Machine Learning and Data Mining

Assignment 3

Adrian Cooney (4BCT, 12394581)

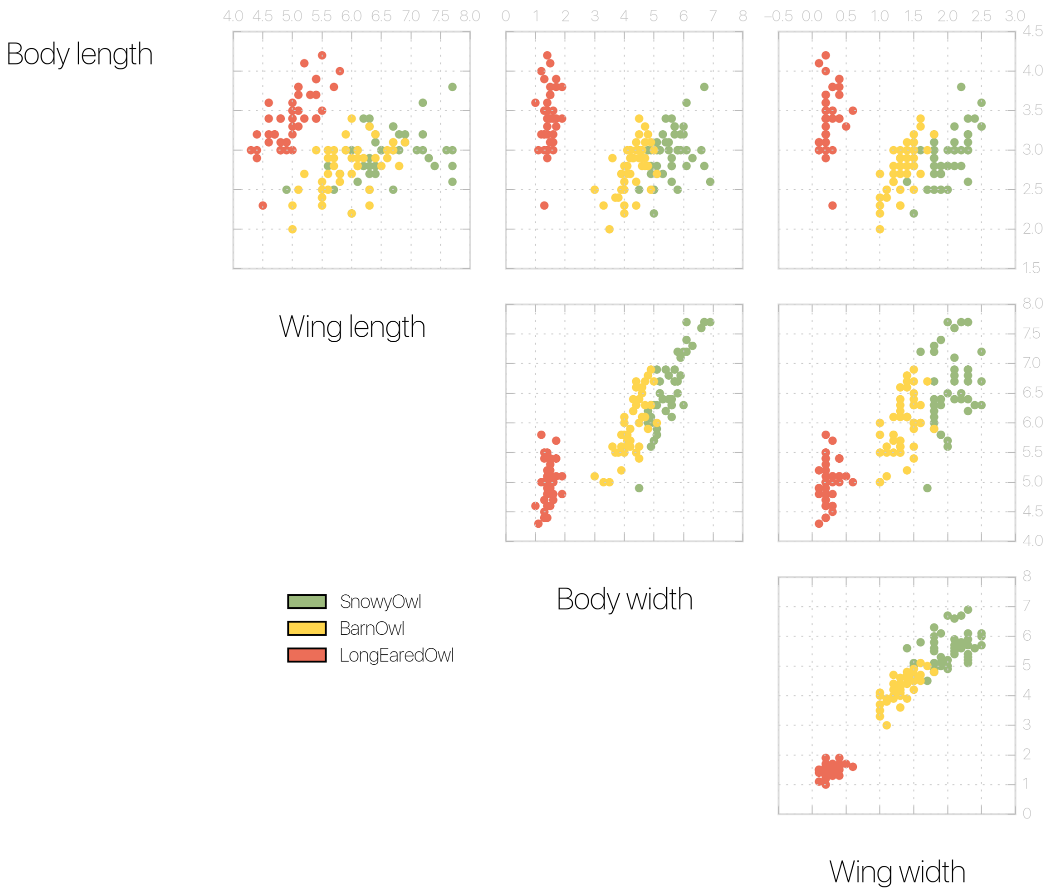

Figure 1: Visualization of the owls.csv data.

Introduction

For this assignment, I spent quite a while picking my algorithm. It was a long process of researching the different algorithms such as C4.5, Neural Networks and even an attempt at my own algorithm (but failed). In the end I chose Support Vector Machines and didn't regret the choice. Once I got linear algebra skills up to scratch and wrapped my head around the math, I could appreciate the algorithm (and it's history) and implement it.

Design Decisions

Support Vector Machines

The classifier I chose to implement was the Support Vector Machine with a Radial Basis Function (RBF) kernel. The reasons for my choice of classifier are as follows:

1. Support Vector Machines have many great resources for understanding the mechanics behind it. Some examples:
   a. MIT OpenCourseWare SVM lecture.[1]
   b. Duality and Geometry in SVM Classifiers [1].
   c. A Practical Guide to Support Vector Classification [2].
   d. Everything you Wanted to know about the Kernel Trick.[2]

2. Preliminary tests using Scikit-Learn's Support Vector Classifier (`sklearn.svm.SVC`) classifier showed it performed well on the data provided. This also served as a basis for comparison with my implementation.

3. Fitted with the requirements of the assignment as being non-trivial.

Python

The language I chose for my implementation of an SVM was Python. Aside from working extensively in it this term and building a great familiarity with the API, Python seems to be the first language for Machine Learning in general and has many resources available. The language itself is concise and very easy to read. Python also has a wealth of libraries and frameworks for every aspect of this assignment which made handling the verbose things of importing the CSV file trivial.

