Part 1 Classification

1 Machine learning basics

1.1 What is machine learning?

data → information

Machine learning uses statistics.

1.2 Key terminology

1. Expert system:

- Feature / attribute
- Instance

2. Training set:

- Training example
  - Feature → individual measurements
  - Target variable → *known*
    - Nominal value in classification (also called *class*)
    - Continuous value in regression

3. Test set:

- Test example
  - Feature
  - Target variable → *not given*

4. Knowledge representation

1.3 Key tasks of machine learning

- **Supervised learning**
  - **Classification:** to predict what class an instance of data should fall into
    - k-Nearest Neighbors
    - Naive Bayes
    - Support vector machines
    - Decision trees
  - **Regression:** the prediction of a numeric value

        - Linear
        - Locally weighted linear
        - Ridge
        - Lasso
- **Unsupervised learning**
    - **Clustering:** to group similar items together
        - k-Means
        - DBSCAN
    - **Density estimation:** to find statistical values that describe the data
        - Expectation maximization
        - Parzen window
    - to reduce the number of features

1.4 How to choose the right algorithm

1. Consider the goal

- Tying to predict or forcast a target value?
    - Yes → supervised learning
        - Target value is discrete → classification
        - Target value is continuous → regression
    - No → unsupervised learning
        - Trying to fit the data into some discrete groups → clustering
        - Having some numerical estimate of how strong the fit is into rach group → density estimation

2. Consider the data

- Investigating the features about data
    - Are the feature nominal or continuous?
    - Are there missing values in the features?
    - ...

3. Try different algorithms and see how they perform

1.5 Steps in developing a machine learning application

1. Collect data
2. Prepare the input data

   Make sure the data is in a useable format (in this book a Python list)

3. Analyze the input data (skippable)
4. Train the algorithm (core)

Unsupervised learning does not have this step.

Where the machine learning takes place

5. Test the algorithm (core)
    - Supervised learning → using known values
    - Unsupervised learning → using some other metrics

    If the algorithm is not satisfying, go back to step 4 or even step 1.
6. Use it

1.6 Why Python?

- Python has clear syntax.
- Python makes text manipulation extremely easy.

1.7 Getting started with the NumPy library

See Jupyter.

1.8 Summary

---

2 Classifying with k-Nearest Neighbors

2.1 Classifying with distance measurements

- **Pros:** High accuracy, insensitive to outliers, no assumptions about data
- **Cons:** Computationally expensive, requires a lot of memory
- **Works with:** Numeric values, nominal values

Fundamentals of kNN:

1. Compare a new piece of data without label to the existing data with labels, and then take *k* most similar pieces (the nearest neighbors) of data from the known dataset, looking at their lables. (*k* is an integer usually smaller than 20.)
2. Take a majority vote from the *k* most similar pieces of data, and the majority is the class assigned to new data.

General approach to kNN:

1. **Collect:** Any method.
2. **Prepare:** Numeric values are needed for a distance calculation. A structured data is best.
3. **Analyze:** Any method.
4. **Train:** Does not apply to the kNN method.
5. **Test:** Calculate the error rate.
6. **Use:** This application needs to get some input data and output structured numeric values. Next,

the application runs the kNN algorithm on this input data and determines which class the input data should belong to. The application then takes some action on the calculated class.

2.1.1 Prepare: importing data with Python

See Jupyter.

2.1.2 Putting the kNN classification algorithm into action

Goal: To use the kNN algorithm to classify one piece of data called *inX*.

Pseudocode for `classify0()`:

```
For every point in our dataset:
    calculate the distance between inX and the current point
    sort the distances in increasing order
    take k items with lowest distances to inX
    find the majority class among these items
    return the majority classas our prediction for the class of inX
```

Listing 2.1 **k-Nearest Neighbors algorithm: `classify0()`**

```
def classify0(inX, dataSet, labels, k):
    dataSetSize = dataSet.shape[0]
    diffMat = tile(inX, (dataSetSize, 1)) - dataSet
    sqDiffMat = diffMat ** 2
    sqDistances = sqDiffMat.sum(axis = 1)
    distances = sqDistances ** 0.5
    sortedDistIndicies = distances.argsort()   # Distance calculation
    classCount = {}
    for i in range(k):
        voteIlabel = labels[sortedDistIndicies[i]]
        classCount[voteIlabel] = classCount.get(voteIlabel, 0) + 1   # Voting
with lowest k distances
    sortedClassCount = sorted(classCount.items(), key = operator.itemgetter(1),
reverse = True)   # Sort dictionary
    return sortedClassCount[0][0]
```

- **Inputs:**
    - *inX:* the input vector to classify
    - *dataSet:* the full matrix of training examples
    - *labels:* a vector of labels → should have as many elements as the number of rows in *dataSet*
    - *k:* the number of nearest neighbors to use in the voting → a positive integer
- `dataSetSize = dataSet.shape[0]`

  `numpy.shape(a)` → Return the shape of an array in a tuple.

  Get the rownumber of *dataSet.*

- `diffMat = tile(inX, (dataSetSize, 1)) - dataSet`

  `numpy.tile(A, reps)` → Construct an array by repeating `A` the number of times given by `reps`.

  `reps` : The number of repetitions of `A` along each axis.

- `sortedDistIndicies = distances.argsort()`

  `numpy.argsort(a, axis = -1, kind = None, order = None)` → Returns the indices that would sort an array.

  `axis` : Axis along which to sort. The default is -1 (the last axis). If `None`, the flattened array is used.

  Get the sequence of indicies each pointing at the distance between *inX* and one of the examples from smallest to largest.

- `voteIlabel = labels[sortedDistIndicies[i]]`

  Get the label of the `(i+1)` th nearest neighbor, stored as a key.

- `classCount[voteIlabel] = classCount.get(voteIlabel, 0) + 1`

  `dict.get(key[, default])` → Return the value for `key` if `key` is in the dictionary, else `default`. If `default` is not given, it defaults to `None`.

  If the label has not been stored in `classCount`, then create a key named as this label and associate it with value "0+1", that is the first vote. If the key exists, add 1 vote.

- `sortedClassCount = sorted(classCount.items(), key = operator.itemgetter(1), reverse = True)`

  `sorted(iterable, *, key = None, reverse = False)` → Return a new sorted list from the items in `iterable`.

  `key` specifies a function of one argument that is used to extract a comparison key from each element in `iterable` (for example, `key = str.lower`). The default value is `None` (compare the elements directly).

  `reverse` is a boolean value. If set to `True`, then the list elements are sorted as if each comparison were reversed.

  `dict.items()` → Return a new view of the dictionary's items (`(key, value)` pairs).

  `operator.itemgetter()` → Return a callable object that fetches `item` from its operand using the operand's `__getitem__()` method. If multiple items are specified, returns a tuple of lookup values.

  Sort the *(label, votes)* tuple sequentially from the one with highest votes.

2.1.3 How to test a classifier

1. Feed the classifier with known data and get a result.

2. Calculate the error rate by adding up the number of wrong results and dividing it by the total number of tests.

2.2 Example: improving matches from a dating site with kNN

Example: using kNN on results from a dating site

1. **Collect:** Text file provided.
2. **Prepare:** Parse a text file in Python.
3. **Analyze:** Use Matplotlib to make 2D plots of our data.
4. **Train:** Doesn't apply to the kNN algorithm.
5. **Test:** Write a function to use some portion of the data Hellen gave us as test examples. The test examples are classified against the non-test examples. If the predicted class doesn't match the real class, we'll count that as an error.
6. **Use:** Build a simple command-line program that Hellen can use to predict whether she'll like somone based on a few inputs.

2.2.1 Prepare: parsing data from a text file

Dataset: datingTestSet.txt

Features:

- Number of frequent flyer miles earned per year
- Percentage of time spent playing video games
- Liters of ice cream consumed per week

List 2.2 **Text record to NumPy parsing code: `file2matrix`**

```python
def file2matrix(filename):
    fr = open(filename)
    numberOfLines = len(fr.readlines())   # Get number of lines in files
    returnMat = zeros((numberOfLines, 3))   # Create NumPy matrix to return
    classLabelVector = []
    fr = open(filename)
    index = 0
    for line in fr.readlines():
        line = line.strip()
        listFromLine = line.split('\t')   # Parse a line to a list
        returnMat[index,:] = listFromLine[0:3]
        classLabelVector.append(int(listFromLine[-1]))
        index += 1
    return returnMat, classLabelVector
```

- `line = line.strip()`

`str.strip([chars])` → Return a copy of the string with the leading and trailing characters removed. The `chars` argument is a string specifying the set of characters to be removed. If omitted or `None`, the `chars` argument defaults to removing whitespace.

2.2.2 Analyze: creating scatter plots with Matplotlib

See Jupyter.

2.2.3 Prepare: normalizing numeric values

When dealing with values that lie in different ranges, it's common to normalize them. Common ranges to normalize them to are 0 to 1 or -1 to 1.

The formula going to be applied: `newValue = (oldValue - min) / (max - min)`

List 2.3 **Data-normalizing code: `autoNorm()`**

```
def autoNorm(dataSet):
    minVals = dataSet.min(0)
    maxVals = dataSet.max(0)
    ranges = maxVals - minVals
    normDataSet = zeros(shape(dataSet))
    m = dataSet.shape[0]
    normDataSet = dataSet - tile(minVals, (m,1))
    normDataSet = normDataSet / tile(ranges, (m,1))   # Element-wise division
    return normDataSet, ranges, minVals
```

- **Function:** to automatically normalize the data to values between 0 and 1.
- `minVals = dataSet.min(0)`

  The *0* in `dataSet.min(0)` allows you to take the minimums from the columns, not the rows.
- `normDataSet = dataSet - tile(minVals, (m,1))`

  The NumPy `tile()` function is used to create a matrix the same size as the input matrix (from *1*3* to *1000*3*) and fill it up with many copies or tiles.
- `normDataSet = normDataSet / tile(ranges, (m,1))`

  The `/` operater can be used for matrix division.

