Project: 2

CS 205. Artificial Intelligence

ChenYang Yu
ID 862052273
cyu059@ucr.edu

Instructor: Dr Eamonn Keogh

22-March-2018

In completing this homework, I consulted...

. The slide from class

All the important code is original. Unimportant subroutines that are not completely original are...

. c++ Read file
https://ideone.com/T8qQGY

Two Evaluation

Achieve

Since the nearest neighbor is sensitive to features. We aim to provide the most related feature through three search algorithm. Forward selection, backward elimination, and the greedy algorithm. We also implement pruning to make the search fast.

Function explanation

`float validation_alpha(vector<int> &feature){` : Validation function. for any given set of feature, we use leave-one-out-cross-validation to calculate the correctness.

Pseudo code:

```
for(data_i in all instance){
    for (data_j != data_i in all instance){
        for (every feature k){
            calculate the distance between data_i and data_j
        }
        depends on the label, update the nearest distance for both label
    }
    compare distance from label_1 and label_2, we may decide which label is data_i
    compare decision and the real label of data_1, to increase _correctness
}
the accuracy is the total correctness count divided by the total number we make decision
```

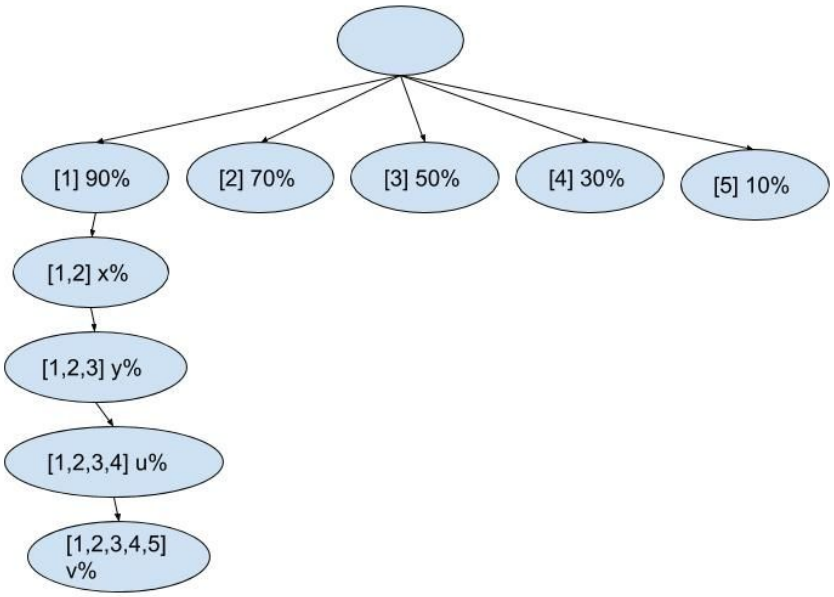
Apply pruning

`float validation_alpha(vector<int> &feature, float current_best){` : Other than correctness, we accumulate the wrong number and compare with the current_best of current layer. If the number of wrong decision if too big to make this set of feature become best decision, we simply return zero rather than finish the whole decision.

```
if(dist_1_output==dist_2_output){
    cout<<"equal"<<endl;
    correct++;
}else if(dist_1_output > dist_2_output){
    // we think it is 2
    if(table[i][0]==2)
        correct++;
    else
        wrong++;
}else if(dist_1_output < dist_2_output){
    // we think it is 1
    if(table[i][0]==1)
        correct++;
    else
        wrong++;
}
if(wrong/table.size() > (1-current_best)){
    return 0;
}
```

Search Algorithm

1. Forward (with pruning)
   We obtain a table to record the best decision (the highest accuracy) and the features we choose for each layer. Starts from a empty list, we add feature one by one and decide which feature is the best. We keep this decision and pick another non-repeat feature in next layer, and so on. Until we reach the bottom, where we consider all features and pick the best decision.
2. Backward
   Similar with forward, but we start from a list contain all feature. Eliminate feature one by one and pick the best decision at each layer. Until we reach the bottom, which has only one feature left.
3. Greedy algorithm: although the forward and backward search before is already greedy algorithm (we choose to add/ eliminate one feature to achieve the best accuracy instead of brute force search all possibilities). We expect to get a faster searching by not re-decide the add/eliminate feature for next layer. Instead, we use the accuracy where we obtain at the first layer. Take a data set with 5 features for example:



   We can expect this approach cannot be better that other two algorithm in accuracy. But the execution speed should be much faster since we only have to measure each feature once. On the other hand, our greedy algorithm can be as good as other two algorithm if the feature are independent from each other.

