Machine Learning - Final project

Thomas Astad Sve thomaasv@stud.ntnu.no
Computer Science, Artificial Intelligence
Norwegian University of Science & Technology
University & Professional Studies ax004206@ucsmail.ucsd.edu
University of California, San Diego

Abstract

This paper compares five different supervised learning methods using the popular python library scikit-learn (2). The paper tries to reproduce some of the results from the work of Caruana & Niculescu-Mizil (1) by creating four classification problems on three datasets from the UDI repository (3). The five learning algorithms are; Decision Tree, K-Nearest-Neighbors, Random Forrest, Boosted Trees and Boosted Stumps. The paper tries to categorize these learning algorithms and see which one of them performs best on the different problems and if there is a single learning algorithm that is universally best.

Keywords: K-Nearest-Neighbors, Decision Tree, Random Forrest, Boosted trees, Boosted stumps

1. Introduction

As a final project for the COGS 118A class at University of California, San Diego. I was told to choose three classifiers out of those tested in Caruana and Niculescu-Mizil (1). I will train my classifiers using the settings described mentioned in their paper and try to reproduce the results they got. I will not be able to get the exact same results because we will not have the exact same code, and we will mostly be using the popular python library Scikit-learn(2) to train our classifiers.

2. Method

2.1 Learning Algorithms

KNN: For k-Nearest-Neighbors I have generated a cross-validation function that searches for the optimal k. This function splits the trainingset into 5 validation set. Because the trainingset have 5000 instances, each of these sets have 1000 instances and are validated against the rest of the instances in the trainingset (the remaining 4000). These sets are tested for 26 k-values ranging from k = 1 to k = length of trainingset. When finished, the function chooses the k-value that overall got the lowest error. This optimal k-value is then used when creating the classifier, and when testing classifier against the testset.

Random Forrest: For random forrest I used scikit-learn's (2) RandomForrestClassifier. Setting the number of estimaters to 1024 the classifier is set to 1024 tree's, which is the same amount used in

Decision Tree: For decision tree we simply used the scikit-learn method (2) DecisionTreeClassifier. The function to measure the quality of the split is set to entropy (instead of sklearns default gini), while the split criteria is by default set to best-split.

Boosted Trees: For tree boosting, I used 3000 decision tree with a max depth of 6 as weak learners. I set the learning rate to 0.4 and used the scikit-learn method Gradient Tree Boosting to boost my trees.

Boosted Stumps: Here I have created a single level decision tree before boosting it using AdaBoost with 400 estimators. Running test for the two two algorithm SAMME.R real boosting algorithm and SAMME discrete boosting algorithm.

SVM and Neural-Networks: I highly considered including SVM and Neural-Network in the project aswell, but due to computational problem, I decided to drop the algorithms. If I had more time, and a more powerful computer, I would have spent more time including these algorithms.

2.2 Datasets

I am comparing the algorithms on three different binary classification problems. Adult, Covertype and Letter-recognition are all from the UCI Repository (3). The **Adult** dataset contains categorial features, which I have transformed into boolean-values using OneHotEncoder. This dataset includes rows with where there is unknown values. In order to clean the dataset, I decided to remove these rows. A different approach could be to insert dummy variables for these unknown values, this have not been done here.

With **Covertype** I have used the largest class (2 - Lodgepole Pine) as positive and the rest as negative. I devided the Letter-recognition problem into two different binary problems. **Letter.p1** threats 'O' as positive and the rest as negative. With **Letter.p2** I used A-M as positive and the rest as negative. The same classification problems is also found in the work of Caruana & Niculescu-Mizil (1).

3. Experiment

Even though this experiment is ment to try to reproduce the results from work of Caruana & Niculescu-Mizil (1), I have been forced to reduce testing of parameters and reduce size of trees due to memory and computation limits. For instance, my computer couldn't classify Covertype for 1024 trees on random forrest, due to memory limits. And I therefore decided to reduce the tree size to 200. In order to save computational time, the datasets are generated seperately and the arrays created are then stored, so when testing a method with a dataset, all it needs to do is load the generated arrays and use them in the experiment.

