LOOKING AT BOOSTING ALGO'S

KARL MUTCH

UCB INTRODUCTION TO WEB ALGORITHMS, 2010.

GOAL:

This week I choose to experiment with the ensemble learning classifiers.

INTENDED RESULTS:

The goal is to experiment with the results of the AdaBoost algorithm and comment on the findings. I also dug further into the arc4x and the boost factor used with it.

WEEK 5 : TASKS

The first tasks involved research on AdaBoost and its relative advantages and disadvantages over the ArcX4 method of ensemble learning. The second task I undertook and which I recorded as the programming portion, due to its limited size, was to fiddle with the multiplier being used to determine its affect on the results.

WEEK 5 : DIARY

RESEARCH AND COMMENT

The text used within class, Algorithms of the Intelligent Web, contains a description of the AdaBoost Algorithm and makes the suggestion that an exercise of use was to implement and compare it with the ArcX4 implementation. In researching this is became apparent that several core disadvantages exist with the AdaBoost mechanisim.

The boosting mechanism in general looked to be vulnerable to overfitting. When running AdaBoost examples from WEKA it appeared that the algorithm did avoid this issue but principally I think this is because we have very clean data. In the case of WEKA there appeared to be a simplified version of the original algorithm implemented within the WeightedInstancesHandler. When using AdaBoost it biases towards items that previously have failed to be classified correctly, this allows it to avoid fixating on correct classifications but I think makes it prone to outliers.

In our case when I returned to the existing ArcX4 implementation and then began to experiment with the multiplier factor I did discover that in the larger population cases there appears to be a decrease in accuracy at certain points.

One way to deal with this might be to use the learning itself to identify and generate new training sets with controlled injection of new training data.

PROJECT TEST

The base implementation of the ArcX4 gave the following tabulated results.

Decision Tree			Naive Bayes			Neural Network		
Population	Accuracy	Seconds	Population	Accuracy	Seconds	Population	Accuracy	Seconds
1	0.61523	1	1	0.66068	22	1	0.58847	7
3	0.63685	1	3	0.66596	66	3	0.62204	21
5	0.64091	2	5	0.66771	114	5	0.63467	35
7	0.65041	2	7	0.66113	162	7	0.62287	49
11	0.654	3	11	0.67033	255	11	0.60167	77
31	0.65697	9	31	0.67178	697	31	0.63064	212
41	0.64998	11	41	0.67108	915	41	0.64376	298
61	0.65467	16	61	0.66964	1475	61	0.63925	475

Given that the power to which the weights are raised will not affect the run times the remaining tables will leave out the time taken to compute results.

When running the first test using powers of 3.5, 4.0, and 4.5 and populations of we can see that the algorithms reacted in very different ways.

To complete this test I also modified the implementation of the ArcX4 to allow weights to be passed in from our test harness. To do this I changed the type of the power variable to using float. This had an effect on the precision of the calculations and resulted in improvements that will be seen in runs of tests previously performed.

Decision Tree			Naïve Bayes			Neural Network		
Power	Accuracy (1)	Accuracy (3)	Power	Accuracy (1)	Accuracy (3)	Power	Accuracy (1)	Accuracy (3)
3.5	0.61407	0.63672	**3.5**	0.65155	0.66645	**3.5**	0.61901	0.63545
4.0	0.60840	0.64946	**4.0**	0.65078	0.66406	**4.0**	0.59101	0.61480
4.5	0.56838	0.63482	**4.5**	0.66803	0.65456	**4.5**	0.35381	0.62936

The power does contribute a lot to the accuracy rates. The ad-hoc nature of the power is rather more ad-hoc than expected especially in the small population. The Bayes is the only one to have survived power which suggests that it may have a greater discrimination with poor scoring data. When continuing the experiment I decided to narrow down the range of power and choose 3.8, 4.0, and 4.2 to try and focus on a useful range of values.