Result

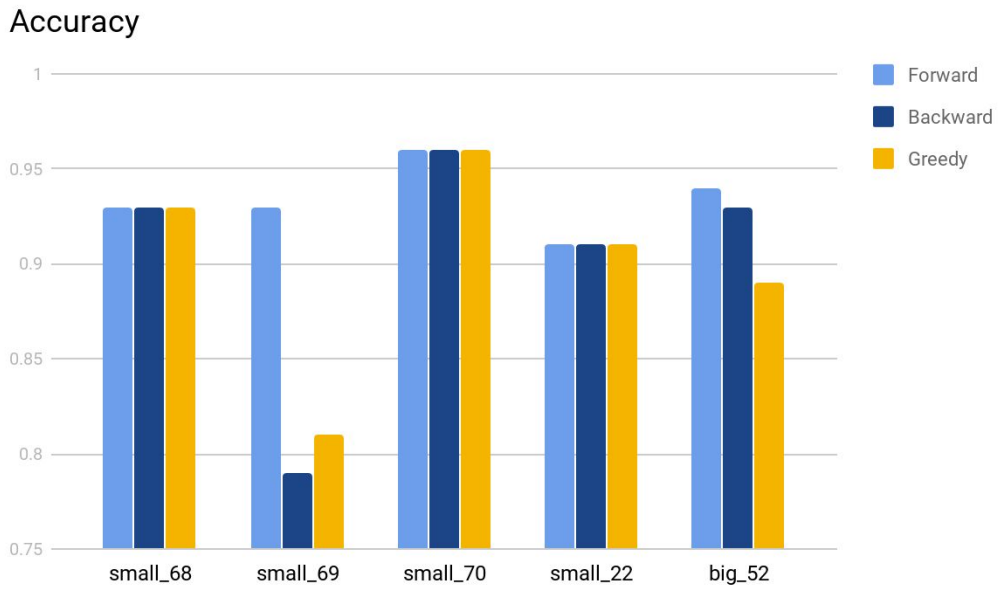
data set	Forward/ pruning	Backward	Greedy Algorithm
small_68	Rate=0.93, feature=6 9	Optimal Rate=0.93, feature=1 10 7 5 8 4 2 3 9	Optimal Rate=0.93, feature=6

small_69	Optimal Rate=0.93, feature=4 3	Optimal Rate=0.79, feature=6 4	Optimal Rate=0.81, feature=4 8 3
small_70	Optimal Rate=0.96, feature=1 8	Optimal Rate=0.96, feature=5 3 2 7 4 6 10 9	Optimal Rate=0.96, feature=1 8
small_22	Optimal Rate=0.91, feature=6 10 3	Optimal Rate=0.91, feature=2 4 7 8 1 9 5 3	Optimal Rate=0.91, feature=10 6 7
big_52	Optimal Rate=0.94, feature=15 49 42	Optimal Rate=0.93, feature=9 26 24 5 14 38 45 43 17 32 15 18 6 10 8 42 41 13 30 12	Optimal Rate=0.89, feature=15 24

