Homework 3

Reet Barik, WSU ID: 11630142
CptS 570 Machine Learning

November 13, 2018

Exercise 1. We need to perform statistical tests to compare the performance of two learning algorithms on a given learning task. Please read the following paper and briefly summarize the key ideas as you understood:
Thomas G. Dietterich: Approximate Statistical Test For Comparing Supervised Classification Learning Algorithms. Neural Computation 10(7): 1895-1923 (1998) http://sci2s.ugr.es/keel/pdf/algorithm/articulo/dietterich1998.pdf

Answer:

Introduction:
There is a hierarchy of statistical questions that need to be asked while tackling a machine learning challenge. The root of the hierarchy asks whether the problem pertains to a single application domain or multiple domains. While working in a single domain setting, the question that is asked is whether we need to come up with a classifier or a learning algorithm. A classifier is a function that given an example, will assign it a class label. A learning algorithm on the other hand, given a set of input examples and their class labels construct a classifier. The next level in the hierarchy distinguishes between two fundamental tasks. estimating accuracy and choosing between classifiers or algorithms. The last level of the hierarchy is concerned with the amount of data available. The focus of this paper is on those problems which deal with a single domain, analyses algorithms to choose from a space of algorithms and deals with a small sample of data.
Keeping the question setting described above in mind, this paper describes 5 statistical tests: McNemar's Test, a test for the difference of two proportions, the resampled t-test, the cross-validated t-test and a new test called the 5x2cv test.

Five Statistical Tests:

1. McNemar's Test:
A sample S of data is divided into training set R and test set T. Algorithms A and B are trained on R to yield two classifiers $\hat{f_A}$ and $\hat{f_B}$ which are then tested on test data. A contingency table is constructed with the values n_{00}, n_{01}, n_{10}, and n_{11}.
The two algorithms will have the same error rate under the null hypothesis which implies $n_{01} = n_{10}$. The McNemar's Test is based on a χ^2 test with a degree of freedom as 1. So a

continuous correction term of -1 is added to the numerator of the following quantity:

$$\frac{(|n_{01} - n_{10}| - 1)^2}{n_{01} + n_{10}}$$

If the above probability that the above quantity is greater than $\chi^2_{1,0.95}$ is less than 0.05 in favor of the hypothesis that the two algorithms have different performance when trained on a particular training set R, then the Null Hypothesis is rejected.
One of the shortcomings of this test is that it does not directly measure variability due to the choice of the training set or the internal randomness of the learning algorithm. The other one is that it compares the performance only on a set of the training set of size $|R|$, which is sufficiently smaller than the whole set $|S|$.

2. A test for the difference of two proportions:
This test is used to measure the difference between the error rates of the learning algorithm A and B. If probability of misclassification by algorithm A is P_A and that of algorithm B is P_B, then under the Null Hypothesis, the mean will be zero and the standard error will be given by:

$$\sqrt{\frac{2p(1-p)}{n}}$$

where p = (P_A + P_B) / 2
If the probabilities of error rates are considered to be independent, then the distribution can be viewed as a normal distribution. The following metric is hence obtained:

$$z = \frac{P_A - P_B}{\sqrt{2p(1-p)/n}}$$

The disadvantage of this test is that P_A and P_B are not independent given that they are obtained from the same data set. Also, like the McNemar's Test, it does not measure the variance due to the choice of training set or internal randomness of the learning algorithm.

3. The re-sampled paired t-test:
A series of 30 trials is conducted. In each, the dataset S is randomly divided into a training set R of a specified size and a test set T. If we assume that the 30 differences $p^{(i)} = p_A^{(i)} - p_B^{(i)}$ were drawn independently from a normal distribution, students t test can be applied to compute statistic:

$$t = \frac{\bar{p}\sqrt{n}}{\sqrt{\frac{\sum_{i=1}^{n}(p^i - \bar{p})^2}{n-1}}}$$

Under the Null Hypothesis, this statistic has a t-distribution with n - 1 degrees of freedom. For 30 trials, the Null Hypothesis can be rejected if $|t| > t_{29,0.975} = 2.04523$.
The potential drawbacks of this approach are that the probabilities might not be independent as the test sets in the trials overlap.

4. K fold cross validated paired t test:
This is similar to the re-sampled paired t-test except that S here is divided into k disjoint sets of equal size and then k trials are conducted. Here, each test set is independent of each other unlike in the previous tests but the overlap between training set still persists.

5. The 5x2cv paired t-test:
The statistic described in the previous one can sometimes become too large. Hence, the numerator is replaced with the observed difference from a single fold of the k fold cross validation. In 5x2cv, 5 replications of 2 fold cross validation is performed. The statistic is obtained as follows:

$$\tilde{t} = \frac{p_1^{(1)}}{\sqrt{\frac{1}{5}\sum_{i=1}^{5} s_i^2}}$$

Under the Null Hypothesis, the statistic has approximately a t distribution and 5 degrees of freedom.

Simulation Experiment Design:
The purpose of the simulation was to measure the Type I error of the algorithms. (Type I error occurs when the Null Hypothesis is true).
Simulating the behavior of learning algorithms: The probability that the classifier will misclassify for learning algorithm A and B is $\epsilon_A(x)$ and $\epsilon_B(x)$. Maximizing variation resulting from choice of test sets and training sets. Target error rate and two distinct values are chosen for the population. The reverse of the values of $\epsilon_A(x)$ and $\epsilon_B(x)$ were chosen for the second half. Because of sampling with replacement, the population size doesn't matter and it is evenly distributed between two training points. The overall error rate ϵ and total variance for each algorithm is same, but different for a given sample. This is taken care by random choice of data sets.
Details of the simulation of each statistical test: Each simulation is divided into a series of 1000 trials in each of which a data set S of size 300 is randomly sampled with replacement and then analyzed using the 5 tests described.

Results:
The probability of Type I error exceeds the target value of 0.05 in case of distance of proportions test and re-sampled t-test while the other tests have acceptable Type I error probabilities.
The probability of Type I error is much higher in case of re-sampled t-test because the randomly drawn data set S is likely to contain an imbalance of points from the first half of the population as compared to the second half. The difference of proportions test also suffers from the same problem. Due to poor performance of the re-sampled t-test it is excluded from any further discussion.

Experiments on Realistic Data:

Methods: To evaluate Type I error rates of the 4 tests with real learning algorithms, we need to find two learning algorithms that have the same kind of performance on the training set of given size. C4.5 Release 1, first Nearest Neighbor algorithm were chosen for this purpose. The hard problems chosen to test this on are: EXP6 problem by Kong, the Letter Recognition Dataset and Pima Indians Diabetes task. The learning algorithms are suitably tampered with so as to give similar performances.
Type I error results: The 10 fold cross-validated t-test has Type I error above 0.05 while the others have acceptable Type I error.
Power Measurements: The power of a test is the probability that it will reject the null hypothesis when the null hypothesis is false. Of the tests with acceptable Type I error rates, the 5x2cv is adjudged to be the most powerful.

Discussions:

5x2cv is observed to be the most powerful among all the tests and it is also the most safe to use since it takes into account the choice of training set as well as the test set.

Exercise 2. Please read the following paper and briefly summarize the key ideas as you understood:
Thomas G. Dietterich (1995) Overfitting and under-computing in machine learning. Computing Surveys, 27(3), 326-327. http://www.cs.orst.edu/ tgd/publications/cs95.ps.gz

Answer:

One of the main problems in Machine Learning revolves around Supervised Learning which is learning from labeled data. The function to be learned by the system is generally represented as a set of rules, a decision tree, etc.

The learning algorithms search for a function in the hypothesis class that fits the training data. This search is usually done by defining an objective function which is then optimized. Since in almost all cases the computational problem of optimizing is an NP-hard problem, heuristic algorithms like Gradient Descent and Greedy Search has enjoyed huge success.

Since the heuristic algorithms are not optimal, a separate branch of research has come forward which tries to come up with algorithms that can search the hypothesis class better. But ironically, it was observed that these algorithms performed even worse than heuristic ones on real datasets.

The main problem with the machine learning algorithm arises from the fact that it trained on the training set and then used to predict on new data points. As a result, if the training data is given too much importance, the algorithm learns the peculiarities by fitting over the noise and hence resulting in a loss of generality. This is called 'overfitting'.