  In NumPy, use `linalg.solve(matA, matB)` for matrix division.

2.2.4 Test: testing the classifier as a whole program

Testing method: 90% to train and 10% to test

$$Error\ rate = \frac{the\ number\ of\ misclassified\ pieces\ of\ data}{the\ total\ number\ of\ data\ points\ tested}$$

List 2.4 **Classifier testing code for dating site: `datingClassTest()`**

See Jupyter.

2.2.5 Use: putting together a useful system

List 2.5 **Dating site predictor function: `classifyPerson()`**

See Jupyter.

2.3 Example: a handwriting recognition system

Example: using kNN on a handwriting recognition system

1. **Collect:** Text file provided.
2. **Prepare:** Write a function to convert from the image format to the list fromat in our classifier, `classify0()`.
3. **Analyze:** We'll look at the prepared data in the Python shell to make sure it's correct.
4. **Train:** Doesn't apply to the kNN algorithm.
5. **Test:** Write a function to use some portion of the data as test examples. The test examples are classified against the non-test examples. If the predicted class doesn't match the real class, we'll count that as an error.
6. **Use:** Not performed in this example. You could build a complete program to extract digits from an image, such a system used to sort the mail in the United States.

2.3.1 Prepare: converting images into test vectors

See Jupyter.

2.3.2 Test: kNN on handwriting digits

Listing 2.6 **Handwriting digits testing code: `handwritingClassTest()`**

See Jupyter.

One modification to kNN: kD-trees

→ able to reduce the number of calculations

2.4 Summary

k-Nearest Neighbors (kNN) algorithm:

- Simple and effective
- Instance-based learning
- Carry around the full dataset
- Need to calculate the distance measurement for every piece of data
- Doesn't give ant idea of the underlying structure of the data

3 Splitting datasets one feature at a time: decision trees

A decision tree

- Decision blocks (rectangles)
- Terminating blocks (ovals) → where some conclusion is reached
- Branches → coming out of the decision blocks and leading to other decision blocks or a terminating block

By buliding a decision tree, humans can easily understand the data.

Decision trees are often used in expert systems.

3.1 Tree construction

- **Pros:** Computationally cheap to use, easy for humans to understand learned results, missing values OK, can deal with irrelevant features
- **Cons:** Prone to overfitting
- **Works with:** Numeric values, nominal values

Pseudo-code for `createBranch()`:

```
Check if every item in the dataset is in the same class:
  If so return the class label
  Else
    find the best feature to split the data
    split the dataset
    create a branch node
      for each split
        call createBranch and add the result to the branch node
  return branch node
```

General approach to decision trees:

1. **Collect:** Any method.
2. **Prepare:** This tree-building algorithm works only on nominal values, so any continuous values will need to be quantized.
3. **Analyze:** Any method. You should visually inspect the tree after it is built.
4. **Train:** Construct a tree data structure.
5. **Test:** Calculate the error rate with the learned tree.
6. **Use:** This can be used in any supervised learning task. Often, trees are used to better understand the data.

We'll follow **the ID3 algorithm**, which tells us how to split the data and when to stop splitting it. We're also going to **split on one and only one feature** at a time.

The ID3 algorithm:

The ID3 algorithm begins with the original set S as the root node. On each iteration of the algorithm, it iterates through every unused attribute of the set S and calculates the entropy $H(S)$ or the information gain $IG(S)$ of that attribute. It then selects the attribute which has the smallest entropy (or largest information gain) value. The set S is then split or partitioned by the selected attribute to produce subsets of the data. The algorithm continues to recurse on each subset, considering only attributes never selected before.

Recursion on a subset may stop in one of these cases:

- Every element in the subset belongs to the same class; in which case the node is turned into a leaf node and labelled with the class of the examples.
- There are no more attributes to be selected, but the examples still do not belong to the same class. In this case, the node is made a leaf node and labelled with the most common class of the examples in the subset.
- There are no examples in the subset , which happens when no example in the parent set was found to match a specific value of the selected attribute. Then a leaf node is created and labelled with the most common class of the examples in the parent node's set.

Throughout the algorithm, the decision tree is constructed with each non-terminal node (internal node) representing the selected attribute on which the data was split, and terminal nodes (leaf nodes) representing the class label of the final subset of this branch.

3.1.1 Information gain

Information theory: A branch of science that’s concerned with quantifying information.

Information gain: The change in information before and after the split.

→ Search across every feature for the split that gives the **highest** information gain

Shannon entropy/Entropy: The measure of information of a set.

→ The expected value of the information

The information for symbol x^i is defined as

$$l(x_i) = log_2 p(x_i)$$

where $p(x_i)$ **is the probability of choosing this class**.

The expected value of all the information of all possible values of the class is given by

$$H = -\sum_{i=1}^{n} p(x_i) log_2(x_i)$$

where n **is the number of classes**.

Listing 3.1 **Function to calculate the Shannon entropy of a dataset: `calcShannonEnt()`**

See Jupyter.

Another common measure of disorder in a set is **the Gini impurity**, which is the probability of choosing an item from the set and the probability of that item being misclassified.

3.1.2 Splitting the dataset

Listing 3.2 **Dataset splitting on a given feature: `splitDataSet()`**

```
def splitDataSet(dataSet, axis, value):
    retDataSet = []   # Create separate list
    for featVec in dataSet:
        if featVec[axis] == value:
            reducedFeatVec = featVec[:axis]
            reducedFeatVex.extend(featVec[axis+1:])
            retDataSet.append(reducedFeatVec)   # Cut on the feature split on
    return retDataSet
```

- **Inputs:**
    - *dataSet:* the dataset to split
    - *axis:* the feature to split on
    - *value:* the value of the feature to return
- `reducedFeatVex.extend(featVec[axis+1:])` & `retDataSet.append(reducedFeatVec)`

  The difference between `extend()` and `append()`:

  ```
  >>> a = [1, 2, 3]
  >>> b = [4, 5, 6]
  >>> a.append(b)
  >>> [1, 2, 3, [4, 5, 6]]
  >>> a.extend(b)
  >>> [1, 2, 3, 4, 5, 6]
  ```

Listing 3.3 **Choosing the best feature to split on: `chooseBestFeatureToSplit()`**

```
def chooseBestFeatureToSplit(dataSet):
    numFeatures = len(dataSet[0]) - 1
    baseEntropy = calcShannonEnt(dataSet)
    bestInfoGain = 0.0
    bestFeature = -1
    for i in range(numFeatures):
        featList = [example[i] for example in dataSet]
        uniqueVals = set(featList)   # Create unique list of class labels
        newEntropy = 0.0
        for value in uniqueVals:
```

```
            subDataSet = splitDataSet(dataSet, i, value)
            prob = len(subDataSet) / float(len(dataSet))
            newEntropy += prob * calcShannonEnt(subDataSet)   # Calculate entropy
for each split
        infoGain = baseEntropy - newEntropy
        if infoGain > bestInfoGain:
            bestInfoGain = infoGain
            bestFeature = i   # Find the best information gain
    return bestFeature
```

- `uniqueVals = set(featList)`

  Sets are like lists, but a value can occur only once.

- `newEntropy += prob * calcShannonEnt(subDataSet)`

  The new entropy is calculated and summed up for all the unique values of the selected feature.

- `infoGain = baseEntropy - newEntropy`

  The information gain is the reduction in entropy or the reduction in messiness.

3.1.3 Recursively building the tree

How to create a decision tree:

1. Split the dataset based on the best attribute to split
2. Traverse the data down the branches to another node and split again
3. Stop when
   - running out of attributes on which to split → take a majority vote → label the node
   - all the instances in a branch are the same class → label the node

   and create a leaf node or terminating block

Listing 3.4 **Tree-building code: `createTree()`**

```
def createTree(dataSet, labels):
    classList = [example[-1] for example in dataSet]
    if classList.count(classList[0]) == len(classList):
        return classList[0]   # Stop when all classes are equal
    if len(dataSet[0]) == 1:
        return majorityCnt(classList)   # When no more features, return majority
    bestFeat = chooseBestFeatureToSplit(dataSet)
    bestFeatLabel = labels[bestFeat]
    myTree = {bestFeatLabel:{}}
    del(labels[bestFeat])
    featValues = [example[bestFeat] for example in dataSet]
    uniqueVals = set(featValues)   # Get list of unique values
    for value in uniqueVals:
        subLabels = labels[:]
        myTree[bestFeatLabel][value] = createTree(splitDataSet(dataSet, bestFeat, 
value), subLabels)
    return myTree
```

- **Inputs:**
    - *dataSet:* the dataset
    - *labels:* a list of labels containing a label for each of the features in the dataset
- `del(labels[bestFeat])`

  `del s[i:j]` is the same as `s[i:j] = []` → deleting elements

- `subLabels = labels[:]`

  Make a copy of labels and place it in a new list to keep the original list to be the same every time calling `createTree()`.

- `myTree[bestFeatLabel][value] = createTree(splitDataSet(dataSet, bestFeat, value), subLabels)`

  Recursively call `createTree()` for each split of the dataset.

3.2 Plotting trees in Python with Matplotlib annotations

3.2.1 Matplotlib annotations

Annotation: A tool in Matplotlib that can add text near data in a plot.

