ﬁxed kernels into it at the end. The flatten layer reshapes the data so that the entire processed image at the point is fed into the softmax\_layer, whose output is proportional to the probability that the image belongs to either class. The set\_weights() function is where we inject our ﬁxed kernels. These 4.13 Deep Learning

375

are not going to be updated by the optimization algorithm because of the prior trainable=False option. With the topology of the neural network deﬁned, we now have to choose the optimization algorithm and pack all of this conﬁguration into the model with the compile step.

>>> lr = 0.01 # learning rate

>>> sgd = SGD(lr=lr, decay=1e-6, momentum=0.9, nesterov=True)

>>> model.compile(loss='categorical\_crossentropy',

... ... ...

optimizer=sgd, metrics=['accuracy', metrics.categorical\_crossentropy])

The metrics speciﬁcation means that we want to training process to keep track of thosenameditems.Next,wegeneratesometrainingdatausingour gen\_rand\_slash function with the associated class of each image (1 or 0). Most of this code is just shaping the tensors for Keras. The ﬁnal model.fit() step is where the internal weights of the neural network are adjusted according to the given inputs.

>>> # generate some training data

>>> ntrain = len(training)

>>> t=np.dstack([training[i][0].T

... for i in range(ntrain)]).T.reshape(ntrain,1,6,6)

>>> y\_binary=to\_categorical(np.hstack([np.ones(ntrain//2), ... np.zeros(ntrain//2)]))

>>> # fit the configured model

>>> h=model.fit(t,y\_binary,epochs=500,verbose=0)

With that completed, we can investigate the functional mapping of each layer with K.function. The following creates a mapping between the input layer and the convolutional layer.

>>> convFunction = K.function([inputs],[clayer])

Now, we can feed the training data into this function as see the output of just the convolutional layer, which is shown.

We can do this again for the pooling layer by creating another Keras function,

>>> maxPoolingFunction = K.function([inputs],[maxpooling])

whose output is shown in Fig.4.86. We can examine the ﬁnal output of this network using the predict function (Fig.4.85),

>>> fixed\_kernels = model.predict(t)

>>> fixed\_kernels

array([[0.0960771 , 0.9039229 ], [0.12564187, 0.8743582 ], [0.14237107, 0.857629 ], [0.4294672 , 0.57053274], [0.13607137, 0.8639286 ], 376

4

Machine Learning

Fig. 4.85 Compare this to Fig.4.82. This shows our hand-tooled convolution is the same as that implemented by Keras

[0.7519819 , 0.24801806], [0.16871268, 0.83128726], [0.0960771 , 0.9039229 ], [0.4294672 , 0.57053274], [0.3497647 , 0.65023535], [0.8890644 , 0.11093564], [0.7882034 , 0.21179655], [0.6911642 , 0.30883583], [0.7882034 , 0.21179655], [0.5335865 , 0.46641356], [0.6458056 , 0.35419443], [0.8880452 , 0.11195483], [0.7702401 , 0.22975995], [0.7702401 , 0.2297599 ],

[0.6458056 , 0.35419443]], dtype=float32)

and we can see the weights given to each of the classes. Taking the maximum of these across the columns gives the following: 4.13 Deep Learning

377

Fig. 4.86 Output of max-pooling layer for ﬁxed kernel kern1. Compare this to Fig.4.84. This shows our hand-tooled implementation is equivalent to that by Keras

>>> np.argmax(fixed\_kernels,axis=1) array([1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

which means that our convolutional neural network with the ﬁxed kernels did well predicting the classes of each of our input images. Recall that our model conﬁguration prevented our ﬁxed kernels from updating in the training process. Thus, the main work of model training was changing the weights of the ﬁnal output layer. We can re-do this exercise by removing this constraint and see how the network performs if it is able to adaptively re-weight the kernel terms as part of training by changing the trainable keyword argument and then re-build and train the model, as shown next.