This means that the objective function being optimized is not proper. Attempts to fix this problem has been to enhance the objective function with various penalty terms like regularization, generalized cross-validation etc. This has been empirically known to remove the discrepancy described in the previous paragraph.

However, it can be observed that the greedy algorithms, though sub-optimal as far as the above mentioned objective functions are concerned, can be optimal for a different set of objectives. From this, the author comes to the conclusion that "By "undercomputing" we avoid "overfitting".

In the end, the author predicts that in the coming decade, research will bring forward interactions which are not dissimilar to how there is a healthy dose of interaction between computaional issues and statistics.

Exercise 3. Please read the following paper and briefly summarize the key ideas as you understood:
Thomas G. Dietterich (2000). Ensemble Methods in Machine Learning. J. Kittler and F. Roli (Ed.) First International Workshop on Multiple Classifier Systems, Lecture Notes in Computer Science (pp. 1-15). New York: Springer Verlag. http://web.engr.oregonstate.edu/ tgd/ publications/mcs-ensembles.pdf

Answer:

Introduction:
An ensemble of classifiers is a set of classifiers whose individual decisions are combined in some way to classify new examples. Constructing good ensembles has been an active area of research in supervised learning.
A necessary and sufficient condition for an ensemble classifier to perform better than nay of its individual member classifiers is if they are accurate and diverse. If a classifier has performance which is better than just random guessing, it is deemed an accurate classifier. If two classifiers makes errors on different set of examples, then they are termed as diverse.
It is possible to create a good ensemble for the following reasons:
Statistical: The averaging of votes by the classifiers reduces the risk of choosing a wrong classifier.
Computational: Running local searches from different staring points may provide a better approximation of the original function whereas there is a chance of getting stuck in local optima while using the individual member classifiers.
Representational: the space of representable functions can be expanded by forming weighted sums of hypotheses drawn from the Hypothesis space H.

Methods for constructing ensembles:
Bayesian Voting:
Manipulating the training examples: Here, the learning algorithm is run for a number of iterations on different subsets of the data. This is all the more useful when the algorithms are highly susceptible to even minor changes in the training data. This is how diversity is injected into the learning process. Manipulation can be done in the form of bagging, or constructing the training sets by leaving out disjoint subsets, or by using the ADABOOST algorithm.
Manipulating the input features: Here, different member classifiers are run on different sets of input features. It should be noted that this technique works well when the input features

are highly redundant.
Manipulating the output targets: A technique called error-correcting output coding is used when the number of classes K is large. This involves partitioning the classes into two broader classes by relabeling the outputs and learning L number of classifiers by repeating this process L times, each time partitioning K differently.
Injecting randomness: Algorithm is applied to the same training examples but with different initial weights. It is easy to inject randomness in decision tree algorithm. Bootstrap sampling can be combined with this method. Markov chain monte carlo methods for constructing Bayesian ensembles also work by incorporating injecting randomness into the learning process, though each hypothesis receives a vote proportional to its posterior probability.

Comparing Different Ensemble Methods:
ADABOOST often gives the best result according to Dietterichs study. Similar results are obtained from bagging and randomized tree. But randomization may be preferred because it creates diversity in all conditions.
The statistical problem and to some extent the computational problem is addressed by bagging and randomization. They don't address the representational problem directly which is not the case with boosting. It does so by eliminating the residual errors and thus optimizing the weighted vote which directly aims at the representation problem. Hence, ADABOOST performs better in low-noise setting whereas bagging and randomization perform well irrespective of noise.
Due to the stage wise nature of ADABOOST, it constructs a new hypothesis at each iteration by re-weighing the training examples and thus helps in reducing overfitting.

Exercise 4. Please read the first five sections of the following paper and briefly summarize the key ideas as you understood:
Jerome Friedman (2001).Greedy function approximation: A gradient boosting machine. The Annals of Statistics, 29(5), pp 11891232. https://statweb.stanford.edu/ jhf/ftp/trebst.pdf

Answer:

This paper can be summarized as follows:

1. Function Estimation:

   - We have n training examples $\{y_i, x_i\}_i^N$ where x and y are the inputs and labels respectively. The prediction problem's goal is be to find a $F(x)$ that approximates the ideal prediction function $F^*(x)$, that minimizes the loss function $L(y, F(x))$ given a joint distribution over all of y given x.
   - A restricted finite set of Parameters P is used to identify each class member. In this paper the author focuses on additive form:
     $F(x, const_m, p_m 1^M) = \sum m = 1^M const_m h(x, p_m)$.

- h is a function that each parameter satisfies along with the input x. The function that needs to be estimated is given by the addition of all such parameters and such additive functions form the basis for multiple approximation method.
- Numerical Optimization: Converting a optimization of function space to optimization of parameterized space, as discussed above, requires $F(x, P)$ and its respective L be solved using some numerical optimization method, $P^* = argmin_P \phi(P) = E_{y,x} L(y, F(x, P))$, i.e. expectation of loss over label and prediction of parameterized model given data and labels. For this we start with an initial guess p_0 and after each successive "boost" or step we increment the successive p_m. There are multiple ways to compute successive p_m's:
- Steepest-decent: This algorithm works by taking the largest steps in the steepest downward direction in each iteration. The gradient, g_m, is obtained by taking the partial derivative of $\phi(P)$ where P is the solution to the parameters in the previous iteration. In the current iteration we find the $argmin_p \phi(P_{m-1} - pg_m)$.

2. Numerical optimization of function space:
   - Here, $F(x)$ itself is considered to be a parameter and optimize $\phi(F)$.
   - As before the optimization function would be $F^*(x) = \sum_{m=0}^{M} f_m(x)$ where f_0 will be the initial guess and for steepest decent $f_m(x) = -p_m g_m(x)$
   - The multiplier is computed as $p_m = argmin_p Ey, xL(y, F_{m-1}(x) - pg_m(x))$

3. Finite Data:
   - It is surmised that the nonparametric way won't work for finite data as some key assumptions are voided and $E_y[.|x]$ cannot be estimated accurately without the help of neighboring x's.
   - The generic h function from the above additive method is used along with the function space minimization, i.e, minimize the cost and p_m over the loss function that we can actually estimate over the finite data and use that in $F_m(x) = F_{m-1}(x) + const_m h(x, p_m)$
   - This is noticeably similar to the gradient decent where we take the highest decent toward $F^*(x)$ in the direction given by h and the approximator as $F_{m-1}(x)$. The paper also shows that we can construct g_m from this formulation.
   - Quoting the paper, "basically, instead of obtaining the solution under a smoothness constraint, the constraint is applied to the constrained solution by fitting $h(x, p)$ to the 'pseudo responses'" of the negative gradient.
   - The algorithm here is used to do gradient boosting, where step 4 finds the parameter for the current stage and step 5 uses the minimized parameter to estimate loss.

4. Applications: Additive modelling

- Least squares regression: Here the loss function is given by $L(y, F) = (y - F)^2/2$ (squared error loss). This makes the initial step to find the gradient, $Y_i = y_i - F_{m-1}(x_i)$ instead of the partial derivative of the loss function. Also, step 5 reduces to $p_m = const_m$ where $const_m$ is obtained from the minimizing in step 4.
- Least absolute deviation regression: Here as $L(y, F) = |y - F|$ (least deviation), and step 5 is the median of weights. The weight is the parameter estimation h function.
- Regression trees: Each regression tree already has an additive form which is determined by the disjoint regions over the joint space of the predictor, x. Each region's boundary is defined by parameters or coefficients, b_j. So $h(x) = b_j$ and we need to only change the additive step in the algorithm to take this into account. Where each h(x) only has a sign {1,-1} if it belongs to a region characterized by b_j.
- M-Regression: This algorithm tries to reduce outliers and hence maintain high efficiency for normally distributed errors. For this it uses the Huber loss function.
- 2-class logistic regression: Here the loss function is negative log-likelihood where F(x) is the log of the ratio of y=1 and y=-1 given x. The paper also shows how the pseudo-response and line functions are found. The final approximation $F_M(x)$ is related to log odds through F(x). This can be inverted to yield probability estimates as shown in the paper.
  In the loss function for the two class logistic regression problem at the m-th iteration, if $y_i F_{m-1}(x_i)$ is very large, then the example has no dependence on the loss function. So all observations with large $y_i F_{m-1}(x_i)$ can be deleted from all computations, this is called Influence Trimming.
- Multiclass logistic regression and classification: The paper shows how we can convert the the gradient boosting into a K-class problem, where loss $L(\{y_k, f_k(x)\}1^k) = -\sum k = 1^K y_k log p_k(x)$ and can be expressed in terms as exp.