[1]http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-034-artificial-intelligence-fall-2010/lecture-videos/lecture-16-learning-support-vector-machines/
[2]http://www.eric-kim.net/eric-kim-net/posts/1/kernel_trick.html

Libraries and external resources

The following libraries are used in my implementation SVM:

1. Pandas[3] - Used for importing the CSV into a workable dataframe structure.
2. CVXOPT[4] - Convex optimization package.
3. Numpy[5] - Scientific computing package (de-facto).
4. Scikit-Learn[6] (SKLearn) - Machine learning package.

Other resources used:

1. Py.test[7] - Full-featured Python testing tool.
2. Jupyter[8] - Interactive computing application.

Code structure

Although the code itself is well documented, here is a high level overview and the structure. The code lives in the `src/` folder with the following files:

1. `SVC.py`

   This is where the implementation of the Support Vector Classifier lives. This is a **binary** classifier. The class itself inherits from SKLearn's `BaseClassifier` and `ClassifierMixin`[9] for compatibility with the metric analysis system for scoring and cross validation. The class implements the SKLearn Classifier interface of two methods `fit(X, y)` and `predict(X)` to allow for classifying any data.

2. `MulticlassSVC.py`

   Here is the implementation of the Multi-class Support Vector classifier. It is also implements the SKLearn classifier interface of `fit` and `predict`.

3. `MulticlassSVC_test.py`

   The tests for the `MulticlassSVC` are kept in this file. The results below are created from this file.

[3]Pandas - http://pandas.pydata.org/
[4]CVXOPT - http://cvxopt.org/
[5]Numpy - http://www.numpy.org/
[6]Scikit-Learn - http://scikit-learn.org/
[7]Py.test - http://pytest.org/
[8]Jupyter - http://jupyter.org/
[9]Rolling your own classifier, http://scikit-learn.org/stable/developers/contributing.html#rolling-your-own-estimator

Algorithm

The algorithm below describes the multi-class SVC which includes the binary SVC (since a multi-class SVC is just an ensemble of binary SVCs).

Algorithm 1 Train and predict a SVM multi-class classifier with feature set X and labels y.

```
$classifiers \leftarrow Map$
$labels \leftarrow Array$
procedure TRAIN($X, y$)                              ▷ Procedure to train the Multiclass SVC
    $labels \leftarrow group(y)$    ▷ Group labels (like an SQL GROUP BY statement)
    for $label$ in $labels$ do
        $iy \leftarrow y[y = label]$     ▷ Convert labels to label and notlabel (true/false)
        $classifier[label] \leftarrow \mathrm{SVC}(X, iy)$                          ▷ Create a
    end for
end procedure
procedure PREDICT($X$)
    $predicted \leftarrow Array$
    for $x$ in $X$ do
        for $label$ in $labels$ do
            $svc \leftarrow classifiers[label]$              ▷ Get the classifier for this label
            $output \leftarrow svc.predict(x)$                          ▷ Ask it to predict x
            if $output$ is True then
                $predicted.append(label)$               ▷ Push the classifier's label
            end if
        end for
    end for
return $predicted$
end procedure
```

Algorithm 2 Train and predict a SVM binary classifier with feature set X and labels y.

```
$b \leftarrow 0$
$X_{sv} \leftarrow Array$                                ▷ The support vector features.
$y_{sv} \leftarrow Array$                                ▷ The support vector labels.
$\mathcal{L}_{sv} \leftarrow Array$
procedure TRAIN($X, y$)
    $k \leftarrow Matrix \in \mathbb{R}^{|X| \times |X|}$
    for $y < |X|$ do                                    ▷ Precompute the gram matrix
        for $x < |X|$ do
            $k_{x,y} \leftarrow K(X_y, X_x)$            ▷ Where K is the RBF kernel
        end for
    end for
    Solve the convex optimization problem to find the support vectors
```

$$\begin{aligned} \mathcal{L} &= \min_x \quad \frac{1}{2} x^T P x + q^T x \\ \text{Subject to} \qquad Gx &\leq h \\ Ax &= b \\ \\ P &= k \cdot (y \otimes y) \\ q &= u_{1 \times |X|} \quad \text{and} \quad u_i = -1 \\ G &= -I_{|X| \times |X|} \\ h &= 0, h \in \mathbb{R}^{1 \times |X|} \\ A &= u_{|X| \times 1} \quad \text{and} \quad u_i = 1 \\ b &= 0 \\ X_{sv} &= X_i, \ where \quad \mathcal{L}_i > 1^{-5} \\ y_{sv} &= y_i, \ where \quad \mathcal{L}_i > 1^{-5} \\ b &= \sum_{i}^{|X_{sv}|} y_{svi} - \sum \mathcal{L} \cdot y_{sv} \cdot k_{i, X_{sv}} \\ b &= b / |X_{sv}| \end{aligned}$$

```
end procedure
procedure PREDICT($X$)
    $predictions \leftarrow Array$
    for $x$ in $X$ do
        $prediction = b + \sum_{i}^{|X_{sv}|} \mathcal{L} \cdot y_{svi} \cdot K(x, X_{svi})$
        predictions.append(sign(prediction))
    end for
return $predictions$
end procedure
```

Results

The following results are the output from running the classifier on a owl data where the train and test data is randomly selected with a split of 66% (89/46). The test was run 10 times.