Listing 3.5 **Plotting tree nodes with text annotations: `plotNode()` & `createPlot()`**

```
import matplotlib.pyplot as plt

decisionNode = dict(boxstyle = "sawtooth", fc = "0.8")
leafNode = dict(boxstyle = "round4", fc = "0.8")
arrow_args = dict(arrowstyle = "<-")   # Define box and arrow formatting

def plotNode(nodeTxt, centerPt, parentPt, nodeType):
    createPlot.ax1.annotate(nodeTxt, xy = parentPt, xycoords = 'axes fraction', 
xytext = centerPt,
                            textcoords = 'axes fraction', va = "center", ha = 
"center", bbox = nodeType,
                            arrowprops = arrow_args)   # Draw annotations with 
arrows

def createPlot():
    fig = plt.figure(1, facecolor = 'white')
    fig.clf()
    createPlot.ax1 = plt.subplot(111, frameon = False)
    plotNode('a decision node', (0.5, 0.1), (0.1, 0.5), decisionNode)
    plotNode('a leaf node', (0.8, 0.1), (0.3, 0.8), leafNode)
    plt.show()
```

- `createPlot.ax1.annotate(nodeTxt, xy = parentPt, xycoords = 'axes fraction', xytext = centerPt, textcoords = 'axes fraction', va = "center", ha = "center", bbox = nodeType, arrowprops = arrow_args)`

  `annotate(text, xy, *args, **kwargs)` → Annotate the point `xy` with text `text`.

  In the simplest form, the text is placed at `xy`.

  Optionally, the text can be displayed in another position `xytext`. An arrow pointing from the text to the annotated point `xy` can then be added by defining `arrowprops`.

  - `text`: str

    The text of the annotation.

  - `xy`: (float, float)

    The point (x, y) to annotate. The coordinate system is determined by `xycoords`.

  - `xytext`: (float, float), default: `xy`

    The position (x, y) to place the text at. The coordinate system is determined by `textcoords`.

  - `xycoords`: str or Artist or Transform or callable or (float, float), default: 'data'

    The coordinate system that `xy` is given in.

    'axes fraction': Fraction of axes from lower left

  - `textcoords`: str or Artist or Transform or callable or (float, float), default: value of `xycoords`

    The coordinate system that `xytext` is given in.

  - `arrowprops`: dict, optional

    The properties used to draw a FancyArrowPatch arrow between the positions `xy` and `xytext`.

  - `bbox`: dict with properties for patches.FancyBboxPatch

    Draw a bounding box around self.

  - `ha`

    Horizontal alignment.

  - `va`

    Vertical alignment.

- `fig.clf()`

Clear the current figure.

3.2.2 Constructing a tree of annotations

Listing 3.6 **Identifying the number of leaves in a tree and the depth: `getNumLeafs()` & `getTreeDepth()`**

See Jupyter.

Listing 3.7 **The `plotTree` function**

See Jupyter.

3.3 Testing and storing the classifier

3.3.1 Test: using the tree for classification

Listing 3.8 **Classification function for an existing decision tree: `classify()`**

See Jupyter.

3.3.2 Use: persisting the decision tree

Serializing objects allows you to store them for later use, and it can be done with any object.

→ use *pickle*

Listing 3.9 **Method for persisting the decision tree with pickle: `storeTree()` & `grabTree()`**

```python
def storeTree(inputTree, filename):
    import pickle
    fw = open(filename, 'w')
    pickle.dump(inputTree, fw)
    fw.close()

def grabTree(filename):
    import pickle
    fr = open(filename)
    return pickle.load(fr)
```

- `pickle.dump(inputTree, fw)`

  `pickle.dump(obj, file, protocol=None, *, fix_imports=True)` → Write the pickled representation of the object `obj` to the open file object `file`.

- `return pickle.load(fr)`

`pickle.load(file, *, fix_imports=True, encoding="ASCII", errors="strict")` →
Read the pickled representation of an object from the openfile object `file` and return the reconstituted object hierarchy specified therein.

3.4 Example: using decision trees to predict contact lens type

Example: using decision trees to predict contact lens type

1. **Collect:** Text file provided.
2. **Prepare:** Parse tab-delimited lines.
3. **Analyze:** Quickly review data visually to make sure it was parsed properly. The final tree will be plotted with `createPlot()`.
4. **Train:** Use `createPlot()` from section 3.1.
5. **Test:** Write a function to descend the tree for a given instance.
6. **Use:** Persist the tree data structure so it can be recalled without building the tree; then use it in any application.

See Jupyter.

The decision tree here probably matches the data too well, which leads to a problem known as **overfitting**. This problem can be solved by **pruning the tree**, that is, going through and removing some leaves whose nodes add only a little information. Such leaves will be cut off and merged with another leaf.

→ *See Chapter9*

The algorithm used in this chapter (ID3) cannot handle numeric values and may run into some other problems. Another decision tree algorithm CART will be investigated in *Chapter9*.

3.5 Summary

4 Classifying with probability theory: naive Bayes

4.1 Classifying with Bayesian decision theory

- **Pros:** Works with a small amount of data, handles multiple classes
- **Cons:** Sensitive to how the input data is prepared
- **Works with:** Nominal values

For a new measurement with features (x, y), we have:

- the probability of a piece of data belonging to Class1: `p1(x, y)`
- the probability of a piece of data belonging to Class2: `p2(x, y)`

To classify (x, y), we use the following rules:

- If `p1(x,y) > p2(x,y)`, then the class is 1.
- If `p2(x,y) > p1(x,y)`, then the class is 2.

→ **Choose the decision with the highest probability**

4.2 Conditional probability

Bayes' rules:

$$p(c|x) = \frac{p(x|c)p(c)}{p(x)}$$

4.3 Classifying with conditional probabilities

- What we have: $p(x, y|c_1)$, $p(x, y|c_2)$
- What we need to compare: $p(c_1|x, y)$, $p(c_2|x, y)$
- How to get the probabilities we need:

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

The Bayesian classification rule:

- If `p(c1|x,y) > p(c2|x,y)`, the class is c_1.
- If `p(c1|x,y) < p(c2|x,y)`, the class is c_2.

4.4 Document classification with naive Bayes

General approach to naive Bayes:

1. **Collect:** Any method. We'll use RSS feeds in this chapter.
2. **Prepare:** Numeric or Boolean values are needed.
3. **Analyze:** With many features, plotting features isn't helpful. Looking at histograms is a better idea.
4. **Train:** Calculate the conditional probabilities of the independent features.
5. **Test:** Calculate the error rate.
6. **Use:** One common application of naive Bayes is document classification. You can use naive Bayes in any classification setting. It doesn't have to be text.

Assumptions:

- Independence among the features → *naive*
- Every feature is equally important

→ Both are not true in reality.

4.5 Classifying text with Python

Features: tokens gotten from the text, each of which is any combination of characters.

→ Not just words but things like URLs, IP addresses, or any string of characters.

Every piece of text will be reduced to **a vector of tokens** where 1 represents the token existing in the document and 0 represents that it isn't present.

Two categories of online messages: abusive and not.

→ 1 to represent abusive and 0 to represent not abusive

4.5.1 Prepare: making word vectors from text

Listing 4.1 **Word list to vector function: `loadDataSet()` & `createVocabList()` & `setOfWords2Vec()`**

See Jupyter.

4.5.2 Train: calculating probabilities from word vectors

You know whether a word occurs in a document, and you know what class the document belongs to.

$$p(c_i|\mathbf{w}) = \frac{p(\mathbf{w}|c_i)p(c_i)}{p(\mathbf{w})}$$

- $p(c_i)$: add up how many times we see class i (abusive posts or non-abusive posts) and then divide by the total number of posts
- $p(\mathbf{w}|c_i) = p(w_0, w_1, w_2, \ldots, w_N|c_i) = p(w_0|c_i)p(w_1|c_i)p(w_2|c_i)\ldots p(w_N|c_i)$

Pseudocode for `trainNB0()`:

```
Count the number of documents in each class
for every training document:
    for each class:
        if a token appears in the document → increment the count for that token
        increment the count for tokens
    for each class:
        for each token:
            divide the token count by the total token count to get conditional probabilities for each class
```

Listing 4.2 **Naive Bayes classifier training function: `trainNB0()`**

```python
def trainNB0(trainMatrix, trainCategory):
    numTrainDocs = len(trainMatrix)
    numWords = len(trainMatrix[0])
    pAbusive = sum(trainCategory) / float(numTrainDocs)
    p0Num = zeros(numWords)
    p1Num = zeros(numWords)
    p0Denom = 0.0
    p1Denom = 0.0   # Initialize probabilities
    for i in range(numTrainDocs):
        if trainCategory[i] == 1:
            p1Num += trainMatrix[i]
            p1Denom += sum(trainMatrix[i])   # Vector addition
        else:
            p0Num += trainMatrix[i]
            p0Denom += sum(trainMatrix[i])
    p1Vect = p1Num / p1Denom   # Change to log()
    p0Vect = p0Num / p0Denom   # Change to log()
    # Element-wise division
    return p0Vect, p1Vect, pAbusive
```

- `numWords = len(trainMatrix[0])`

  Get the size of the vocabulary.

- `p1Num += trainMatrix[i]`

  Count the time for each word appearing at abusive messages.

- `p1Denom += sum(trainMatrix[i])`

  Count the total number of words of abusive messages.

- `p1Vect = p1Num / p1Denom`

  Get the vector containing $p(w_j|c_1)$.

  This can't be done with regular Python lists.

4.5.3 Test: modifying the classifier for real-world conditions

$$p(\mathbf{w}|c_i) = p(w_0, w_1, w_2, \ldots, w_N|c_i) = p(w_0|c_i)p(w_1|c_i)p(w_2|c_i)\ldots p(w_N|c_i)$$

Two flaws of the prior algorithm:

1. If any $p(w_j|c_i) = 0$, then $p(\mathbf{w}|c_i) = 0$.

   To lessen the impact of this, we'll initialize all of occurrence counts to 1 and the denominators to 2.

2. Underflow: doing too many multiplications of small numbers

   Underflow means the generation of a number that is too small to be represented in the device meant to store it.

   One solution to this is to take the natural logarithm of this product. Such modification won't lose anything.