5. Regularization:

- Constraining the fitting procedure is done to prevent overfitting. The value of M, which is the number of components is used to regulate the degree to which the expected loss can be minimized.
- But in contradiction, regularization through shrinkage provides superior results to that obtained by restricting the number of components. This can be done by replacing step 6 with this: $F_m(x) = F_{m-1}(x) + v p_m h(x, p_m)$ $0 < v \leq 1$. Each update is simply scaled by the learning rate, v.
- In case of gradient boosting means both v and M control the degree-of-fit. As decreasing v increases M, their effect has to be minimized jointly.
- The paper proved that with tree boost as the base learner, small values of v, result in better performance and in large values overfitting is observed with the performance reaching an optimum at M and diminishing beyond that.

- Although these results are shown for one method, they are fairly universal. The best value of v depends on a large M. Empirical evidence indicates that global shrinkage provides better improvement over no shrinkage, far from the effect of incremental shrinkage which is under investigation.

Exercise 5. Please read the following paper and briefly summarize the key ideas as you understood:
Tianqi Chen, Carlos Guestrin: XGBoost: A Scalable Tree Boosting System. KDD 2016. https://www.kdd.org/kdd2016/papers/files/rfp0697-chenAemb.pdf

Answer:

Abstract:
The paper talks about XGBoost which is a scalable end-to-end tree boosting system. This has been generally observed to perform well for sparse data and uses insights on cache access patterns, data compression and sharding to make it more scalable than existing systems all the while using fewer resources.

Introduction:
Gradient tree boosting has been shown to provide state of the art results for a variety of machine learning challenges among the many machine learning methods widely used. For example, LambdaMART is used for ranking problems with great success. It has also been proven good enough to be the default choice in ensemble methods.
This paper describes XGBoost, a scalable end-to-end tree boosting system that has found much success so much so that an overwhelming majority of winning contestants in various reputed Machine Learning competitions have used it exclusively or incorporated it in their solutions to the challenges assigned.
The winning solutions include XGBoost being applied to challenges such as store sales prediction, high energy physics event classification, web text classification, customer behavior prediction, motion detection, ad click through rate prediction, malware classification, product categorization, hazard risk prediction, massive online course dropout rate prediction etc. This shows the versatility and scalability of the system.
XGBoost uses a novel tree learning algorithm which is particularly useful for sparse data. It also uses a theoretically justified weighted quantile sketch procedure and parallel/distributed computing to make model exploration faster. It combines all these techniques to provide an end-to-end scalable solution.

Tree Boosting in a nutshell:
Regularized Learning Objective: Here, the objective function that has to be minimized is regularized by adding a second term to the loss function which penalizes the complexity of the model. This helps smooth the objective function and hence prevent overfitting. Setting the regularizing parameter to zero will result in a traditional gradient tree boosting.
Gradient Tree Boosting: The ensemble model takes in functions as parameters. Hence it cannot be optimized by algorithms that are generally applicable on Euclidean space. This is

why the model is trained in an additive manner where that function is chosen greedily which most improves the model. An easy to optimize function is arrived at which gives an impurity score for decision trees except that it works for a wider range of objective functions. Since it is impossible to evaluate all tree structures, a greedy function to compute the reduction in loss after the split is derived which can be then used in practice.
Shrinkage and Column Subsampling: Apart from the regularized objective function, two other techniques are used to prevent overfitting. The first is called Shrinkage, where the learned weights are scaled down by some factor to reduce the influence of individual trees leaving room for future trees to improve the model. The second one is called Column Subsampling, a technique used in random forests which has proven more effective than row subsampling in preventing overfitting. It also speeds up the computation of the parallel algorithms.

Split finding algorithms:
Basic Exact Greedy algorithm: All possible splits are enumerated over and the best one is chosen. The computation is made efficient by sorting the data according to feature values.
Approximate algorithm: These are used in cases where the data cannot fit into the memory or if the setting is of a distributed nature. Here, candidate splitting points are proposed and the data is then binned along these split points. These are evaluated and the best one is then chosen based on the aggregated statistics. The global variant of this algorithm uses the same proposal for split finding at all levels whereas the local variant re-proposes after each split.
Weighted Quantile Search: This is used when dealing with weighted datasets that has no existing quantile sketch. A data structure is proposed that supports merge and prune operations that can provably give a theoretical guarantee a certain accuracy level.
Sparsity aware Split-finding: XGBoost handles sparse data by adding a default direction in each node on encountering a missing value in the sparse matrix. The optimal default direction is learned from the data. It has been observed that the sparsity awareness makes the algorithm run 50 times faster than its naive version.

System Design:
Column Block for Parallel Learning: To reduce the cost of sorting the data, it is stored in in-memory units called 'block's which are in turn stored in Compressed Column format. This layout is constructed just once and is reused throughout all iterations. This reduces the split finding step into a linear scan over sorted columns. This can then be easily parallelized. Moreover, column subsampling also becomes very convenient.
Cache-Aware Access: the new algorithm fetches gradient statistics by row index which can be classified as non-continuous memory access making it susceptible to Cache miss. To solve this, prefetching of the gradient statistic is done into an internal buffer of each thread. This changes the direct read/write dependency into a longer one and helps reduce the runtime overhead. Moreover, the cache size has also been observed to be a significant factor in improving performance.
Blocks for Out-of-core Computation: To reduce the overgead and increase the throughput of disk I/O, the following two techniques are used. Block Compression: here, each block is compressed by columns and decompressed on the fly on a separate thread while loading

on to main memory. Black sharding: This shards the data onto multiple disks in an alternative manner. A pre-fetcher thread is assigned to each disk and fetches the data into an in-memory buffer. The training thread then alternatively reads the data from each buffer.

End-to end Evaluations:
System Implementation: XGBoost is implemented as an open source package available in popular languages like Python, R, Julia etc. Moreover, the distributed XGBoost runs natively on Hadoop, MPI Sun Grid engine. More such integrations are expected in the future.
Dataset and Setup: The datasets used in the experiments described in the paper are the Allstate insurance claim dataset, the Higgs boson dataset, the Yahoo! learning to rank challenge dataset, and s the criteo terabyte click log dataset.
Classification: The classification performance of XGBoost is better than R's GBM and almost 10x faster than sci-kit learn.
Learning to Rank: XGBoost performs better than pGBRT which was the best previously published system for this task.
Out-of-core experiment: A very large dataset was used to fill up the system file cache and produce an out-of-core setting. It is seen that compression helps to speed up computation by factor ofthree, and sharding into two disks further gives 2x speedup.
Distributed experiment: A YARN cluster on EC2 with m3.2xlarge machines was set up to give a distributed setting. XGBoost runs faster than other baseline systems and scales linealry as more machines are added.