Table 1: Scores for each learning algorithm by problem

Model	Adult	Covertype	Letter.p1	Letter.p2	MEAN
BST-DT	0.846	0.816	0.989	**0.959**	0.903
RF	0.847	**0.818**	0.985	0.946	0.899
KNN	0.781	0.786	**0.990**	0.958	0.879
BST-STMP	**0.858**	0.764	0.982	0.820	0.856
DT	0.801	0.736	0.982	0.888	0.852

Table 1 shows the results when running the classifiers on the different classification problems. Here the results are based on accuracy and ranked by mean in the last column. From the results we can see that although new learning methods such as boosting and random forrest achives excellent performance on all of the four problems, we can not pick out one method that is universal best.

The overall results matches the results from the work of Caruana & Niculescu-Mizil (1). But the induvidual result for each problems do not match. For instance, I have gotten a accuracy of 76% on Covertype for the BST-STMP classifier, which is over ten percent better than they did for the same problem. While for Adult I have gotten a ten percent worse than they did for the same algorithm. This may be because of my parameter settings wern't exactly the same due to computational limits, or simply because their implementations are different from the ones used by scikit-learn. For boosted stumps they considered the boosted trees after while boosting them to prevent from overfitting, something I did not. I simply created a lvl one decision tree classifier and boosted it using AdaBoost with 3000 estimators and a learning-rate of 0.04.

For both the letter classification problems, the results are very good, with close to 99% accuracy on p1 (which threated only 'O' as positive, rest as negative). This may be because the classification problem was considered easy to predict, where most of the data were labeled negative. The p2 classification problem (Where A-M where positive and rest negative) also gave great results on most learning methods, but gave a bigger diversity than p1.

New learning methods such as boosting and random forrest achieves excellent performance on all of the four problemsets. While boosting decision trees gave an overall best score, it was only best on one induvidual classification problem. This matches the results from Caruana & Niculescu-Mizil (1), where they concluded that as the No Free Lunch Theorem suggests, there is no universally best learning algorithm. Some learning algorithms may perform really bad on some problem, while they perform best on others.

4. Conclusion

It is hard to reproduce exact same results from other papers when I don't have the exact same code as they had and I am not able to use the exact same parameters due to computational limit. While not having all the datasets and all the learning methods they had, I still

feel like I have a sufficent result in that shows what learning algorithms performs best overall.

With excellent performance on all five classification problems, boosted trees were the best learning algorithm overall with Random Trees as close second. While these algoritms performed good on most problems, they were not best on all. The KNN algorithm, which came in third overall, had great results for both classification problems on the letter dataset and did best on Letter.p1, but did poorly on some problems. We can therefore conclude with the same conclusion Caruana & Niculescu-Mizil (1) did, that even the best models sometimes perform poorly, and models with poor average might perform great on some problems.

References

[1] Rich Caruana, Alexandru Niculescu-Mizil. An Empirical Comparison of Supervised Learning Algorithms

[2] Scikit-learn: Machine Learning in Python, Pedregosa, F. and Varoquaux, G. and Gramfort, A. and Michel, V. and Thirion, B. and Grisel, O. and Blondel, M. and Prettenhofer, P. and Weiss, R. and Dubourg, V. and Vanderplas, J. and Passos, A. and Cournapeau, D. and Brucher, M. and Perrot, M. and Duchesnay, E., Journal of Machine Learning Research, 12, 2825–2830, 2011

[3] Blake, C., Merz, C. (1998). UCI repository of machine learning databases.