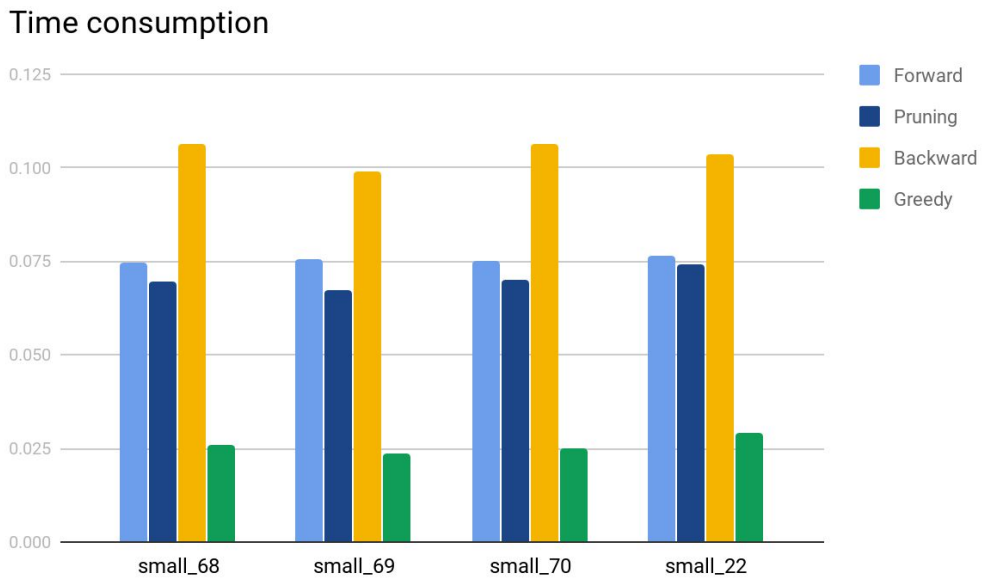
Fig1. The accuracy and the features we select

data set	Forward	Backward	Prune Algorithm	Greedy Algorithm
small_68	0.074646 sec	0.106576 sec	0.069667 sec	0.026029 sec
small_69	0.075684 sec	0.098859 sec	0.067389 sec	0.023692 sec
small_70	0.075247 sec	0.106444 sec	0.069971 sec	0.0248 sec
small_22	0.076577 sec	0.103447 sec	0.074273 sec	0.029059 sec
big_52	5.82681 sec	11.116 sec	4.88047 sec	0.384093 sec

Fig2. The time consumption



Since pruning does not affect the accuracy, we combine their accuracy and compare with other two algorithm. As we can see, forward search achieves the best accuracy. It is fit our prediction that greedy algorithm is hard to out performance.



When it comes to time consumption, the pruning algorithm is slightly better than other two algorithm. The greedy algorithm achieve the best performance as we expect. Compare with forward algorithm, the greedy algorithm saves almost 70% of time.

Testing environment

g++ 4.2.1 (Apple LLVM version 9.0.0) under OSX 10.13.3

Trace back

(A) Small_22:

```
test 1 test 2 test 3 test 4 test 5 test 6 test 7 test 8 test 9 test 10

layer[1] end: winner rate=0.8, feature: 6
test 1 test 2 test 3 test 4 test 5 test 7 test 8 test 9 test 10

layer[2] end: winner rate=0.91, feature: 6 10
test 1 test 2 test 3 test 4 test 5 test 7 test 8 test 9

layer[3] end: winner rate=0.91, feature: 6 10 3
test 1 test 2 test 4 test 5 test 7 test 8 test 9

layer[4] end: winner rate=0.9, feature: 6 10 3 7
```

The first three layer: choose feature 6, 10, 3 separately

```
test 5 test 8 test 9

layer[8] end: winner rate=0.77, feature: 6 10 3 7 2 4 1 8
test 5 test 9

layer[9] end: winner rate=0.72, feature: 6 10 3 7 2 4 1 8 9
test 5

layer[10] end: winner rate=0.66, feature: 6 10 3 7 2 4 1 8 9 5



Optimal Rate=0.91, feature=6 10 3
0.075991 sec
```

The last three layer, choose 8, 9, 5 separately. Search all decision and get the final feature sets and the accuracy.