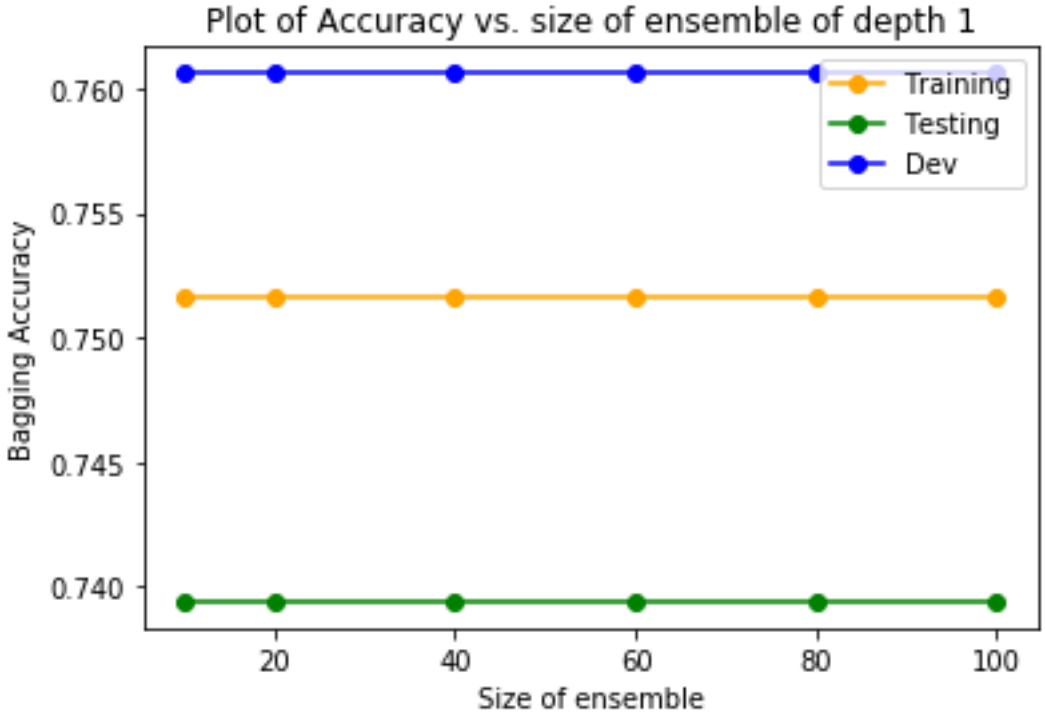
Exercise 6. Empirical analyis question. Income Classifier using Bagging and Boosting. You will use the Adult Income dataset from HW1 for this question. You can use Weka or scikitlearn software.
a. Bagging (weka.classifiers.meta.Bagging). You will use decision tree as the base supervised learner. Try trees of different depth (1, 2, 3, 5, 10) and different sizes of bag or ensemble, i.e., number of trees (10, 20, 40, 60, 80, 100). Compute the training accuracy, validation accuracy, and testing accuracy for different combinations of tree depth and number of trees; and plot them. List your observations.
b. Boosting (weka.classifiers.meta.AdaBoostM1). You will use decision tree as the base supervised learner. Try trees of different depth (1, 2, 3) and different number of boosting iterations (10, 20, 40, 60, 80, 100). Compute the training accuracy, validation accuracy, and testing accuracy for different combinations of tree depth and number of boosting iterations; and plot them. List your observations.

Answer:

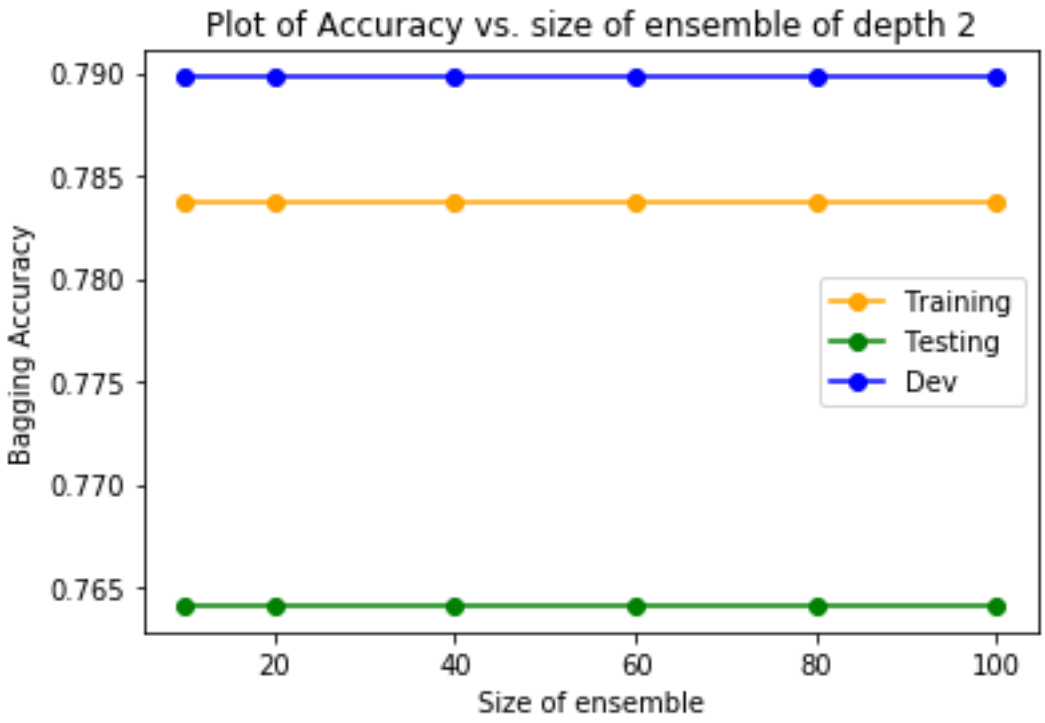
a)Sklearn Bagging:

Depth = 1, Ensemble size = 10
Training Accuracy: 0.7516, Testing Accuracy: 0.7394, Dev Accuracy: 0.7606
Depth = 1, Ensemble size = 20
Training Accuracy: 0.7516, Testing Accuracy: 0.7394, Dev Accuracy: 0.7606

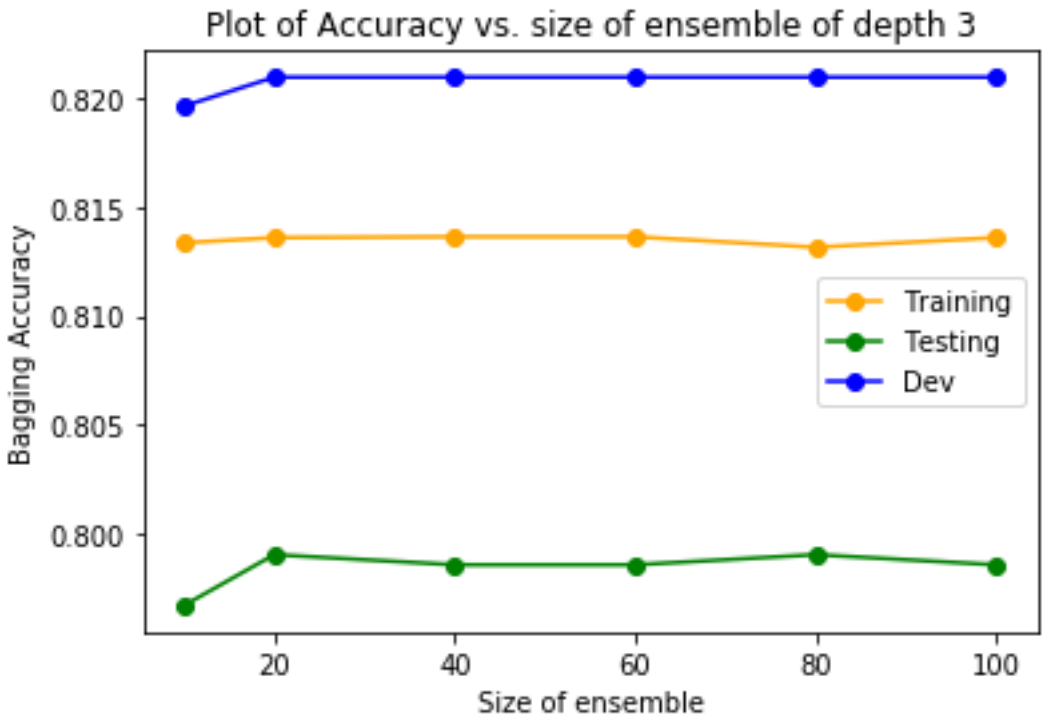
Depth = 1, Ensemble size = 40
Training Accuracy: 0.7516, Testing Accuracy: 0.7394, Dev Accuracy: 0.7606
Depth = 1, Ensemble size = 60
Training Accuracy: 0.7516, Testing Accuracy: 0.7394, Dev Accuracy: 0.7606
Depth = 1, Ensemble size = 80
Training Accuracy: 0.7516, Testing Accuracy: 0.7394, Dev Accuracy: 0.7606
Depth = 1, Ensemble size = 100
Training Accuracy: 0.7516, Testing Accuracy: 0.7394, Dev Accuracy: 0.7606