>>> clayer = Conv2D(2,(3,3),padding='same',

... input\_shape=(1,6,6),name='conv', ... use\_bias=False)(inputs)

>>> relu\_layer= Activation('relu')(clayer)

>>> maxpooling = MaxPooling2D(pool\_size=(2,2),

... name='maxpool')(relu\_layer)

>>> flatten = Flatten()(maxpooling) 378

4

Machine Learning

>>> softmax\_layer = Dense(2,

... activation='softmax', ... name='softmax')(flatten)

>>> model = Model(inputs=inputs, outputs=softmax\_layer)

>>> model.compile(loss='categorical\_crossentropy',

... optimizer=sgd)

>>> h=model.fit(t,y\_binary,epochs=500,verbose=0)

>>> new\_kernels = model.predict(t)

>>> new\_kernels

array([[1.4370615e-03, 9.9856299e-01], [3.6707439e-03, 9.9632925e-01], [1.0132928e-04, 9.9989867e-01], [4.6108435e-03, 9.9538910e-01], [2.5441888e-05, 9.9997461e-01], [7.4225911e-03, 9.9257737e-01], [1.3943247e-03, 9.9860567e-01], [1.4370615e-03, 9.9856299e-01], [4.6108435e-03, 9.9538910e-01], [3.4720991e-03, 9.9652785e-01], [9.9974054e-01, 2.5950689e-04], [9.9987161e-01, 1.2833292e-04], [9.9983239e-01, 1.6753815e-04], [9.9987161e-01, 1.2833292e-04], [9.8536682e-01, 1.4633193e-02], [9.9561429e-01, 4.3856688e-03], [9.9778903e-01, 2.2109088e-03], [9.9855381e-01, 1.4462060e-03], [9.9855381e-01, 1.4462066e-03],

[9.9561429e-01, 4.3856665e-03]], dtype=float32)

with corresponding max output,

>>> np.argmax(new\_kernels,axis=1) array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

The newly updated kernels are shown in Fig.4.87. Note how different these are from the original ﬁxed kernels. We can see the change in the respective predictions in Fig.4.88.Thus,thebeneﬁtofupdatingthekernelsinthetrainingprocessistoimprove the accuracy overall, but at the cost of interpretability of the kernels themselves. Nonetheless, it is seldom the case that the kernels are known ahead of time, as in our artiﬁcial example here, so in practice, there may be nothing to really interpret anyway. Nonetheless, for other problems where there is a target feature in the data for which good a priori exemplars exist that could serve a kernels, then priming these kernels early in training may help to tune into those target features, especially if they are rare in the training data. References

379

Fig. 4.87 Kernels updated during the training process. Compare to Fig.4.81

Fig. 4.88 Recall that the second half of the training set was classiﬁed as category 1. The updated kernels provide a wider margin for classiﬁcation than our ﬁxed kernels, even though the ultimate performance is very similar between them

References

1. L. Wasserman, All of Statistics: A Concise Course in Statistical Inference (Springer, Berlin, 2004)

2. V. Vapnik, The Nature of Statistical Learning Theory. Information Science and Statistics (Springer, Berlin, 2000)

3. R.E. Schapire, Y. Freund, Boosting Foundations and Algorithms. Adaptive Computation and Machine Learning (MIT Press, Cambridge, 2012)

4. C. Bauckhage, Numpy/Scipy recipes for data science: Kernel least squares optimization (1) (2015). researchgate.net

5. W. Richert, Building Machine Learning Systems with Python (Packt Publishing Ltd., Birmingham, 2013)

6. E. Alpaydin, Introduction to Machine Learning (Wiley Press, New York, 2014)

7. H. Cuesta, Practical Data Analysis (Packt Publishing Ltd., Birmingham, 2013)

8. A.J. Izenman, Modern Multivariate Statistical Techniques, vol. 1 (Springer, Berlin, 2008)

9. A. Hyvärinen, J. Karhunen, E. Oja, Independent Component Analysis, vol. 46 (Wiley, New York