Listing 4.3 **Naive Bayes classify function: `classifyNB()` & `testingNB()`**

```
def classifyNB(vec2Classify, p0Vec, p1Vec, pClass1):
    p1 = sum(vec2Classify * p1Vec) + log(pClass1)
    p0 = sum(vec2Classify * p0Vec) + log(1.0 - pClass1)
    # Element-wise multiplication
    if p1 > p0:
        return 1
    else:
        return 0

def testingNB():
    listOPosts, listClasses = loadDataSet()
    myVocabList = createVocabList(listOPosts)
    trainMat = []
    for postinDoc in listOPosts:
        trainMat.append(setOfWords2Vec(myVocabList, postinDoc))
    p0V, p1V, pAb = trainNB0(array(trainMat), array(listClasses))
    testEntry = ['love', 'my', 'dalmation']
    thisDoc = array(setOfWords2Vec(myVocabList, testEntry))
    print(testEntry, 'classified as: ', classifyNB(thisDoc, p0V, p1V, pAb))
    testEntry = ['stupid', 'garbage']
    thisDoc = array(setOfWords2Vec(myVocabList, testEntry))
    print(testEntry, 'classified as: ', classifyNB(thisDoc, p0V, p1V, pAb))
```

- `p1 = sum(vec2Classify * p1Vec) + log(pClass1)`

$$
\begin{aligned}
p(c_1|\mathbf{w}) &= \frac{p(\mathbf{w}|c_1)p(c_1)}{p(\mathbf{w})} \\
ln(p(c_1|\mathbf{w})) = ln\left(\frac{p(\mathbf{w}|c_1)p(c_1)}{p(\mathbf{w})}\right) &= ln(p(\mathbf{w}|c_1)) + ln(p(c_1)) - ln(p(\mathbf{w})) \\
&= ln(p(w_0|c_1)p(w_1|c_1)p(w_2|c_1)\ldots p(w_N|c_1)) + ln(p(c_1)) - ln(p(\mathbf{w})) \\
&= \sum_{j=1}^{N} ln(p(w_j|c_1)) + ln(p(c_1)) - ln(p(\mathbf{w}))
\end{aligned}
$$

`sum(vec2Classify * p1Vec)`: $\sum_{j=1}^{N} ln(p(w_j|c_1))$

`log(pClass1)`: $ln(p(c_1))$

4.5.4 Prepare: the bag-of-words document model

- **a bag-of-words model:** can have *multiple* occurrences of each word
- **a set-of-words model:** can have only *one* occurrence of each word

Listing 4.4 **Naive Bayes bag-of-words model: `bagOfWords2VecMN()`**

See Jupyter.

4.6 Example: classifying spam email with naive Bayes

Example: using naive Bayes to classify email

1. **Collect:** Text files provided.
2. **Prepare:** Parse text into token vectors.
3. **Analyze:** Inspect the tokens to make sure parsing was done correctly.
4. **Train:** Use `trainNB0()` that we created earlier.
5. **Test:** Use `classifyNB()` and create a new testing function to calculate the error rate over a set of documents.
6. **Use:** Build a complete program that will classify a group of documents and print misclassified documents to the screen.

4.6.1 Prepare: tokenizing text

See Jupyter.

4.6.2 Test: cross validation with naive Bayes

Listing 4.5 **File parsing and full spam test functions: `textParse()` & `spamTest()`**

See Jupyter.

4.7 Example: using naive Bayes to reveal local attitudes from personal ads

Example: using naive Bayes to find locally used words

1. **Collect:** Collect from RSS feeds. We'll need to build an interface to the RSS feeds.
2. **Prepare:** Parse text into token vectors.
3. **Analyze:** Inspect the tokens to make sure parsing was done correctly.
4. **Train:** Use `trainNB0()` that we created earlier.
5. **Test:** We'll look at the error rate to make sure this is actually working. We can make modifications to the tokenizer to improve the error rate and results.
6. **Use:** We'll build a complete program to wrap everything together. It will display the most common words given in two RSS feeds.

4.7.1 Collect: importing RSS feeds

Universal Feed Parser is the most common RSS library for Python.

More about `feedparser`.

Listing 4.6 **RSS feed classifier and frequent word removal functions: `calcMostFreq()` & `localWords()`**

```
def calcMostFreq(vocabList, fullText):
    import operator
    freqDict = {}
    for token in vocabList:
        freqDict[token] = fullText.count(token)   # Calculates frequency of
occurrence
    sortedFreq = sorted(freqDict.items(), key = operater.itemgetter(1), reverse =
True)
    return sortedFreq[:30]

def localWords(feed1, feed0):
    import feedparser
    docList = []
    classList = []
    fullText = []
    minLen = min(len(feed1['entries']), len(feed0['entries']))
    for i in range(minLen):
        wordList = textParse(feed1['entries'][i]['summary'])   # Accesses one
feed at a time
        docList.append(wordList)
        fullText.extend(wordList)
        classList.append(1)
        wordList = textParse(feed0['entries'][i]['summary'])
        docList.append(wordList)
        fullText.extend(wordList)
        classList.append(0)
    vocabList = createVocabList(docList)
    top30Words = calcMostFreq(vocabList, fullText)
    for pairW in top30Words:
        if pairW[0] in vocabList:
            vocabList.remove(pairW[0])   # Removes most frequently occurring
words
    trainingSet = list(range(2 * minLen))   # Convert to a list since 'range'
object doesn't support item deletion
    testSet = []
    for i in range(20):
        randIndex = int(random.uniform(0, len(trainingSet)))
        testSet.append(trainingSet[randIndex])
        del(trainingSet[randIndex])
    trainMat = []
    trainClasses = []
    for docIndex in trainingSet:
        trainMat.append(bagOfWords2VecMN(vocabList, docList[docIndex]))
        trainClasses.append(classList[docIndex])
    p0V, p1V, pSpam = trainNB0(array(trainMat), array(trainClasses))
    errorCount = 0
    for docIndex in testSet:
        wordVector = bagOfWords2VecMN(vocabList, docList[docIndex])
        if classifyNB(array(wordVector), p0V, p1V, pSpam) != classList[docIndex]:
            errorCount += 1
    print('the error rate is: ', float(errorCount) / len(testSet))
    return vocabList, p0V, p1V
```

- `top30Words = calcMostFreq(vocabList, fullText)`

  The top 30 words in these posts make up close to 30% of all the words used since a large percentage of language is redundancy and *structural glue*. This structural glue can also be removed as well as the most common words from the vocabulary to form a *stop word* list so as to improve the correctness.

4.7.2 Analyze: displaying locally used words

Listing 4.7 **Most descriptive word display function: `getTopWords()`**

```
def getTopWords(nasa, chelsea):
    import operator
    vocabList, p0V, p1V = localWords(nasa, chelsea)
    topNasa = []
    topChe = []
    for i in range(len(p0V)):
        if p0V[i] > -5.0:   # Modified the threshold to limit the list of words
            topChe.append((vocabList[i], p0V[i]))
        if p1V[i] > -5.0:
            topNasa.append((vocabList[i], p1V[i]))
    sortedChe = sorted(topChe, key = lambda pair: pair[1], reverse = True)
    print("CHELSEA:")
    for item in sortedChe:
        print(item[0])
    sortedNasa = sorted(topNasa, key = lambda pair: pair[1], reverse = True)
    print("\n")
    print("NASA:")
    for item in sortedNasa:
        print(item[0])
```

- `sortedChe = sorted(topChe, key = lambda pair: pair[1], reverse = True)`

  `lambda_expr ::=  "lambda" [parameter_list] ":" expression` :

  ```
  def <lambda>(parameters):
      return expression
  ```

  `pair`: `(vocabList[i], p0V[i])`

4.8 Summary

5 Logistic regression

General approach to logistic regression:

1. **Collect:** Any method.
2. **Prepare:** Numeric values are needed for a distance calculation. A structured data format is best.
3. **Analyze:** Any method.
4. **Train:** We'll spend most of the time training, where we try to find optimal coefficients to classify our data.
5. **Test:** Classification is quick and easy once the training step is done.
6. **Use:** This application needs to get some input data and output structured numeric values. Next, the application applies the simple regression calculation on this input data and determines which class the input data should belong to. The application then takes some action on the calculated class.

5.1 Classification with logistic regression ane sigmoid function: a tractable step function

- **Pros:** Computationally inexpensive, easy to implement, knowledge representation easy to interpret
- **Cons:** Prone to underfitting, may have low accuracy
- **Works with:** Numeric values, nominal values

Heaviside step function / Step function:

$$H(x) := \begin{cases} 0, \text{ for } x < 0 \\ \\ 1, \text{ for } x \geq 0 \end{cases}$$

The sigmoid:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

- $\sigma(0) = 0.5$
- $\lim\limits_{z \to +\infty} \sigma(z) = 1$
- $\lim\limits_{z \to -\infty} \sigma(z) = 0$

How a logistic regression classifier works:

1. Take features and multiply each one by a weight and then add them up.
2. Put the result into the sigmoid and get a number between 0 and 1.
3. If the number is above 0.5, then classify the instance as a 1; else, classify as a 0.

5.2 Using optimization to find the best regression coefficients

The input to the sigmoid function is z.

$$z = \mathbf{w}^T\mathbf{x} = w_0x_0 + w_1x_1 + w_2x_2 + \ldots + w_nx_n$$

The vector $\mathbf{x}$ is our input data and we need to find the best coefficients $\mathbf{w}$ so that this classifier will be as successful as possible.