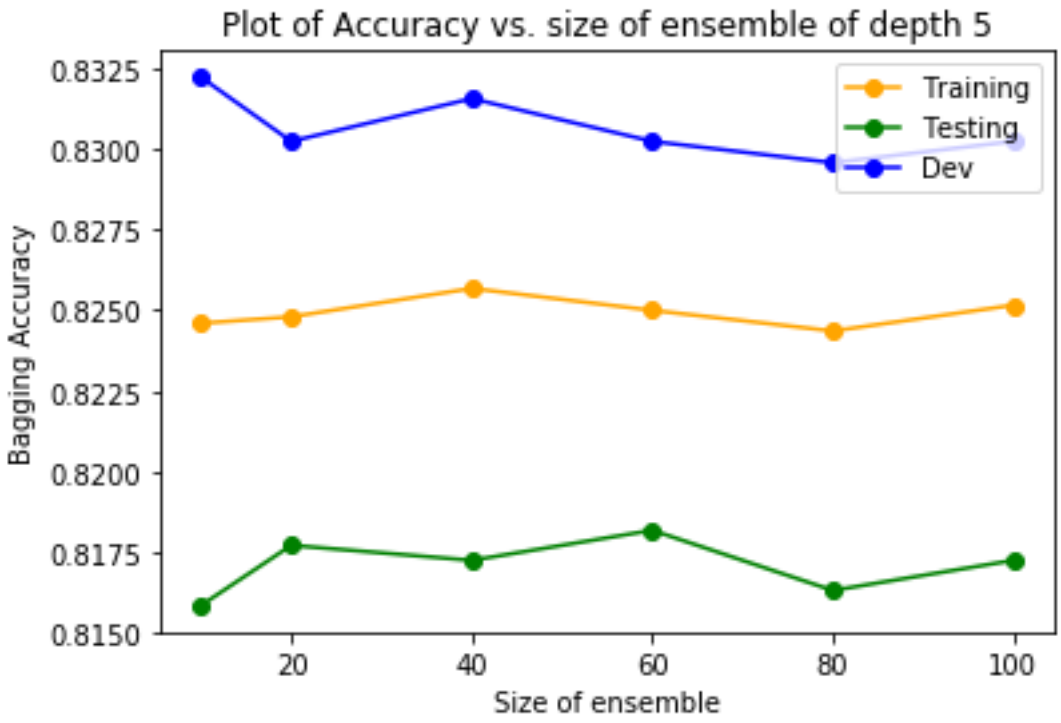
Depth = 2, Ensemble size = 10
Training Accuracy: 0.7837, Testing Accuracy: 0.7641, Dev Accuracy: 0.7898
Depth = 2, Ensemble size = 20
Training Accuracy: 0.7837, Testing Accuracy: 0.7641, Dev Accuracy: 0.7898
Depth = 2, Ensemble size = 40
Training Accuracy: 0.7837, Testing Accuracy: 0.7641, Dev Accuracy: 0.7898
Depth = 2, Ensemble size = 60
Training Accuracy: 0.7837, Testing Accuracy: 0.7641, Dev Accuracy: 0.7898
Depth = 2, Ensemble size = 80
Training Accuracy: 0.7837, Testing Accuracy: 0.7641, Dev Accuracy: 0.7898
Depth = 2, Ensemble size = 100
Training Accuracy: 0.7837, Testing Accuracy: 0.7641, Dev Accuracy: 0.7898



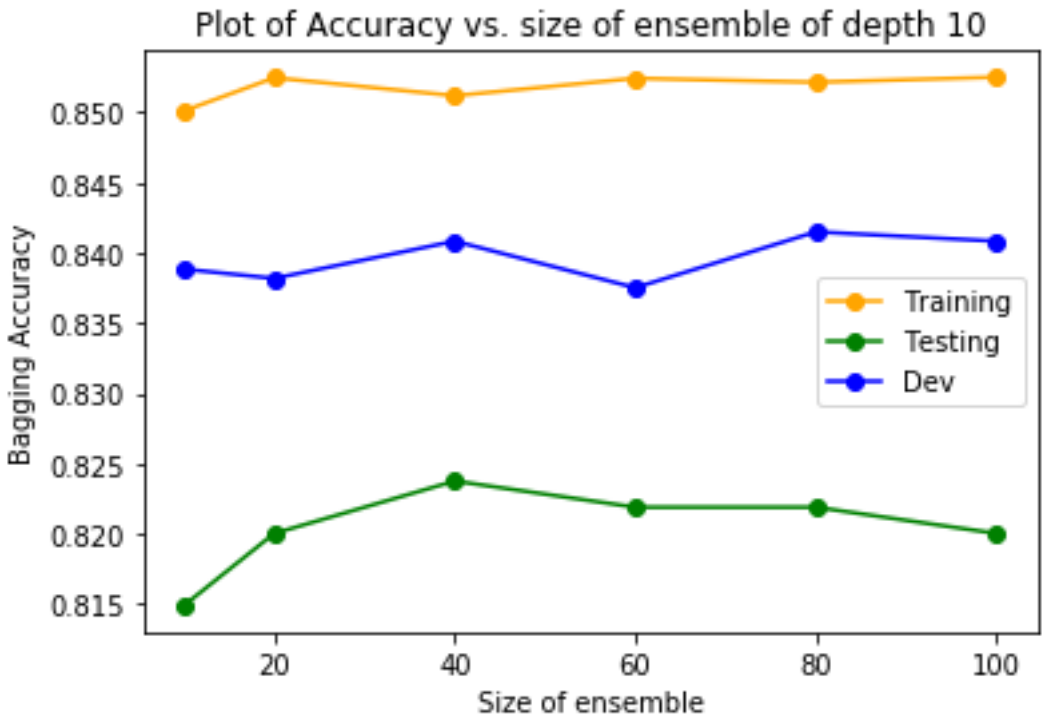
Depth = 3, Ensemble size = 10
Training Accuracy: 0.8134, Testing Accuracy: 0.7967, Dev Accuracy: 0.8196
Depth = 3, Ensemble size = 20
Training Accuracy: 0.8136, Testing Accuracy: 0.7991, Dev Accuracy: 0.821
Depth = 3, Ensemble size = 40
Training Accuracy: 0.8136, Testing Accuracy: 0.7986, Dev Accuracy: 0.821
Depth = 3, Ensemble size = 60
Training Accuracy: 0.8136, Testing Accuracy: 0.7986, Dev Accuracy: 0.821
Depth = 3, Ensemble size = 80
Training Accuracy: 0.8132, Testing Accuracy: 0.7991, Dev Accuracy: 0.821
Depth = 3, Ensemble size = 100
Training Accuracy: 0.8136, Testing Accuracy: 0.7986, Dev Accuracy: 0.821



Depth = 5, Ensemble size = 10
Training Accuracy: 0.8246, Testing Accuracy: 0.8159, Dev Accuracy: 0.8322
Depth = 5, Ensemble size = 20
Training Accuracy: 0.8248, Testing Accuracy: 0.8177, Dev Accuracy: 0.8302
Depth = 5, Ensemble size = 40
Training Accuracy: 0.8257, Testing Accuracy: 0.8172, Dev Accuracy: 0.8316
Depth = 5, Ensemble size = 60
Training Accuracy: 0.825, Testing Accuracy: 0.8182, Dev Accuracy: 0.8302
Depth = 5, Ensemble size = 80
Training Accuracy: 0.8244, Testing Accuracy: 0.8163, Dev Accuracy: 0.8296
Depth = 5, Ensemble size = 100
Training Accuracy: 0.8252, Testing Accuracy: 0.8172, Dev Accuracy: 0.8302



Depth = 10, Ensemble size = 10
Training Accuracy: 0.8501, Testing Accuracy: 0.8149, Dev Accuracy: 0.8389
Depth = 10, Ensemble size = 20
Training Accuracy: 0.8525, Testing Accuracy: 0.82, Dev Accuracy: 0.8382
Depth = 10, Ensemble size = 40
Training Accuracy: 0.8512, Testing Accuracy: 0.8238, Dev Accuracy: 0.8408
Depth = 10, Ensemble size = 60
Training Accuracy: 0.8524, Testing Accuracy: 0.8219, Dev Accuracy: 0.8375
Depth = 10, Ensemble size = 80
Training Accuracy: 0.8522, Testing Accuracy: 0.8219, Dev Accuracy: 0.8415
Depth = 10, Ensemble size = 100
Training Accuracy: 0.8525, Testing Accuracy: 0.82, Dev Accuracy: 0.8408



Observations:

1. Hyperparameters chosen on the basis of performance on dev data:
Depth: 10, Size of ensemble: 80

2. There is no significant change in performance on training, testing or dev set for the same depth over different values of ensemble size.