(B) Big_52:

```
test 1 test 2 test 3 test 4 test 5 test 6 test 7 test 8 test 9 test 10 test 11 test 12 test 13 test
14 test 15 test 16 test 17 test 18 test 19 test 20 test 21 test 22 test 23 test 24 test 25 test 26 t
est 27 test 28 test 29 test 30 test 31 test 32 test 33 test 34 test 35 test 36 test 37 test 38 test
39 test 40 test 41 test 42 test 43 test 44 test 45 test 46 test 47 test 48 test 49 test 50

layer[1] end: winner rate=0.86, feature: 15
test 1 test 2 test 3 test 4 test 5 test 6 test 7 test 8 test 9 test 10 test 11 test 12 test 13 test
14 test 16 test 17 test 18 test 19 test 20 test 21 test 22 test 23 test 24 test 25 test 26 test 27 t
est 28 test 29 test 30 test 31 test 32 test 33 test 34 test 35 test 36 test 37 test 38 test 39 test
40 test 41 test 42 test 43 test 44 test 45 test 46 test 47 test 48 test 49 test 50

layer[2] end: winner rate=0.93, feature: 15 49
test 1 test 2 test 3 test 4 test 5 test 6 test 7 test 8 test 9 test 10 test 11 test 12 test 13 test
14 test 16 test 17 test 18 test 19 test 20 test 21 test 22 test 23 test 24 test 25 test 26 test 27 t
est 28 test 29 test 30 test 31 test 32 test 33 test 34 test 35 test 36 test 37 test 38 test 39 test
40 test 41 test 42 test 43 test 44 test 45 test 46 test 47 test 48 test 50

layer[3] end: winner rate=0.94, feature: 15 49 42
```

The first three layer: choose feature 15, 49, 42 separately

```
test 9 test 38 test 44

layer[48] end: winner rate=0.82, feature: 15 49 42 5 6 3 16 12 10 24 8 45 28 23 37 2 21 4 41 22 11 1
8 34 19 36 14 50 20 31 1 40 26 35 32 47 48 33 39 46 27 17 25 13 29 43 30 7 38
test 9 test 44

layer[49] end: winner rate=0.83, feature: 15 49 42 5 6 3 16 12 10 24 8 45 28 23 37 2 21 4 41 22 11 1
8 34 19 36 14 50 20 31 1 40 26 35 32 47 48 33 39 46 27 17 25 13 29 43 30 7 38 44
test 9

layer[50] end: winner rate=0.8, feature: 15 49 42 5 6 3 16 12 10 24 8 45 28 23 37 2 21 4 41 22 11 18
 34 19 36 14 50 20 31 1 40 26 35 32 47 48 33 39 46 27 17 25 13 29 43 30 7 38 44 9



Optimal Rate=0.94, feature=15 49 42
5.73536 sec
```

The last three layer, choose 38, 44, 9 separately. Search all decision and get the final feature sets and the accuracy.

Conclusion:

Among first two algorithm, backward elimination not only worse in accuracy, slower in execution time, also the final decision could be a large number of feature. According to occam razor theory, we should avoid using these large amount of features even if some of them are at the same accuracy. Since it is more possible an overfit result. The greedy algorithm we implement cannot not achieve as good accuracy as forward selection. However, we can see a significant improvement of speed. It is useful when some compromise has to be made, depending on the application. Sometime we may rather have a less optimal solution in a short time interval.

Code:

Validation function

```
for(int i=0;i<table.size();i++){
    //i is the validation data
    float dist_1_output=100000;
    float dist_2_output=100000;
    for(int j=0;j<table.size();j++){
        // go through all other data (training data)
        //j is the data used for calculating dist
        if(j==i)
            continue;
        // go through feature
        float dist_1=0, dist_2=0;
        for(int k=0;k<feature.size();k++){
            int target = feature[k];
            if(table[j][0]==1){
                dist_1+=(table[j][target]-table[i][target])*(table[j][target]-table[i][target]);
            }else if(table[j][0]==2){
                dist_2+=(table[j][target]-table[i][target])*(table[j][target]-table[i][target]);
            }else{
                cout<<"illegal label1"<<endl;
            }
        }
        // keep update the nearest distance for both label
        if(table[j][0]==1 && dist_1<=dist_1_output){
            dist_1_output=dist_1;
        }else if(table[j][0]==2 && dist_2<=dist_2_output){
            dist_2_output=dist_2;
        }else{
            //cout<<"illegal label2"<<endl;
        }

    }// j loop
    // now we hv distance for both lable, time to decide which lable is data_i
    if(dist_1_output==dist_2_output){
        cout<<"equal"<<endl;
        correct++;
    }else if(dist_1_output > dist_2_output){
        // we think it is 2
        if(table[i][0]==2)
            correct++;
    }else if(dist_1_output < dist_2_output){
        // we think it is 1
        if(table[i][0]==1)
            correct++;
    }
}// i loop
```