5.2.1 Gradient ascent

To find the *maximum* point on a function, the best way to move is in the direction of the gradient.

$$\nabla f(x,y) = \begin{pmatrix} \frac{\partial f(x,y)}{\partial x} \\ \frac{\partial f(x,y)}{\partial y} \end{pmatrix}$$

- **Direction:** Move in the x direction by amount $\frac{\partial f(x,y)}{\partial x}$ and in the y direction by amount $\frac{\partial f(x,y)}{\partial y}$
- **Magnitude/Step size:** $\mathbf{w} := \mathbf{w} + \alpha\nabla_{\mathbf{w}} f(w)$
- **Stop condition:** (1) a specified number of steps; (2) the algorithm is within a certain tolerance margin

Gradient descent: $\mathbf{w} := \mathbf{w} - \alpha\nabla_{\mathbf{w}} f(w)$

→ To find the *minimum* point on a function

5.2.2 Train: using gradient ascent to find the best parameters

Pseudocode for `gradAscent()`:

```
Start with the weights all set to 1
Repeat R number of times:
    Calculate the gradient of the entire dataset
    Update the weights vector by alpha*gradient
    Return the weights vector
```

Listing 5.1 **Logistic regression gradient ascent optimization functions: `loadDataSet()` & `sigmoid()` & `gradAscent()`**

```python
def loadDataSet():
    dataMat = []
    labelMat = []
    fr = open('/Users/duoduo/Desktop/Data/Machine Learning/Machine Learning in Action/MLA_SourceCode/Ch05/testSet.txt')
    for line in fr.readlines():
        lineArr = line.strip().split()
        dataMat.append([1.0, float(lineArr[0]), float(lineArr[1])])
        labelMat.append(int(lineArr[2]))
    return dataMat, labelMat

def sigmoid(inX):
    return 1.0 / (1 + exp(-inX))

def gradAscent(dataMatIn, classLabels):
    dataMatrix = mat(dataMatIn)
    labelMat = mat(classLabels).transpose()   # Convert to NumPy matrix data type
    m, n = shape(dataMatrix)
    alpha = 0.001
    maxCycles = 500
    weights = ones((n,1))
    for k in range(maxCycles):
        h = sigmoid(dataMatrix*weights)
        error = labelMat - h
        weights = weights + alpha * dataMatrix.transpose() * error   # Matrix
multiplication
    return weights
```

- `h = sigmoid(dataMatrix*weights)`

  The predicted class of each data point. → a *100*1* vector here

- `error = labelMat - h`

  The difference between the real class (1 or 0) and the predicted class (some number between 0 and 1) of each data point.

  → going to move in this direction

  → a *100*1* vector here

5.2.3 Analyze: plotting the decision boundary

List 5.2 **Plotting the logistic regression best-fit line and dataset: `plotBestFit()`**

```
def plotBestFit(wei):
    import matplotlib.pyplot as plt
    weights = wei.getA()
    dataMat, labelMat = loadDataSet()
    dataArr = array(dataMat)
    n = shape(dataArr)[0]
    xcord1 = []
    ycord1 = []
    xcord2 = []
    ycord2 = []
    for i in range(n):
        if int(labelMat[i]) == 1:
            xcord1.append(dataArr[i,1])
            ycord1.append(dataArr[i,2])
        else:
            xcord2.append(dataArr[i,1])
            ycord2.append(dataArr[i,2])
    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.scatter(xcord1, ycord1, s = 30, c = 'red', marker = 's')
    ax.scatter(xcord2, ycord2, s = 30, c = 'green')
    x = arange(-3.0, 3.0, 0.1)
    y = (- weights[0] - weights[1] * x) / weights[2]   # Best-fit line
    ax.plot(x, y)
    plt.xlabel('X1')
    plt.ylabel('X2')
    plt.show()
```

- `weights = wei.getA()`

  `matrix.getA(self)` $\rightarrow$ Return `self` as an *ndarray* object. Equivalent to `np.asarray(self)`.

  This function can convert a matrix to an array.

- `y = (- weights[0] - weights[1] * x) / weights[2]`

  `weights[0] + weights[1] * x + weights[2] * y = 0` $\rightarrow 0 = w_0 x_0 + w_1 x_1 + w_2 x_2$

  Here $x_0 \equiv 1$ (as set in `loadDataSet()`).

  An input of 0 is the center line to split things classified as a 1 and a 0.

5.2.4 Train: stochastic gradient ascent

- **Gradient ascent:** use the whole dataset on each update

  $\rightarrow$ A kind of *batch processing* with all-at-once updating

- **Stochastic gradient ascent:** update the weights using only one instance at a time

  $\rightarrow$ An *online* learning algorithm which can be updated incrementally

Pseudocode for `stocGradAscent0()`:

```
Start with the weights all set to 1
For each piece of data in the dataset:
    Calculate the gradient of one piece of data
    Update the weights vector by alpha*gradient
    Return the weights vector
```

Listing 5.3 **Stochastic gradient ascent: `stocGradAscent0()`**

See Jupyter.

One way to look at how well the optimization algorithm is doing is to see if it's converging to a steady value.

Listing 5.4 **Modified stochastic gradient ascent: `stocGradAscent1()`**

```
def stocGradAscent1(dataMatrix, classLabels, numIter = 150):
    m, n = shape(dataMatrix)
    weights = ones(n)
    for j in range(numIter):
        dataIndex = list(range(m))
        for i in range(m):
            alpha = 4 / (1.0 + j + i) + 0.01   # Alpha changes with each
iteration
            randIndex = int(random.uniform(0, len(dataIndex)))   # Update vectors
are randomly selected
            h = sigmoid(sum(dataMatrix[randIndex]*weights))
            error = classLabel[randIndex] - h
            weights = weights + alpha * error * dataMatrix[randIndex]
            del(dataIndex[randIndex])
    return weights
```

- `alpha = 4 / (1.0 + j + i) + 0.01`

  `j` is the index of the number of times going through the dataset.

  `i` is the index of the example in the training set.

  `alpha` decreases by `1/(j+i)` as the number of iterations increases, which will improve the oscillations, and it will never reach 0 since there is a constant associated. `alpha` doesn't strictly decreasing when `j<<max(i)`.

- `randIndex = int(random.uniform(0, len(dataIndex)))`

  Randomly selecting instance used in updating the weights will reduce the periodic variations of them.

`stocGradAscent1()` converges more quickly.

5.3 Example: estimating horse fatalities from colic

Example: using logistic regression to estimate horse fatalities from colic

1. **Collect:** Data file provided.
2. **Prepare:** Parse a text file in Python, and fill in missing values.
3. **Analyze:** Visually inspect the data.
4. **Train:** Use an optimization algorithm to find the best coefficients.
5. **Test:** To measure the success, we'll look at error rate. Depending on the error rate, we may decide to go back to the training step to try to find better values for the regression coefficients

by adjusting the number of iterations and step size.
6. **Use:** Building a simple command-line program to collect horse symptoms and output live/die diagnosis.

5.3.1 Prepare: dealing with missing values in the data

Options to deal with missing values:

- Use the feature's mean value from all the available data.
- Fill in the unknown with a special value like -1.
- Ignore the instance.
- Use a mean value from similar items.
- Use another machine learning algorithm to predict the value.

Preprocessing:

1. Replace all the unknown values with 0 since it won't impact the weight and the error term during the update. Also, none of the features take on 0 in the data, so in some sense it's a special value.
   - `weights = weights + alpha * error * dataMatrix[randIndex] = weights + alpha * error * 0 = weights`
   - `sigmoid(0) = 0.5`
2. Discard the missing class label.

5.3.2 Test: classifying with logistic regression

Listing 5.5 **Logistic regression classification function: `classifyVector()` & `colicTest()` & `multiTest()`**

See Jupyter.

5.4 Summary

6 Support vector machines

Support vector machines (SVMs) are considered to be the best stock classifier which means it performs well even for datas outside the training set without modification.

6.1 Separating data with the maximum margin

- **Pros:** Low generalization error, computationally inexpensive, easy to interpret results
- **Cons:** Sensitive to tuning parameters and kernel choice; natively only handles binary classification
- **Works with:** Numeric values, nominal values

Terminologies:

- *Linearly separable:* There exists a straight line with all the points of one class on one side of it and all the points of the other class on the other side of it.
- *Separating hyperplane:* The line or hyperplane used to separate the dataset which is applied as the decision boundary of two classes.
- *Margin:* The distance from the separating line/hyperplane to the support vectors. Need to keep the point closest to the separating line/hyperplane as far away from the separating line/hyperplane as possible.
- *Support vectors:* The points closest to the separating line/hyperplane.

6.2 Finding the maximum margin

Separating hyperplane: $\mathbf{w}^T\mathbf{x}+b$

Distance: $\frac{|\mathbf{w}^T\mathbf{x}+b|}{||\mathbf{w}||}$

6.2.1 Framing the optimization problem in terms of our classifier

To get the class label:

$$f(u)=f(\mathbf{w}^T\mathbf{x}+b)=\begin{cases}-1, \text{ for } u<0\\ 1, \text{ for } u\geq 0\end{cases}$$

Why -1 and 1?

We can write a single equation to describe the margin or how close a data point is to our separating hyperplane and not have to worry about if the data is in the *-1* or *+1* class since they are only different by the sign.

Margin calculation: $label*(\mathbf{w}^T\mathbf{x}+b)$

- If a point is far away from the separating plane on the positive side with a positive label, then $label*(\mathbf{w}^T\mathbf{x}+b)$ will be a large positive number.
- If a point is far away from the separating plane on the negative side with a negative label, then $label*(\mathbf{w}^T\mathbf{x}+b)$ will also be a large positive number.

Goal: To find the $\mathbf{w}$ and b values that will define our classifier.

1. Find the points with the smallest margin.
2. Maximize that margin.

$$arg\ \max_{w,b}\left\{\min_n(label\cdot(\mathbf{w}^T\mathbf{x}+b))\cdot\frac{1}{||\mathbf{w}||}\right\}$$

3. Set $label*(\mathbf{w}^T\mathbf{x}+b)$ to be 1 for the support vectors and then maximize $||\mathbf{w}||^{-1}$.
   $\rightarrow label*(\mathbf{w}^T\mathbf{x}+b)$ will be larger than 1 if values are further away from the hyperplane
4. Use *Lagrange multipliers* to solve the constrained optimization problem.