	Decision Tree							
Power	Accuracy (1)	Accuracy (3)	Accuracy (5)	Accuracy (7)	Accuracy (11)	Accuracy (31)	Accuracy (41)	Accuracy (61)
3.8	0.61889	0.65796	0.65251	0.64984	0.65132	0.65969	0.65621	0.65909
4	0.62749	0.63478	0.63785	0.65232	0.65735	0.65984	0.65781	0.65959
4.2	0.59986	0.63671	0.63651	0.65043	0.64544	0.64917	0.65609	0.66005

	Naive Bayes							
Power	Accuracy (1)	Accuracy (3)	Accuracy (5)	Accuracy (7)	Accuracy (11)	Accuracy (31)	Accuracy (41)	Accuracy (61)
3.8	0.65637	0.66475	0.67062	0.66292	0.66766	0.67257	0.67234	0.67032
4	0.67070	0.66437	0.66738	0.66533	0.67261	0.67241	0.67173	0.67080
4.2	0.66307	0.66271	0.66369	0.66632	0.66988	0.67094	0.67057	0.67175

	Neural Network							
Power	Accuracy (1)	Accuracy (3)	Accuracy (5)	Accuracy (7)	Accuracy (11)	Accuracy (31)	Accuracy (41)	Accuracy (61)
3.8	0.59297	0.60940	0.63344	0.59879	0.63677	0.63793	0.63316	0.63837
4	0.56641	0.59398	0.60642	0.63536	0.62840	0.63551	0.63600	0.64266
4.2	0.59372	0.60096	0.61079	0.59836	0.63265	0.63780	0.64190	0.64149

On this occasion we can see that the differences between the approaches did narrow. The results also show some amount of instability between how the power function performed between population counts; given the Ad-Hoc nature of the ensemble we can expect some amount of this. The decision tree appears to remain close enough to justify the reduction in runtimes for a live scenario.

LESSONS LEARNED

The execution times for each test case remain with decision tree at 1-2 seconds the neural net at 7 – 60 seconds, and the Naïve Bayes at 3 times the neural net. While the bayes did outperform on accuracy careful analysis by experts with the data would need to consider the cost of the computation.

Having said this Naïve Bayes does lend itself to using a framework such as Mahout, for which there is a MapReduce class https://cwiki.apache.org/MAHOUT/bayesian.html.

With using the ArcX4 it would always pay to completely benchmark the algorithm and data chosen for the trainer as this could have a large effect on the results. Especially in light of the results we obtained. It would be hard to justify the change from the default power of 4 being used. In all of our testing 4 was the only power which seemed to produce results that improved in a steady predictable manner as the population increased. This alone might be worth sticking with the number as it is a least one point of predictability.

Again as with all of the recent projects undertaken, use of Hadoop would be of great help to anyone performing large scale measurements and testing. The largest run I undertook ran for over 8 hours for several populations and algorithms within the ensemble.

IMPLEMENTATION

The running of this code was done using Eclipse. Two new classes were added to the system. The Main class contains the test itself including code permuting the tests. This code used a RunResult class to store intermediate test results as shown below:

```
/**
 * @brief This class allows test results to be stored within a sorted collection
 */
public class RunResult implements Comparable<RunResult> {
    String trainer;
    int population;
    float boostFactor;
    int classified;
    int misclassified;
    double accuracy;
    long timeSeconds;

    RunResult (String inTrainer, int inPopulation, float inBoostFactor, int inClassified, int inMisclassified,
double inAccuracy, long inTimeSeconds)
    {
        trainer = inTrainer;
        population = inPopulation;
        boostFactor = inBoostFactor;
        classified = inClassified;
        misclassified = inMisclassified;
        accuracy = inAccuracy;
        timeSeconds = inTimeSeconds;
    }

    public String getTrainer() {
        return(trainer);
    }

    public int hashCode() {
        int hash = 17;
        hash = hash * 31 + trainer.hashCode();
        hash = hash * 16 + population;
        hash = hash * 16 + (int)(boostFactor * 10);
        return hash;
    }

    public boolean equals(Object other)
    {
        if (this == other) return true;
        if (other == null || getClass() != other.getClass()) return false;

        RunResult theResult = (RunResult) other;

        if (trainer != null ? !trainer.equals(theResult.trainer) : theResult.trainer != null) return false;
        if (population != theResult.population) return false;
        if (boostFactor != theResult.boostFactor) return false;

        return true;
    }

    @Override
    public int compareTo(RunResult other) {

        if (equals(other)) {
            return(0);
        }

        if (trainer.compareTo(other.trainer) > 0) {
            return(1);
        }

        if (trainer.compareTo(other.trainer) < 0) {
            return(-1);
        }

        if (population > other.population) {
            return(1);
        }

        if (population < other.population) {
            return(-1);
        }

        if (boostFactor > other.boostFactor) {
            return(1);
        }

        if (boostFactor < other.boostFactor) {
            return(-1);
        }

        return (0);
    }

}
```