Forwarding

```
// candidate is sequence vecotr [1,2,3...9,10], choose feature from this one by one
// pop out the sure feature, so when we finish when candidate.size()
while(candidate.size()>0){
    tmp.clear();
    if(i>=1){
        // boundry condition, copy feature from previous layer
        tmp.clear();
        if(feature_table.size()>0)
            tmp.assign(feature_table[i-1].begin(),feature_table[i-1].end());
    }
    // choose from candidate, add into tmp and test
    // the best result is store in accuracy_level and accuracy_candicate
    for(int s=0;s<candidate.size();s++){
        cout<<"push "<<candidate[s]<<endl;
        tmp.push_back(candidate[s]);
        float tmp_acc = validation(tmp);
        if(tmp_acc > accuracy_level[i]){
            accuracy_level[i] = tmp_acc;
            accuracy_candicate.assign(tmp.begin(),tmp.end());
        }
        tmp.pop_back();
    }
    // after we have a sure decision, add accuracy_candicate into feature_table[i]
    // represent the feature set for layer i
    cout<<"layer["<<i+1<<"] end: winner rate="<<accuracy_level[i]<<", feature: ";
    for(int m=0;m<accuracy_candicate.size();m++)
        cout<<accuracy_candicate[m]<<" ";
    cout<<endl;
    feature_table.push_back(accuracy_candicate);
    // delete this feature
    candidate.erase(find(candidate.begin(),candidate.end(),accuracy_candicate.back()));
    i++;
}
```

Greedy

```
vector<pair<int,float> >greedy_v;
map<int, float> greedy_map;

// this is for layer_1, calculate accuracy for each feature
// and insert into greedy_map
for(int s=0;s<candidate.size();s++){
    cout<<"push "<<candidate[s]<<endl;
    tmp.push_back(candidate[s]);
    float tmp_acc = validation(tmp);
    greedy_map.insert(make_pair(candidate[s],tmp_acc));
    tmp.pop_back();
}

// the greedy_map is <feature, accuracy>
// move is into greedy_v (sorting by the accuracy value)
for(auto it=greedy_map.begin();it!=greedy_map.end();it++){
    cout<<"("<<it->first<<", "<<it->second<<")";
    greedy_v.push_back(make_pair(it->first,it->second));
}
cout<<endl;
sort(greedy_v.begin(),greedy_v.end(),cmp);

// tmp used for calculate accuracy for each layer
// push featurn from the back of greedy_v (the highest left feature)
tmp.clear();
tmp.push_back(greedy_v.rbegin()->first);
feature_table.push_back(tmp);
accuracy_level[0] = greedy_v.rbegin()->second;
greedy_v.pop_back();
cout<<"layer["<<0<<"] end: winner rate="<<accuracy_level[0]<<", feature: ";
for(int m=0;m<feature_table[0].size();m++)
    cout<<feature_table[0][m]<<" ";
cout<<endl;

// finish the rest layer, record the accuracy and feature in accuracy_level and feature_table
int count = greedy_v.size();
for(int i=0;i<count;i++){
    tmp.push_back(greedy_v.rbegin()->first);
    float tmp_acc = validation(tmp);
    feature_table.push_back(tmp);
    accuracy_level[i+1] = tmp_acc;
    greedy_v.pop_back();
    cout<<"layer["<<i+1<<"] end: winner rate="<<accuracy_level[i+1]<<", feature: ";
    for(int m=0;m<feature_table[i+1].size();m++)
        cout<<feature_table[i+1][m]<<" ";
    cout<<endl;
}
```