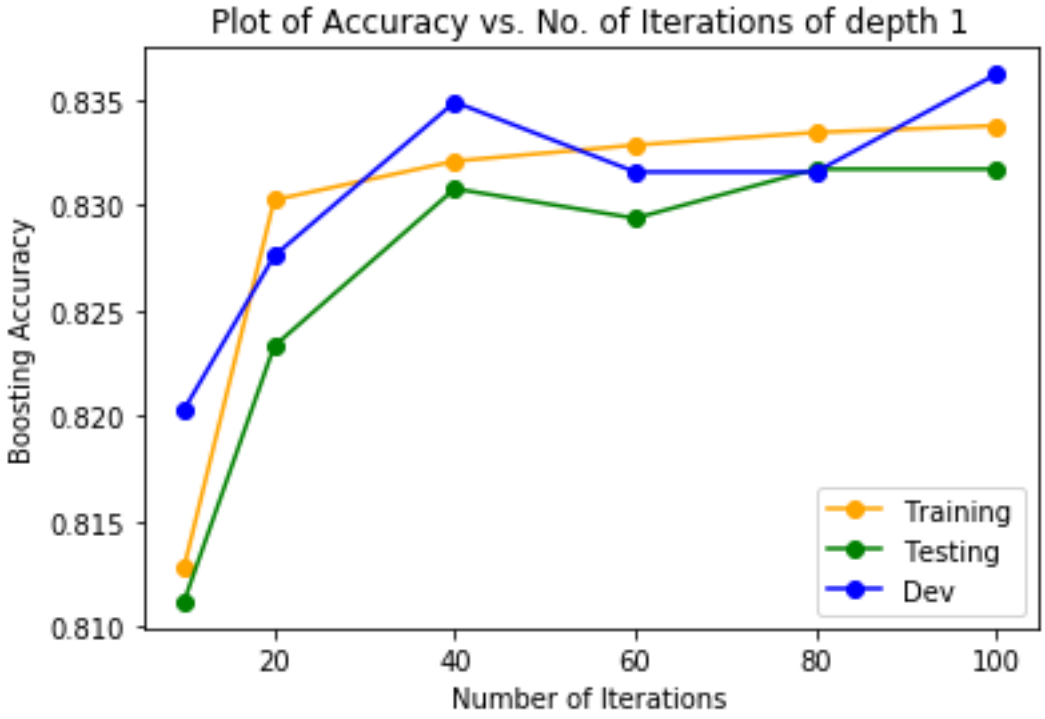
3. The training, test and dev accuracy increases with increasing the depth.

4. The variation in performance for different values of ensemble size increases with increasing depth.

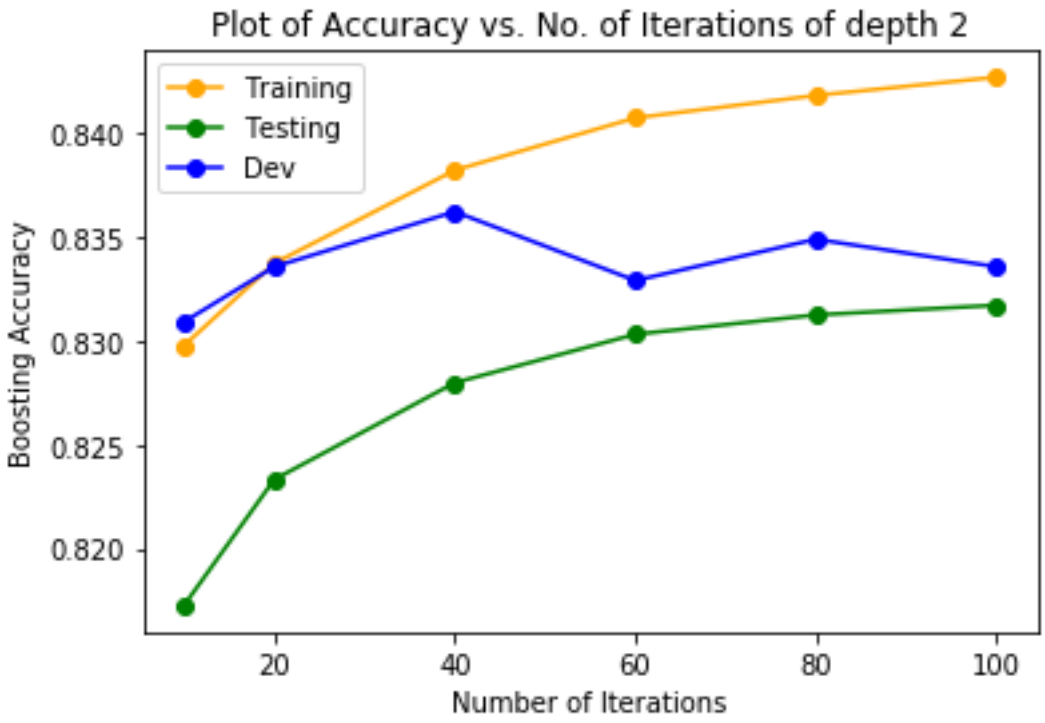
b)Sklearn AdaBoost:

Depth = 1, Number of Iterations = 10
Training Accuracy: 0.8128, Testing Accuracy: 0.8112, Dev Accuracy: 0.8203
Depth = 1, Number of Iterations = 20
Training Accuracy: 0.8302, Testing Accuracy: 0.8233, Dev Accuracy: 0.8276
Depth = 1, Number of Iterations = 40
Training Accuracy: 0.8321, Testing Accuracy: 0.8308, Dev Accuracy: 0.8349
Depth = 1, Number of Iterations = 60
Training Accuracy: 0.8328, Testing Accuracy: 0.8294, Dev Accuracy: 0.8316
Depth = 1, Number of Iterations = 80

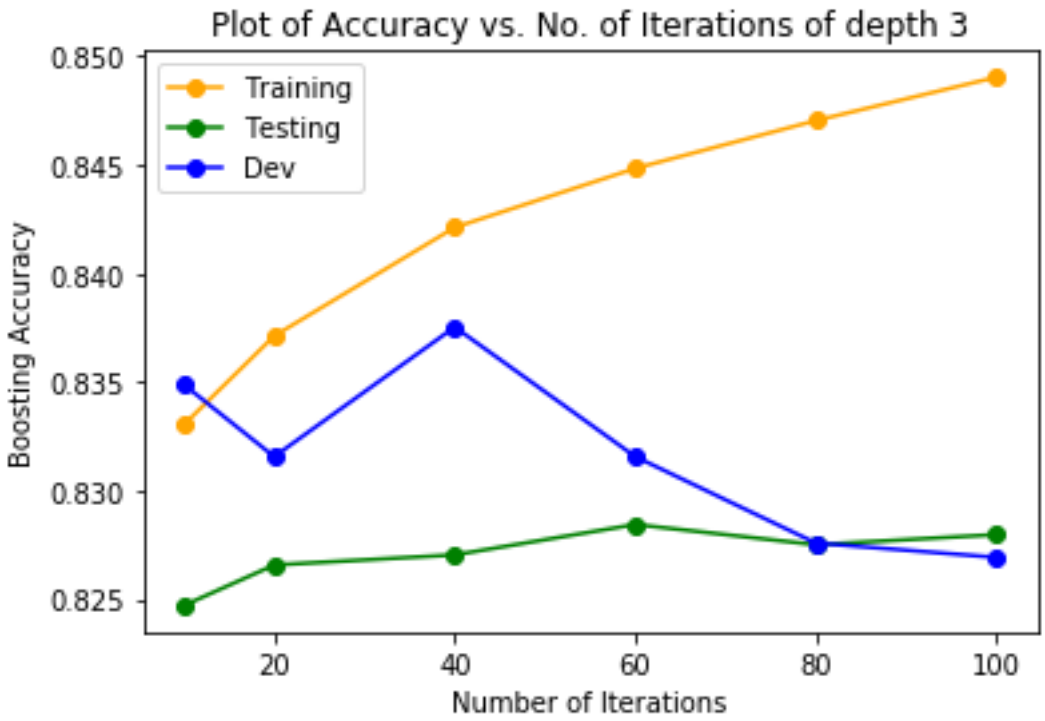
Training Accuracy: 0.8334, Testing Accuracy: 0.8317, Dev Accuracy: 0.8316
Depth = 1, Number of Iterations = 100
Training Accuracy: 0.8338, Testing Accuracy: 0.8317, Dev Accuracy: 0.8362