The RunResults class contains methods to allow for sorting of the results when the tests have completed. Unfortunately the short time frame did not allow for extensive comments.

```
/**
 * @author karl Mutch
 *
 * Assignment for Week 5 of the Web Algorithms Class Summer 2010
 */
import java.util.Set;
import java.util.TreeSet;

import iweb2.ch6.usecase.credit.BoostingCreditClassifier;
import iweb2.ch6.usecase.credit.data.UserDataset;
import iweb2.ch6.usecase.credit.data.UserLoader;
import iweb2.ch6.usecase.credit.util.CreditErrorEstimator;


public class Main {

    public static void main(String[] args) {

        Set<RunResult> results = new TreeSet<RunResult>();
        UserDataset ds = UserLoader.loadTrainingDataset();
        UserDataset testDS = UserLoader.loadTestDataset();

        float [] boostFactors = {3.8f, 4.0f, 4.2f};
        String [] trainers = {"decision tree", "neural network", "naive bayes"};

        for (String aTrainer : trainers) {

            for (float boostFactor : boostFactors) {
                BoostingCreditClassifier arcx4 = new BoostingCreditClassifier(ds, boostFactor);

                // set verbose level to true to see more details.
                // ATTENTION: If set to true then every classification will be reported
                arcx4.setVerbose(false);
                arcx4.setClassifierType(aTrainer);

                int [] populations = { 1, 3, 5, 7, 11, 31, 41, 61 };

                for (int populationCount : populations) {

                    arcx4.setClassifierPopulation(populationCount);
                    arcx4.train();

                    CreditErrorEstimator arcx4ee  = new CreditErrorEstimator(testDS, arcx4);
                    long begin = System.currentTimeMillis();

                    arcx4ee.run();

                    {
                        RunResult runResult = new RunResult(aTrainer, populationCount, boostFactor,
arcx4ee.getCorrectCount(),
                                    arcx4ee.getMisclassifiedInstanceCount(),  arcx4ee.getAccuracy(),
                                    (System.currentTimeMillis() - begin) / 1000);
                        results.add(runResult);
                    }
                }
            }
        }

        System.out.println("Trainer,Population,BoostFactor,Accuracy,Seconds");
        for (RunResult aResult : results) {
            System.out.println(aResult.trainer + "," + aResult.population + "," + aResult.boostFactor + "," +
aResult.accuracy + "," + aResult.timeSeconds);
        }
    }
}
```

To support the power factor being passed into the ArcX4 algorithm the following lines were changed in the indicated files

BoostingCreditClassifier.java Line 17 and following :

```
    public BoostingCreditClassifier(UserDataset ds, ) {

        this(BoostingCreditClassifier.class.getSimpleName(),
                ds, new UserInstanceBuilder(false), 4.0f);

    }

    public BoostingCreditClassifier(String name, UserDataset ds, UserInstanceBuilder instanceBuilder, float
boostFactor) {
        this(name, instanceBuilder, instanceBuilder.createTrainingSet(ds), boostFactor);
    }

    public BoostingCreditClassifier(String name,
            UserInstanceBuilder instanceBuilder, TrainingSet tSet, float boostFactor) {

        super(name, tSet, boostFactor);

        this.instanceBuilder = instanceBuilder;
    }
```

BoostingARCX4Classifier.java Line 19 and following :

```
    private float boostFactor = 4.0f;

    public BoostingARCX4Classifier(String name, TrainingSet tSet, float boost) {
        super(name);
        this.originalTSet = tSet;
        if (boost > 1.0) {
         boostFactor = boost;
        }

    }
```

Line 127 and following:

```
        // update weights
        double sum = 0.0;
        for( int i = 0; i < n; i++) {
            sum += ( 1.0 + Math.pow(m[i], boostFactor) );
        }

        for( int i = 0; i < n; i++) {
            w[i] = ( 1.0 + Math.pow(m[i], boostFactor) ) / sum;
        }
```

FUTURE WORK

In the following document, http://www.cs.princeton.edu/~schapire/papers/FilterBoost_paper.pdf, reference is made to a Boosting technique for large datasets. It represents an improvement on the basic method of AdaBoost and could overcome some of the initial problems that caused me to avoid AdaBoost's overfitting issues.