→ the constraint is $label * (\mathbf{w}^T\mathbf{x} + b) \geq 1.0$

$$\max_{\alpha}\left[\sum_{i=1}^{m}\alpha_i - \frac{1}{2}\sum_{i,j=1}^{m} label^{(i)} \cdot label^{(j)} \cdot \alpha_i \cdot \alpha_j < x^{(i)}, x^{(j)} > \right]$$

$$\begin{aligned} s.t. \quad & c \geq \alpha_i \geq 0, \qquad i = 1, \cdots, m \\ & \sum_{i=1}^{m}\alpha_i \cdot label^{(i)} = 0 \end{aligned}$$

The constant c controls weighting between our goal of making the margin large and ensuring that most of the examples have a functional margin of at least 1.0, since the data may not be 100% linearly separable.

5. Write the separating hyperplane in terms of alphas solved.

6.2.2 Approaching SVMs with our general framework

General approach to SVMs:

1. **Collect:** Any method.
2. **Prepare:** Numeric values are needed.
3. **Analyze:** It helps to visualize the separating hyperplane.
4. **Train:** The majority of the time will be spent here. Two parameters can be adjusted during this phase.
5. **Test:** Very simple calculation.
6. **Use:** You can use an SVM in almost any classification problem. One thing to note is that SVMs are binary classifiers. You'll need to write a little more code to use an SVM on a problem with more than two classes.

6.3 Efficient optimization with the SMO algorithm

6.3.1 Platt's SMO algorithm

SMO → Sequential Minimal Optimization

The SMO algorithm works to find a set of alphas and b. Once we have a set of alphas, we can easily compute our weights $\mathbf{w}$ and get the separating hyperplane.

How SMO works:

1. Choose two alphas to optimize on each cycle.
2. Find a suitable pair of alphas meeting criteria below:
    - Both of the alphas have to be outside their margin boundary
    - The alphas aren't already clamped or bounded
3. Once a suitable pair of alphas is found, one is increased and one is deceased.

6.3.2 Solving small datasets with the simplified SMO

Listing 6.1 **Helper functions for the SMO algorithm: `loadDataSet()` & `selectJrand()` & `clipAlpha()`**

See Jupyter.

Pseudocode for `smoSimple()`:

```
Create an alphas vector filled with 0s
While the number of iterations is less than MaxIterations:
    For every data vector in the dataset:
        If the data vector can be optimized:
            Select another data vector at random
            Optimize the two vectors together
            If the vectors can't be optimized → break
    If no vectors were optimized → increment the iteration count
```

Listing 6.2 **The simplified SMO algorithm: `smoSimple()`**

```
def smoSimple(dataMatIn, classLabels, C, toler, maxIter):
    dataMatrix = mat(dataMatIn)
    labelMat = mat(classLabels).transpose()
    b = 0
    m, n = shape(dataMatrix)
    alphas = mat(zeros((m,1)))
    Iter = 0   # 'iter' is a built-in function in Python3, so the variable will
be named as 'Iter'
    while Iter < maxIter:
        alphaPairsChanged = 0
        for i in range(m):
            fXi = float(multiply(alphas, labelMat).T * (dataMatrix *
dataMatrix[i,:].T) + b)
            Ei = fXi - float(labelMat[i])
            if ((labelMat[i]*Ei < -toler) and (alphas[i] < C)) or \
            ((labelMat[i]*Ei > toler) and (alphas[i] > 0)):   # Enter
optimization if alphas can be changed
                j = selectJrand(i, m)   # Randomly select second alpha
                fXj = float(multiply(alphas, labelMat).T * (dataMatrix *
dataMatrix[j,:].T) + b)
                Ej = fXj - float(labelMat[j])
                alphaIold = alphas[i].copy()
                alphaJold = alphas[j].copy()
                if labelMat[i] != labelMat[j]:
                    L = max(0, alphas[j] - alphas[i])
                    H = min(C, C + alphas[j] - alphas[i])
                else:
                    L = max(0, alphas[j] + alphas[i] - C)
                    H = min(C, alphas[j] + alphas[i])   # Guarantee alphas stay
between 0 and C
                if L == H:
                    print("L == H")
                    continue
                eta = 2.0 * dataMatrix[i,:] * dataMatrix[j,:].T \
                - dataMatrix[i,:] * dataMatrix[i,:].T - dataMatrix[i,:] *
dataMatrix[j,:].T
                if eta >= 0:
                    print("eta >= 0")
                    continue
                alphas[j] -= labelMat[j] * (Ei - Ej) / eta
                alphas[j] = clipAlpha(alphas[j], H, L)
                if abs(alphas[j] - alphaJold) < 0.00001:
                    print("j not moving enough")
                    continue
                alphas[i] += labelMat[j] * labelMat[i] * (alphaJold - alphas[j])
                # Update i by same amount as j in opposite direction
                b1 = b - Ei - labelMat[i] * (alphas[i] - alphaIold) *
dataMatrix[i,:] * dataMatrix[i,:].T \
                - labelMat[j] * (alphas[j] - alphaJold) * dataMatrix[i,:] *
dataMatrix[j,:].T
                b2 = b - Ej - labelMat[i] * (alphas[i] - alphaIold) *
dataMatrix[i,:] * dataMatrix[j,:].T \
                - labelMat[j] * (alphas[j] - alphaJold) * dataMatrix[j,:] *
dataMatrix[j,:].T
                if (0 < alphas[i]) and (C > alphas[i]):
                    b = b1
                elif (0 < alphas[j]) and (C > alphas[j]):
                    b = b2
                else:
                    b = (b1 + b2) / 2.0   # Set the constant term
                alphaPairsChanged += 1
                print("iter: %d i: %d, pairs changed %d" % (Iter, i,
alphaPairsChanged))
            if alphaPairsChanged == 0:
                Iter += 1
            else:
                Iter = 0
            print("iteration number: %d" % Iter)
        return b, alphas
```

- **Inputs:**
  - *dataMatIn:* the dataset

    - *classLabels:* the class labels
    - *C:* a constant C
    - *toler:* the tolerance
    - *maxIter:* the maximum number of iterations before quitting

- `Iter = 0`

  The variable `Iter` will hold a count of number of times you've gone through the dataset without any alphas changing. When this number reached the value of the input `maxIter`, you exit.

- `alphaPairsChanged = 0`

  In each iteration, you set `alphaPairsChanged` to 0 and then go through the entire set sequentially. The variable `alphaPairsChanged` is used to record if the attempt to optimize any alphas worked.

- `fXi = float(multiply(alphas, labelMat).T * (dataMatrix * dataMatrix[i,:].T) + b)`

  `fXi` is the prediction of the class of the `i` th instance.

- `Ei = fXi - float(labelMat[i])`

  The error `Ei` is the difference between the prediction and the real class of the `i` th instance. If this error is large, then the alpha corresponding to this data instance can be optimized.

- `if ((labelMat[i]*Ei < -toler) and (alphas[i] < C)) or ((labelMat[i]*Ei > toler) and (alphas[i] > 0))`

  To test if the error `Ei` is large enough to adjust the corresponding alpha. Both the positive and the negative margins are tested, and whether the alpha is equal to 0 or `C` will also be checked since alphas will be clipped at 0 or `C` if they're equal to these, not able to increase or decrease.

- `alphaIold = alphas[i].copy()` & `alphaJold = alphas[j].copy()`

  To make a copy of `alpha[i]` and `alpha[j]` respectively with the `copy()` method for convenience of comparing new alphas and the old ones.

  ```
  if labelMat[i] != labelMat[j]:
      L = max(0, alphas[j] - alphas[i])
      H = min(C, C + alphas[j] - alphas[i])
  else:
      L = max(0, alphas[j] + alphas[i] - C)
      H = min(C, alphas[j] + alphas[i])   # Guarantee alphas stay between 0 and C
  if L == H:
      print("L == H")
      continue
  ```

`L` and `H` are used for clamping `alpha[j]` between 0 and `C`. If `L` and `H` are equal, nothing can be changed and the `continue` statement will be issued to quit the `for` loop.

- `eta = 2.0 * dataMatrix[i,:] * dataMatrix[j,:].T - dataMatrix[i,:] * dataMatrix[i,:].T - dataMatrix[j,:] * dataMatrix[j,:].T`

  `Eta` is the optimal amount to change `alpha[j]`.

  ```
  alphas[j] -= labelMat[j] * (Ei - Ej) / eta
  alphas[j] = clipAlpha(alphas[j], H, L)
  ```

  Calculate a new `alpha[j]` and clip it using `clipAlpha()` with `L` and `H`.

- `alphas[i] += labelMat[j] * labelMat[i] * (alphaJold - alphas[j])`

  `alpha[i]` is changed by the same amount as `alpha[j]` but in the opposite direction.

  ```
  if alphaPairsChanged == 0:
      Iter += 1
  else:
      Iter = 0
  ```

  Check to see if any alphas have been updated. If so, set `iter` to 0 and continue.

6.4 Speeding up optimization with the full Platt SMO

The *only* difference between simplified SMO and the full version of it is how to select which alpha to use in the optimization.

The Platt SMO algorithm:

- The first alpha → outer loop
  - Single passes over the entire dataset
  - Single passes over non-bound alphas

    Create a list of alphas that aren't bounded at the limits 0 or C, and then loop over the list
- The second alpha → inner loop

  This alpha is chosen in a way that will maximize the step size or `Ei - Ej` during optimization by creating a global cache of error values.