Depth = 2, Number of Iterations = 10
Training Accuracy: 0.8298, Testing Accuracy: 0.8172, Dev Accuracy: 0.8309
Depth = 2, Number of Iterations = 20
Training Accuracy: 0.8337, Testing Accuracy: 0.8233, Dev Accuracy: 0.8336
Depth = 2, Number of Iterations = 40
Training Accuracy: 0.8382, Testing Accuracy: 0.828, Dev Accuracy: 0.8362
Depth = 2, Number of Iterations = 60
Training Accuracy: 0.8407, Testing Accuracy: 0.8303, Dev Accuracy: 0.8329
Depth = 2, Number of Iterations = 80
Training Accuracy: 0.8418, Testing Accuracy: 0.8312, Dev Accuracy: 0.8349
Depth = 2, Number of Iterations = 100
Training Accuracy: 0.8427, Testing Accuracy: 0.8317, Dev Accuracy: 0.8336



Depth = 3, Number of Iterations = 10
Training Accuracy: 0.833, Testing Accuracy: 0.8247, Dev Accuracy: 0.8349
Depth = 3, Number of Iterations = 20
Training Accuracy: 0.8371, Testing Accuracy: 0.8266, Dev Accuracy: 0.8316
Depth = 3, Number of Iterations = 40
Training Accuracy: 0.8421, Testing Accuracy: 0.827, Dev Accuracy: 0.8375
Depth = 3, Number of Iterations = 60
Training Accuracy: 0.8448, Testing Accuracy: 0.8284, Dev Accuracy: 0.8316
Depth = 3, Number of Iterations = 80
Training Accuracy: 0.847, Testing Accuracy: 0.8275, Dev Accuracy: 0.8276
Depth = 3, Number of Iterations = 100
Training Accuracy: 0.849, Testing Accuracy: 0.828, Dev Accuracy: 0.8269



Observations:

1. There is a general increase in training performance with increase in number of iterations for a specified depth.

2. The hyperparameters chosen on the basis of performance on dev set:
Depth: 3, Number of iterations: 40.

3. The nature of curves for training and test remain similar for all combinations of hyperparameters.

4. Number of iterations = 40 is optimal or near-optimal for all values of depth.

Exercise 7. Automatic hyper-parameter tuning via Bayesian Optimization. For this homework, you need to use BO software to perform hyper-parameter search for Bagging and Boosting classifiers: two hyper-parameters (size of ensemble and depth of decision tree).
You will employ Bayesian Optimization (BO) software to automate the search for the best hyper-parameters by running it for 50 iterations. Plot the number of BO iterations on x-axis and performance of the best hyper-parameters at any point of time (performance of the corresponding trained classifier on the validation data) on y-axis.
Additionally, list the sequence of candidate hyper-parameters that were selected along the BO iterations.
You can use one of the following BO softwares or others as needed.
Spearmint: https://github.com/JasperSnoek/spearmint

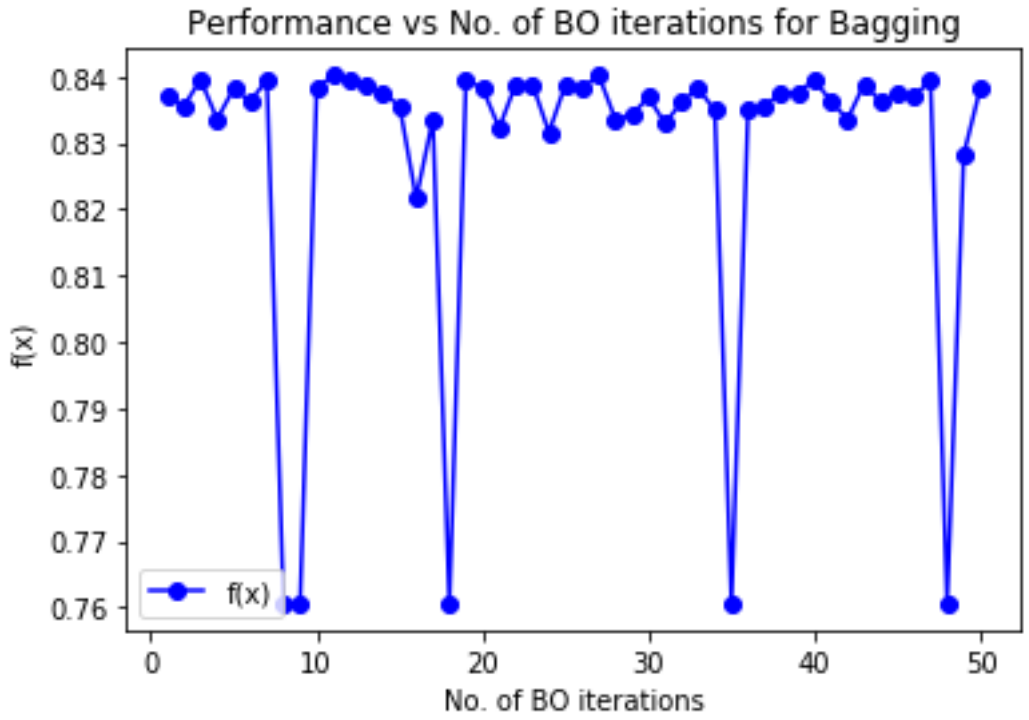
SMAC: http://www.cs.ubc.ca/labs/beta/Projects/SMAC/

Answer:

List of candidate Hyper-parameters for Bagging:
Depth: 8.0, Size of ensemble: 21.0, f(x): 0.8381962864721485
Depth: 10.0, Size of ensemble: 100.0, f(x): 0.8348806366047745
Depth: 10.0, Size of ensemble: 10.0, f(x): 0.8375331564986738
Depth: 10.0, Size of ensemble: 50.0, f(x): 0.8415119363395226
Depth: 10.0, Size of ensemble: 40.0, f(x): 0.8375331564986738
Depth: 10.0, Size of ensemble: 75.0, f(x): 0.8368700265251989
Depth: 1.0, Size of ensemble: 100.0, f(x): 0.7606100795755968
Depth: 10.0, Size of ensemble: 59.0, f(x): 0.8408488063660478
Depth: 10.0, Size of ensemble: 53.0, f(x): 0.8375331564986738
Depth: 1.0, Size of ensemble: 48.0, f(x): 0.7606100795755968
Depth: 10.0, Size of ensemble: 66.0, f(x): 0.8395225464190982
Depth: 10.0, Size of ensemble: 90.0, f(x): 0.8388594164456233
Depth: 1.0, Size of ensemble: 10.0, f(x): 0.7606100795755968
Depth: 10.0, Size of ensemble: 16.0, f(x): 0.843501326259947
Depth: 10.0, Size of ensemble: 47.0, f(x): 0.8368700265251989
Depth: 10.0, Size of ensemble: 32.0, f(x): 0.8395225464190982
Depth: 1.0, Size of ensemble: 76.0, f(x): 0.7606100795755968
Depth: 10.0, Size of ensemble: 84.0, f(x): 0.8381962864721485
Depth: 1.0, Size of ensemble: 29.0, f(x): 0.7606100795755968
Depth: 10.0, Size of ensemble: 19.0, f(x): 0.8395225464190982
Depth: 10.0, Size of ensemble: 14.0, f(x): 0.8375331564986738
Depth: 6.0, Size of ensemble: 62.0, f(x): 0.8342175066312998
Depth: 10.0, Size of ensemble: 96.0, f(x): 0.8388594164456233
Depth: 8.0, Size of ensemble: 27.0, f(x): 0.8362068965517241
Depth: 10.0, Size of ensemble: 62.0, f(x): 0.8362068965517241
Depth: 8.0, Size of ensemble: 70.0, f(x): 0.8355437665782494
Depth: 8.0, Size of ensemble: 58.0, f(x): 0.8355437665782494
Depth: 10.0, Size of ensemble: 71.0, f(x): 0.8408488063660478
Depth: 1.0, Size of ensemble: 63.0, f(x): 0.7606100795755968
Depth: 6.0, Size of ensemble: 88.0, f(x): 0.830238726790451
Depth: 8.0, Size of ensemble: 36.0, f(x): 0.8355437665782494
Depth: 10.0, Size of ensemble: 36.0, f(x): 0.8368700265251989
Depth: 8.0, Size of ensemble: 96.0, f(x): 0.8342175066312998
Depth: 8.0, Size of ensemble: 49.0, f(x): 0.833554376657825
Depth: 8.0, Size of ensemble: 81.0, f(x): 0.8348806366047745
Depth: 10.0, Size of ensemble: 27.0, f(x): 0.8328912466843501
Depth: 9.0, Size of ensemble: 87.0, f(x): 0.8362068965517241
Depth: 10.0, Size of ensemble: 80.0, f(x): 0.8395225464190982
Depth: 8.0, Size of ensemble: 32.0, f(x): 0.8395225464190982
Depth: 9.0, Size of ensemble: 92.0, f(x): 0.8368700265251989