Test	1	2	3	4	5	6	7	8	9	10	**Mean**
Score	0.98	0.98	0.89	0.98	0.93	0.98	0.93	0.98	0.98	0.98	**0.961**

Using SKLearn's `cross_val_score` cross validation module, the classifier received a score of **0.948** with 5 folds.

Conclusions

Support Vector Machines are a very beautiful and complex beast that perform exceptionally well on the owl data. Some test results yielded 1.0 score on some runs and make it a very good fit to classify this data.

Appendix 1: Code

src/SVC.py

```
import numpy as np
import cvxopt
from cvxopt import matrix, solvers
from sklearn.base import BaseEstimator, ClassifierMixin

class SVC(BaseEstimator, ClassifierMixin):
    @staticmethod
    def gaussian_kernel(x, y):
        # Formula for the kernel:
        #   K(x, x') = exp( -\sqrt{\frac{|| x - x' ||^2}{2\sigma^2}})
        return np.exp(-np.sqrt(np.linalg.norm(x - y) ** 2 / (2 * 0.5 ** 2)))

    def __init__(self, kernel = None):
        if not kernel:
            # Default to Radial Basis Function (RBF) kernel
            kernel = SVC.gaussian_kernel

        self.kernel = kernel
        self.b = 0
```

```python
def fit(self, X, y):
    """Fit the data into the classifier:

    Args:

        X: (numpy.ndarray)

        The n*m dimensional array where n is the amount of features
        and m is the amount of samples.

        y: (numpy.ndarray)

        Array of labels for the sample. This is a BINARY classifier (`bool(label)`).
    """

    # Ensure the labels are binary. Convert labels to True/False.
    # WARNING: None, 0 and False are the considered False and any other
    # value is considered True. See what python's truthy values for more
    # information. Here we convert the labels to 1 or -1
    y = np.array([1 if bool(n) else -1 for n in y])

    sample_count, feature_count = X.shape

    # Create the memory for the kernel space. This is a matrix
    # that contains the simliarity of each vector to each other.
    K = np.zeros((sample_count, sample_count))

    # Generate a matrix of the similarities of each
    # feature against each other.
    for i in range(sample_count**2):
        j = i % sample_count
        k = int(np.floor(i / sample_count))

        # Apply the kernel trick (RBF in our case)
        K[j, k] = self.kernel(X[j], X[k])

    # Since the Support Vector machine is a convex optimization
    # problem, we can solve it using a convex optimzation package,
    # and in our case CVXOPT. We have to mangle our input to suit
    # the standard form QP:
    #
    #   min_x   \frac{1}{2} x^T Px + q^T x
    #   s.t.    Gx \leq h
    #           Ax = b
    #
    # See: https://courses.csail.mit.edu/6.867/wiki/images/a/a7/Qp-cvxopt.pdf
    P = np.outer(y, y) * K
    q = np.ones(sample_count) * -1

    # Inequality constraint for each sample
    #    -I^(sample_count*sample_count) < 0
    G = np.diag(np.ones(sample_count) * -1)
    h = np.zeros(sample_count)

    # Equality constraint for each samples label
    A = matrix(y, (1, sample_count), "d") # Cast to double
    b = 0.0

    # Solve the quadratic programming problem using CVXOPT
    solved = solvers.qp(*tuple(
        [matrix(u) if not isinstance(u, cvxopt.base.matrix) else u \
            for u in [P, q, G, h, A, b]]
        )
    )

    # Grab the multipliers
    multipliers = np.array(solved["x"]).flatten()

    # Now select the indices of the support vectors that are over the threshold
    selection = multipliers > 1e-5
    index = np.arange(len(multipliers))[selection]

    # Grab the support vectors
    self.support_vectors = X[selection]
    self.support_vector_multipliers = multipliers[selection]
    self.support_vector_labels = y[selection]

    # Now calculate the intercept b
    b = 0
    for i in range(len(self.support_vector_multipliers)):
        b += self.support_vector_labels[i]
        b -= np.sum(self.support_vector_multipliers * \
            self.support_vector_labels * \
            K[index[i], selection])

    # Normalize and save b for prediction
    self.b = b / len(self.support_vector_multipliers)

def predict(self, X):
    """Predict the label y for each sample in X"""
    labels = []
```

```
        for i in range(len(X)):
            prediction = 0

            for multiplier, label, support_vector in zip(self.support_vector_multipliers,
                self.support_vector_labels, self.support_vectors):

                # Calculate the label by summing all the lagrange multipliers
                prediction += multiplier * label * self.kernel(X[i], support_vector)

            labels.append(prediction)

        labels = np.array(labels) + self.b

        return np.sign(labels)

    def __repr__(self):
        """Pretty print the classifier"""
        return "SVC(kernel={}, b={})".format(self.kernel.__name__, self.b)
```