Listing 6.3 **Support functions for full Platt SMO: `calcEk()` & `selectJ()` & `updateEk()`**

```
class optStruct:
    def __init__(self, dataMatIn, classLabels, C, toler):
        self.X = dataMatIn
        self.labelMat = classLabels
        self.C = C
```

```
        self.tol = toler
        self.m = shape(dataMatIn)[0]
        self.alphas = mat(zeros((self.m,1)))
        self.b = 0
        self.eCache = mat(zeros((self.m,2)))   # Error cache

def calcEk(oS, k):
    fXk = float(multiply(oS.alphas, oS.labelMat).T * (oS.X * oS.X[k,:].T)) + oS.b
    Ek = fXk - float(oS.labelMat[k])
    return Ek

def selectJ(i, oS, Ei):   # Inner-loop heuristic
    maxK = -1
    maxDeltaE = 0
    Ej = 0
    oS.eCache[i] = [1, Ei]
    validEcacheList = nonzero(oS.eCache[:,0].A)[0]
    if len(validEcacheList) > 1:
        for k in validEcacheList:
            if k == i:
                continue
            Ek = calcEk(oS, k)
            deltaE = abs(Ei - Ek)
            if deltaE > maxDeltaE:
                maxK = k
                maxDeltaE = deltaE
                Ej = Ek   # Choose j for maximum step size
        return maxK, Ej
    else:
        j = selectJrand(i, oS.m)
        Ej = calcEk(oS, j)
    return j, Ej

def updateEk(oS, k):
    Ek = calcEk(oS, k)
    oS.eCache[k] = [1, Ek]
```

- **Functions:**
  - `calcEk(oS, k)`: Calculate an `E` value for a given alpha and return the `E` value.
  - `selectJ(i, oS, Ei)`: Select the second alpha, or the inner loop alpha.
  - `updateEk(oS, k)`: Calculate the error and put it in the cache.
- `__init__(self, dataMatIn, classLabels, C, toler)`

  Populate the member variables of object `optStruct`.
- `self.eCache = mat(zeros((self.m,2)))`

`eCache` is an *m*2* matrix whose first column is a flag bit stating whether `eCache` is valid and the second column is the actual `E` value.

- `validEcacheList = nonzero(oS.eCache[:,0].A)[0]`

  `matrix.A` → Return `self` as an *ndarray* object.

  `nonzero()` → Return a list containing indices of the input list that are not zero.

  This function takes the error value associated with the first choice alpha (`Ei`) and the index `i`. First the input `Ei` needs to be set to valid the in the cache. Valid means that it has been calculated ([0, 0] → [1, `Ei`]).

  Return the alphas corresponding to non-zero `E` values.

Listing 6.4 **Full Platt SMO optimization routine: `innerL()`**

See Jupyter.

Listing 6.5 **Full Platt SMO outer loop: `smoP()`**

```
def smoP(dataMatIn, classLabels, C, toler, maxIter, kTup = ('lin', 0)):
    oS = optStruct(mat(dataMatIn), mat(classLabels).transpose(), C, toler)
    Iter = 0   # 'iter' is a built-in function in Python3, so the variable will
be named as 'Iter'
    entireSet = True
    alphaPairsChanged = 0
    while (Iter < maxIter) and ((alphaPairsChanged > 0) or (entireSet)):
        alphaPairsChanged = 0
        if entireSet:   # Go over all values
            for i in range(oS.m):
                alphaPairsChanged += innerL(i, oS)
                print("fullSet, iter: %d i: %d, pairs changed %d" % (Iter, i,
alphaPairsChanged))
            Iter += 1
        else:   # Go over non-bound values
            nonBoundIs = nonzero((oS.alphas.A > 0) * (oS.alphas.A < C))[0]
            for i in nonBoundIs:
                alphaPairsChanged += innerL(i, oS)
                print("non-bound, iter: %d i: %d, pairs changed %d" % (Iter, i,
alphaPairsChanged))
            Iter += 1
        if entireSet:
            entireSet = False
        elif alphaPairsChanged == 0:
            entireSet = True
        print("iteration number: %d" % Iter)
    return oS.b, oS.alphas
```

- In this function, an iteration is defined as one pass through the loop regardless of what was done, and it will stop if there are any oscillations in the optimization.
- `while (Iter < maxIter) and ((alphaPairsChanged > 0) or (entireSet)):`

  The `while` loop will exit whenever the number of iterations exceeds the specified maximum or the entire set has been passed through without changing any alpha pairs.

  ```
  for i in range(oS.m):
      alphaPairsChanged += innerL(i, oS)
      print("fullSet, iter: %d i: %d, pairs changed %d" % (Iter, i,
  alphaPairsChanged))
  ```

  The first `for` loop goes over any alphas in the dataset and call `innerL()` to choose a second alpha and do optimization if possible.

  ```
  for i in nonBoundIs:
      alphaPairsChanged += innerL(i, oS)
      print("non-bound, iter: %d i: %d, pairs changed %d" % (Iter, i,
  alphaPairsChanged))
  ```

  The second `for` loop goes over all the non-bound alphas, the values that aren't bound at 0 or `C`.

If `C` is large, the classifier will try to make all of the examples properly classified by separating hyperplane..

Classification:

See Jupyter.

6.5 Using kernels for more complex data

6.5.1 Mapping data to higher dimensions with kernels

Mapping from one feature space to another feature space: Transforming the data from one feature space to another, usually going from a lower-dimensional feature space to a higher-dimensional feature space.

→ This mapping is done by a *kernel*.

All of the operations of SVM optimization can be written in terms of *inner products*. Replacing the innner products with a kernel is known as *kernel trick* or *kernel substation*.

6.5.2 The radial bias function as a kernel

A **radial bias function** is a function that takes a vector and outputs a scalar based on the vector's distance. This distance can be either from (0, 0) or from another vector.

The Gaussian version of radial bias function:

$$k(x,y) = exp\left(\frac{-||x-y||^2}{2\sigma^2}\right)$$

σ is a user-defined parameter that determines the "reach", or how quickly this falls off to 0.

Listing 6.6 **Kernel tranformation function: `kernelTrans()`**

```
def kernelTrans(X, A, kTup):
    m, n = shape(X)
    K = mat(zeros((m, 1)))
    if kTup[0] == 'lin':
        K = X * A.T
    elif kTup[0] == 'rbf':
        for j in range(m):
            deltaRow = X[j,:] - A
            K[j] = deltaRow * deltaRow.T
        K = exp(K / (-1*kTup[1]**2))    # Element-wise division
    else:
        raise NameError('Houston We Have a Problem -- That Kernel is not 
recognized')
    return K

class optStruct:
    def __init__(self, dataMatIn, classLabels, C, toler, kTup):
        self.X = dataMatIn
        self.labelMat = classLabels
        self.C = C
        self.tol = toler
        self.m = shape(dataMatIn)[0]
        self.alphas = mat(zeros((self.m,1)))
        self.b = 0
        self.eCache = mat(zeros((self.m,2)))
        self.K = mat(zeros((self.m, self.m)))
        for i in range(self.m):
            self.K[:,i] = kernelTrans(self.X, self.X[i,:], kTup)
```

- `kTup`

  `kTup` is a generic tuple containing the information about the kernel. The first argument in this tuple is a string describing what type of kernel should be used. The other argumrnts are optional arguments that may be needed for a kernel.

  There are two types of kernel:

  1. Linear kernel: A dot product is taken between the two inputs, which are the full dataset and a row of the dataset.

2. The radial bias function: The Gaussian function is evaluated for every element in the matrix in the `for` loop.

Lastly, the function raises an exception if encountering a tuple not able to be recognized.

Listing 6.7 **Changes to `innerL()` and `calcEk()` needed to use kernels**

See Jupyter.

6.5.3 Using a kernel for testing

Listing 6.8 **Radial bias test function for classifying witha kernel: `testRbf()`**

See Jupyter.

6.6 Example: revisiting handwriting classification

Example: digit recognition with SVMs

1. **Collect:** Text file provided.
2. **Prepare:** Create vectors from the binary images.
3. **Analyze:** Visually inspect the image vectors.
4. **Train:** Run the SMO algorithm with two different kernels and different settings for the radial bias kernel.
5. **Test:** Write a function to test the different kernels and different settings for the radial bias kernel.
6. **Use:** A full application of image recognition requires some image processing, which we don't get into.

Listing 6.9 **Support vector machine handwriting recognition: `loadImages()` & `testDigits()`**

See Jupyter.

6.7 Summary

---

7 Improving classification with the AdaBoost meta-algorithm

Meta-algorithms are a way of combining other algorithms.

Classification imbalance is a general problem for all classifiers which occurs when trying to classify items without having an equal number of examples.

7.1 Classifiers using multiple samples of the dataset

- **Pros:** Low generalization error, easy to code, works with most classifiers, no parameters to

adjust
- **Cons:** Sensitive to outliers
- **Works with:** Numeric values, nominal values

Ensemble methods:

- Use different algorithms
- Use the same algorithm with different settings
- ssign different parts of the dataset to different classifiers

7.1.1 Building classifiers from randomly resampled data: bagging

Bootstrap aggregating, which is known as **bagging**, is a technique where the data is taken from the original dataset S times to make S new datasets with the same size as the original, each of which is built by randomly selecting an example from the original with replacement.

→ "With replacement" allows to have values in the new dataset that are repeated, and some values from the original won't be present in the new set.

After the S datasets are built, a learning algorithm is applied to each one individually. When classifying a new piece of data, apply the S classifiers to the new piece of data and take a majority vote.

7.1.2 Boosting

In **boosting**, the different classifiers are trained *sequentially*. Each new classifier is trained based on the performance of those already trained, that is, focusing on data that was previously misclassified by previous classifiers.

The output of boosting is calculated from a *weighted* sum of all classifiers.

AdaBoost is one of the most popular versions of boosting.

General approach to AdaBoost:

1. **Collect:** Any method.
2. **Prepare:** It depends on which type of weak learner you're going to use. In this chapter, we'll use decision stumps, which can take any type of data. You could use any classifier, so any of the classifiers from chapter 2-6 would work. Simple classifiers work better for a weak learner.
3. **Analyze:** Any method.
4. **Train:** The majority of the time will be spent here. The classifier will train the weak learner multiple times over the same dataset.
5. **Test:** Calculate the error rate.
6. **Use:** Like support vector machines, AdaBoost predicts one of two classes. If you want to use it for classification involving more than two classes, then you'll need to apply some of the same methods as for support vector machines.

7.2 Train: improving the classifier by focusing on errors

- **Weak classifier:** A classifier does a better job than randomly guessing but not by much.
- **Strong classifier:** A classifier having a much lower error rate than a weak classifier.