Depth: 9.0, Size of ensemble: 17.0, f(x): 0.8328912466843501
Depth: 5.0, Size of ensemble: 22.0, f(x): 0.8322281167108754
Depth: 10.0, Size of ensemble: 14.0, f(x): 0.8342175066312998
Depth: 7.0, Size of ensemble: 10.0, f(x): 0.8315649867374005
Depth: 9.0, Size of ensemble: 12.0, f(x): 0.8368700265251989
Depth: 1.0, Size of ensemble: 92.0, f(x): 0.7606100795755968
Depth: 1.0, Size of ensemble: 39.0, f(x): 0.7606100795755968
Depth: 1.0, Size of ensemble: 84.0, f(x): 0.7606100795755968
Depth: 6.0, Size of ensemble: 43.0, f(x): 0.8342175066312998

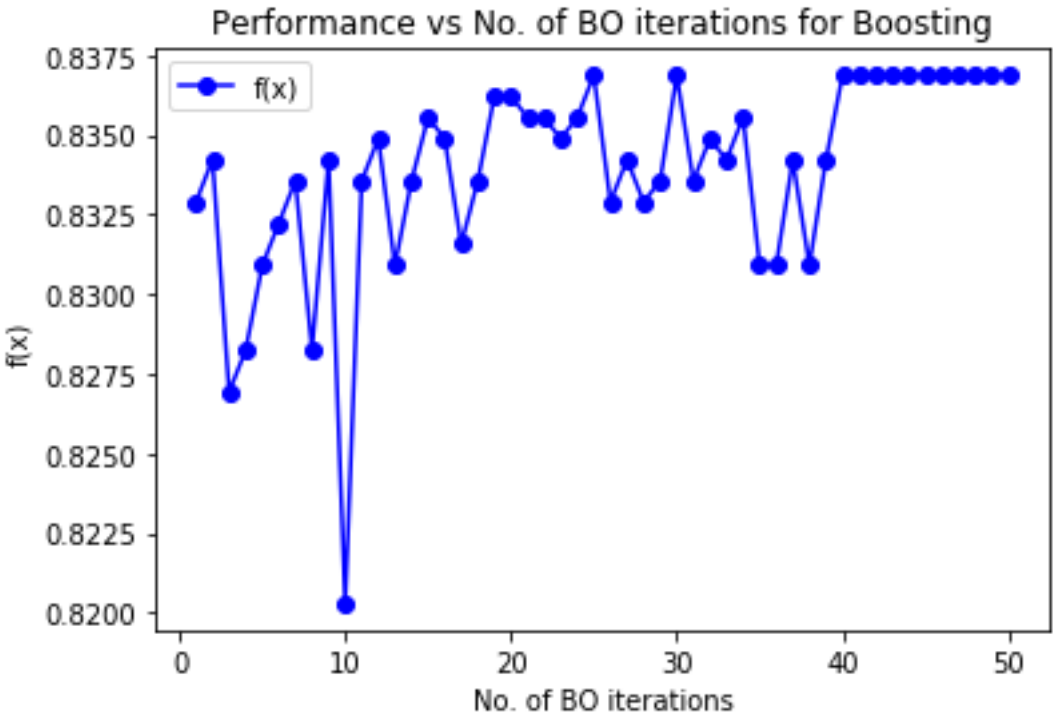
Hyperparameters chosen for Bagging by Bayesian Optimization:
Depth: 10.0, Size of ensemble: 16.0



List of candidate Hyper-parameters for Boosting:
Depth: 3.0, Number of iterations: 65.0, f(x): 0.8342175066312998
Depth: 3.0, Number of iterations: 100.0, f(x): 0.8269230769230769
Depth: 3.0, Number of iterations: 80.0, f(x): 0.8282493368700266
Depth: 3.0, Number of iterations: 10.0, f(x): 0.8309018567639257
Depth: 3.0, Number of iterations: 69.0, f(x): 0.8322281167108754
Depth: 1.0, Number of iterations: 70.0, f(x): 0.833554376657825
Depth: 1.0, Number of iterations: 23.0, f(x): 0.8282493368700266
Depth: 1.0, Number of iterations: 92.0, f(x): 0.8342175066312998
Depth: 1.0, Number of iterations: 10.0, f(x): 0.8202917771883289
Depth: 3.0, Number of iterations: 17.0, f(x): 0.833554376657825

Depth: 3.0, Number of iterations: 91.0, f(x): 0.8348806366047745
Depth: 3.0, Number of iterations: 30.0, f(x): 0.8309018567639257
Depth: 1.0, Number of iterations: 66.0, f(x): 0.833554376657825
Depth: 3.0, Number of iterations: 24.0, f(x): 0.8355437665782494
Depth: 1.0, Number of iterations: 86.0, f(x): 0.8348806366047745
Depth: 3.0, Number of iterations: 94.0, f(x): 0.8315649867374005
Depth: 2.0, Number of iterations: 89.0, f(x): 0.833554376657825
Depth: 3.0, Number of iterations: 43.0, f(x): 0.8362068965517241
Depth: 3.0, Number of iterations: 40.0, f(x): 0.8362068965517241
Depth: 3.0, Number of iterations: 41.0, f(x): 0.8355437665782494
Depth: 3.0, Number of iterations: 48.0, f(x): 0.8355437665782494
Depth: 3.0, Number of iterations: 45.0, f(x): 0.8348806366047745
Depth: 3.0, Number of iterations: 53.0, f(x): 0.8355437665782494
Depth: 3.0, Number of iterations: 36.0, f(x): 0.8368700265251989
Depth: 1.0, Number of iterations: 38.0, f(x): 0.8328912466843501
Depth: 1.0, Number of iterations: 100.0, f(x): 0.8342175066312998
Depth: 1.0, Number of iterations: 81.0, f(x): 0.8328912466843501
Depth: 1.0, Number of iterations: 50.0, f(x): 0.833554376657825
Depth: 3.0, Number of iterations: 37.0, f(x): 0.8368700265251989
Depth: 3.0, Number of iterations: 21.0, f(x): 0.833554376657825
Depth: 3.0, Number of iterations: 85.0, f(x): 0.8348806366047745
Depth: 3.0, Number of iterations: 51.0, f(x): 0.8342175066312998
Depth: 3.0, Number of iterations: 56.0, f(x): 0.8355437665782494
Depth: 3.0, Number of iterations: 13.0, f(x): 0.8309018567639257
Depth: 1.0, Number of iterations: 44.0, f(x): 0.8309018567639257
Depth: 1.0, Number of iterations: 55.0, f(x): 0.8342175066312998
Depth: 1.0, Number of iterations: 75.0, f(x): 0.8309018567639257
Depth: 1.0, Number of iterations: 97.0, f(x): 0.8342175066312998
Depth: 3.0, Number of iterations: 37.0, f(x): 0.8368700265251989
Depth: 3.0, Number of iterations: 36.0, f(x): 0.8368700265251989
Depth: 3.0, Number of iterations: 36.0, f(x): 0.8368700265251989
Depth: 3.0, Number of iterations: 37.0, f(x): 0.8368700265251989
Depth: 3.0, Number of iterations: 37.0, f(x): 0.8368700265251989
Depth: 3.0, Number of iterations: 36.0, f(x): 0.8368700265251989
Depth: 3.0, Number of iterations: 37.0, f(x): 0.8368700265251989
Depth: 3.0, Number of iterations: 36.0, f(x): 0.8368700265251989
Depth: 3.0, Number of iterations: 38.0, f(x): 0.8368700265251989
Depth: 3.0, Number of iterations: 36.0, f(x): 0.8368700265251989
Depth: 3.0, Number of iterations: 37.0, f(x): 0.8368700265251989

Hyperparameters chosen for Boosting by Bayesian Optimization:
Depth: 3.0, Number of iterations: 36.0


Performance vs No. of BO iterations for Boosting
f(x)
0.8375
0.8350
0.8325
0.8300
0.8275
0.8250
0.8225
0.8200
f(x)
0
10
20
30
40
50
No. of BO iterations