src/MulticlassSVC.py

```
import numpy as np
from itertools import groupby
from sklearn.base import BaseEstimator, ClassifierMixin
from SVC import SVC


class MulticlassSVC(BaseEstimator, ClassifierMixin):
    def __init__(self):
        self.classifiers = {} # Store the classifiers <label:str>: <LinearSVC>
        self.labels = None

    def fit(self, X, y):
        """Fit the data to the classifier.

        This generates N classifier's where N is the amount of features within the data.
        We then generate hyperplanes within each classifier to which can classify whether
        a new data point *is the feature* and *not the feature* (binary).

        Args:

            X: (numpy.ndarray)

            The n*m dimensional array where n is the amount of features
            and m is the amount of samples.

            y: (numpy.array)

            An array of labels for the samples. Length == sample_count
        """
        self.labels = MulticlassSVC.labels(y)
        sample_count, feature_count = X.shape

        print "Fitting {} samples with {} features and {} labels: {}".format(sample_count, f

        # Loop over each label in the set and generate a classifier
        # that can decide between is label or is not label.
        for label in self.labels:
            # Convert the labels to boolean
            ny = np.array([l == label for l in y])

            # Create the classifier
            classifier = SVC()

            # Fit the data
            classifier.fit(X, ny)

            # And save it for voting in the prediction
            self.classifiers[label] = classifier

    def predict(self, X):
        """Predict the label for each sample in X

        Args:

            X: (numpy.ndarray)

            The sample to predict the label for.

        Returns:
            y: (numpy.ndarray)

            Returns labels for each sample
        """
        y = []

        print self.classifiers

        # Loop over each sample in X
        for sample in X:
            # Loop over each classifier and check if the label returns true
            for label, classifier in self.classifiers.iteritems():
                # Prediect the label in the classifier
                [prediction] = classifier.predict([sample])

                # Set the sample = label if the classifier returns true
                if prediction > 0:
                    y.append(label)
                    break

        return np.array(y)

    def __repr__(self):
        output = "MulticlassSVC: {} labels \"{}\"\n".format(len(self.labels), "\", \"".join

        for label, classifier in self.classifiers.iteritems():
            output += "  {:>12} -> {}\n".format(label, repr(classifier))

        return output

    @staticmethod
    def labels(y):
        """Group the labels and return them.

        Args:
            y: (list) List of labels.
        """

        return [label for label, ls in groupby(sorted(y))]
```

test/MulticlassSVC_test.py

```
import pandas
import numpy as np
from os import path
from MulticlassSVC import MulticlassSVC
from sklearn.cross_validation import cross_val_score

# Read in the data from the CSV file
data = pandas.read_csv(path.join(path.dirname(__file__), "../data/owls.csv"))

def test_multiclass_svc_cv():
    owl_classifier = MulticlassSVC()

    X = data[["body-length", "body-width", "wing-length", "wing-width"]].values
    y = data["species"].values
    folds = 5

    print "Cross validation ({} folds) score: {:.3f}".format(
        folds, np.mean(cross_val_score(owl_classifier, X, y, cv=folds, verbose=5))
    )

def test_multiclass_svc():
    owl_classifier = MulticlassSVC()

    X = data[["body-length", "body-width", "wing-length", "wing-width"]].values
    y = data["species"].values

    np.random.seed(13)
    index = np.arange(len(X))
    split = int(np.floor(len(index) * 0.66))

    def score_classifier():
        np.random.shuffle(index)

        # Split the samples in train/test
        train_index, test_index = index[:split], index[split:]
        train_features, train_labels = X[train_index], y[train_index]
        test_features, test_labels = X[test_index], y[test_index]

        # Fit the training data
        owl_classifier.fit(train_features, train_labels)

        # Score the classifier (inherited from ClassifierMixin)
        return owl_classifier.score(test_features, test_labels)

    scores = [score_classifier() for i in range(10)]

    print "Train/test split: %d/%d (%d total)" % (split, len(X) - split, len(X))
    print "Scores: " + ", ".join([("%.2f" % s) for s in scores])
    print "Mean score: %.3f" % np.mean(np.array(scores))
```

Bibliography

[1] K. P. Bennett and E. J. Bredensteiner, “Duality and geometry in sVM classifiers.”

[2] C.-W. Hsu, C.-C. Chang, C.-J. Lin, and others, “A practical guide to support vector classification.” 2003.