AdaBoost is short for *adaptive boosting*.

How AdaBoost works:

1. A weight is applied to every example in the training data. The weight vector is noted `D`, weights in which are all equal initially.
2. A weak classifier is first trained on the training data. Calculate the errors.
3. Train the weak classifier a second time and adjust the weights of the training set so the examples properly classified the first time are weighted less and those incorrectly classified are weighted more.
4. Assign α values to each of these weak classifiers based on the respective error ϵ given by

$$\epsilon = \frac{number\ of\ incorrectly\ classified\ examples}{total\ number\ of\ examples}$$

5. Calculate α by

$$\alpha = \frac{1}{2} ln\Big(\frac{1-\epsilon}{\epsilon}\Big)$$

6. Update `D` according to α. `D` is given by
   - if correctly predicted:

$$D_i^{(t+1)} = \frac{D_i^{(t)} e^{-\alpha}}{Sum(D)}$$

   - if incorrectly predicted:

$$D_i^{(t+1)} = \frac{D_i^{(t)} e^{\alpha}}{Sum(D)}$$

7. After `D` is calculated, AdaBoost starts on the next iteration, and repeat the training and weight adjusting iterations.
8. Stop when training error is 0 or the number of weak classifiers reaches a user-defined value.

7.3 Creating a weak learner with a decision stump

A *decision stump* is a simple decision tree which makes a decision on one feature only.

Pseudocode for `stumpClassify()` and `buildStump()`:

```
Set the minError to $+\infty$
For every feature in the dataset:
    For every step:
        For each inequality:
            Build a decision stump and test it with the weighted dataset
            If the error is less than minError: set this stump as the best stump
Return the best stump
```

Listing 7.1 **Decision stump-generating functions: `stumpClassify()` & `buildStump()`**

See Jupyter.

7.4 Implementing the full AdaBoost algorithm

Pseudocode for `adaBoostTrainDS()`:

```
For each iteration:
  Find the best stump using buildStump()
    Add the best stump to the stump array
    Calculate alpha
    Calculated the new weight vector -- D
    Update the aggregate class estimate
    If the error rate == 0.0: break out of the for loop
```

Listing 7.2 **AdaBoost training with decision stumps: `adaBoostTrainDS()`**

```python
def adaBoostTrainDS(dataArr, classLabels, numIt = 40):
    weakClassArr = []
    m = shape(dataArr)[0]
    D = mat(ones((m,1)) / m)
    aggClassEst = mat(zeros((m,1)))
    for i in range(numIt):
        bestStump, error, classEst = buildStump(dataArr, classLabels, D)
        print("D: ", D.T)
        alpha = float(0.5 * log((1.0 - error) / max(error, 1e-16)))
        bestStump['alpha'] = alpha
        weakClassArr.append(bestStump)
        print("classEst: ", classEst.T)
        expon = multiply(-1 * alpha * mat(classLabels).T, classEst)
        D = multiply(D, exp(expon))
        D = D / D.sum()    # Calculate D for next iteration
        aggClassEst += alpha * classEst
        print("aggClassEst: ", aggClassEst.T)
        aggErrors = multiply(sign(aggClassEst) != mat(classLabels).T,
ones((m,1)))
        errorRate = aggErrors.sum() / m   # Aggregate error calculation
        print("total error: ", errorRate, "\n")
        if errorRate == 0.0:
            break
    return weakClassArr
```

- `alpha = float(0.5 * log((1.0 - error) / max(error, 1e-16)))`

  The statement `max(error, 1e-16)` is there to make sure not having a divide-by-zero error in the case where there's no error.

7.5 Test: classifying with AdaBoost

Listing 7.3 **AdaBoost classification function: `adaClassify()`**

See Jupyter.

7.6 Example: AdaBoost on a difficult dataset

Example: using AdaBoost on a difficult dataset

1. **Collect:** Text file provided.
2. **Prepare:** We need to make sure the class labels are +1 and -1, not 1 and 0.
3. **Analyze:** Manually inspect the data.
4. **Train:** We'll train a series of classifiers on the data using the `adaBoostTrainDS()` function.
5. **Test:** We have two datasets. With no randomization, we can have an apples-to-apples comparison of the AdaBoost results versus the logistic regression results.
6. **Use:** We'll look at the error rates in this example. But you could create a websute that asks a trainer for the horse's symptoms and then predicts whether the horse will live or die.

Listing 7.4 **Adaptive load data function: `loadDataSet()`**

See Jupyter.

For well-behaved datasets, the test error for AdaBoost reaches a plateau and won't improve with more classifiers.

→ *Overfitting*

AdaBoost and support vector machines are considered to be the most powerful algorithms in supervised learning.

Crossovers:

- The weak learner in AdaBoost as a kernel in support vector machines
- Write the AdaBoost algorithm in terms of maximizing a minimum margin

7.7 Classification imbalance

In most cases, the costs of misclassification are not equal.

7.7.1 Alternative performance metrics: precision, recall, and ROC

1. **Confusion matrix**

Confusion matrix for a two-class problem:

Actual	**Predicted**	
	+1	**-1**
+1	True Positive (TP)	False Negative (FN)
-1	False Positive (FP)	True Negative (TN)

- *Precision* = TP / (TP +FP)

  Precision tells us the fraction of records that were positive from the group that the classifier predicted to be positive.

- *Recall* = TP / (TP + FN)

  Recall measures the fraction of positive examples the classifier got right. Classifiers with a large recall don't have many positive examples classified incorrectly.

2. **ROC curve**

ROC = receiver operating characteristic

- Two axes
  - X-axis: false positive rate

    FPR = FP / (FP + TN)

    The leftmost point corresponds to classifying everything as the negative class.

    The rightmost point corresponds to classifying everything as the positive class.
  - Y-axis: true positive rate

    TPR = TP / (TP + FN)
- Two lines
  - Solid line

    How the two rates FPR & TPR change asthe threshold changes.
  - Dashed line

    The curve you'd get by randomly guessing.

Ideally, the best classifier would be in upper left as much as possible, which means to have a high true positive rate for a low false positive rate.

The area under the curve (AUC) gives an average value of the classifier's performance.

- A perfect classifier: AUC = 1.0
- A random guess: AUC = 0.5

Listing 7.5 **ROC plotting and AUC calculating function: `plotROC()`**

```
def plotROC(predStrengths, classLabels):
    import matplotlib.pyplot as plt
    cur = (1.0, 1.0)
    ySum = 0.0
    numPosClas = sum(array(classLabels) == 1.0)
    yStep = 1 / float(numPosClass)
    xStep = 1 / float(len(classLabels) - numPosClass)
    sortedIndices = predStrengths.argsort()   # Get sorted index
    fig = plt.figure()
    fig.clf
    ax = plt.subplot(111)
    for index in sortedIndices.tolist()[0]:
        if classLabels[index] == 1.0:
            delX = 0
            delY = yStep
        else:
            delX = xStep
            delY = 0
            ySum += cur[1]
        ax.plot([cur[0], cur[0] - delX], [cur[1], cur[1] - delY], c = 'b')
        cur = (cur[0] - delX, cur[1] - delY)
    ax.plot([0,1], [0,1], 'b--')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('ROC curve for AdaBoost Horse Colic Detection System')
    ax.axis([0,1,0,1])
    plt.show()
    print("the Area Under the Curve is: ", ySum * xStep)
```

- `cur = (1.0, 1.0)`

  The variable `cur` holds the cursor for plotting.

- `ySum = 0.0`

  The variable `ySum` is used for calculating the AUC.

- `numPosClas = sum(array(classLabels) == 1.0)`

The variable `numPosClas` holds the number of positive instances.

- `sortedIndices = predStrengths.argsort()`

  `argsort(a, axis = -1, kind = None, order = None)` → Returns the indices that would sort an array.

  Perform an indirect sort along the given axis using the algorithm specified by the `kind` keyword. It returns an array of indices of the same shape as `a` that index data along the given axis in sorted order.

- `for index in sortedIndices.tolist()[0]:`

  `tolist()` → Return the NumPy array od matrix as a (possibly nested) list.

- `delY = yStep`

  Take a step down in the y direction every time getting a class of 1.0, which decreases the true positive rate.

- `delX = xStep`

  Take a step backward in the x direction for every other class, which decreases the false positive rate.

7.7.2 Manipulating the classifier's decision with a cost function

Cost-sensitive learning:

- Example 1

Actual	**Predicted**	
	+1	**-1**
+1	0	1
-1	1	0

  Total cost = TP * 0 + FN * 1 + FP * 1 + TN * 0

- Example 2

Actual	**Predicted**	
	+1	**-1**
+1	-5	1
-1	50	0

  Total cost = TP * (-5) + FN * 1 + FP * 50 + TN * 0

The two types of incorrect classification will have different costs while two types of correct classification will have different benefits.

Select a classifier with the minimum cost.

- AdaBoost: adjust the error weight vector `D` based on the cost function
- Naive Bayes: predict the class with the lowest expected cost instead of the class with the highest probability
- SVMs: use different `C` parameters in the cost function for the different classes → more weight to the smaller class in which allowing fewer errors

7.7.3 Data sampling for dealing with classification imbalance

To alter the data used to train the classifier to deal with imbalanced classification tasks:

- **Oversample:** to duplicate examples
- **Undersample:** to delete examples

These two ways can be done either randomly or in a predetermined fashion.

The rare case is the positive class whose information needs to be preserved as much as possible, so the examples from the negative class should be undersampled or discarded.

→ Drawback: Don't know which negative examples to toss out since those discarded may carry valuable information that the remaining examples don't contain.

→ Solution: Discard samples that are not near the decision boundary, or use a hybrid approach of undersampling the negative class and oversampling the positive class

To oversample the positive class, you could replicate the existing examples or add new points similar to the existing points like adding a data point interpolated between existing data points. This process can lead to *overfitting*.

7.8 Summary