Code

```
import numpy as np
from utils import *
#from k_nearest_neighbors import run_kNN
from k_nearest_sklearn import run_kNN_sklearn
from decision_tree import run_decisiontree
from svm import run_svm
#from neural_networks import run_neural_nets
from neural_network_pybrain import run_neural_network
from random_forrest import run_random_forrest
from boosted_trees import run_boosted_trees
from boosted_stumps import run_boosted_stumps

def main():

    # In order to save computational-time, the data is preprocessed
    datasets = []
    datasets.append(load_generated_data("adult_generated"))
    datasets.append(load_generated_data("letterp1_generated"))
    datasets.append(load_generated_data("letterp2_generated"))
    datasets.append(load_generated_data("covtype_generated"))

    experiments = ['Adult', 'Letter.p1', 'Letter.p2', 'Covertype']

    for i in range(len(datasets)):
        print("--------", experiments[i], "------------")
        # Split dataset into train and test. Setting train-set to fixed 5000, and the rest to test
        X_train, X_test, y_train, y_test = split_into_train_test(datasets[i][0], datasets[i][1], train_size = 5000)
        # Run k_Nearest_Neighbors
        #run_kNN(X_train, y_train, X_test, y_test)
        run_kNN_sklearn(X_train, y_train, X_test, y_test)

        # Run decision_tree http://scikit-learn.org/stable/modules/tree.html
        run_decisiontree(X_train, y_train, X_test, y_test)

        # Run support_vector_machine http://scikit-learn.org/stable/modules/svm.html
        #run_svm(X_train, y_train, X_test, y_test)

        # Run neural_nets http://pybrain.org/docs/tutorial/fnn.html
        #run_neural_network(X_train, y_train, X_test, y_test)

        # Run random_forrest http://scikit-learn.org/stable/modules/ensemble.html#random-forests
        run_random_forrest(X_train, y_train, X_test, y_test)

        # Run boosted trees
        run_boosted_trees(X_train, y_train, X_test, y_test)

        # Run boosted stumps using Adaboost
        run_boosted_stumps(X_train, y_train, X_test, y_test)

        print(" ")


if __name__ == "__main__":
    main()
```

```
import scipy.io
import numpy as np
from sklearn.grid_search import GridSearchCV
from math import floor
from random import shuffle


def save_generated_data(X, y, filename):
    np.savez("generated_arrays/" + filename, X=X, y=y)

def load_generated_data(filename):
    data = np.load("generated_arrays/" + filename + ".npz")
    return data['X'], data['y']

def grid_search_cross_val(classifier, params, X, y):
    clf = GridSearchCV(estimator=classifier, param_grid=params, cv=5)
    clf.fit(X, y)
    return clf

def split_into_train_test(X, y, train_size):
    data = zip(X, y)
    shuffle(data)
    X, y = zip(*data)

    return X[:train_size], X[train_size:], y[:train_size], y[train_size:]

def calculate_error(predictions, actual_values):
```

```
    errors = 0
    for i in range(len(predictions)):
        if predictions[i] != actual_values[i]:
            errors += 1
    return errors / float(len(predictions))
```

Decisiontree - code:

```
from sklearn import tree
from utils import *
"""
Decision trees (DT): we vary the splitting criterion,
pruning options, and smoothing (Laplacian or
Bayesian smoothing). We use all of the tree models
in Buntine's IND package (Buntine & Caruana, 1991):
BAYES, ID3, CART, CART0, C4, MML, and SMML.
We also generate trees of type C44LS (C4 with no
pruning and Laplacian smoothing), C44BS (C44 with
Bayesian smoothing), and MMLLS (MML with Laplacian
smoothing). See (Provost & Domingos, 2003) for
a description of C44LS.
"""

def build_decision_tree(X, y):
    # Decision tree using entropy
    clf = tree.DecisionTreeClassifier(criterion="entropy")
    clf = clf.fit(X, y)
    return clf

def test_decisiontree_classifier(clf, X, y):
    return clf.score(X, y)

def run_decisiontree(X_train, y_train, X_test, y_test):
    clf = build_decision_tree(X_train, y_train)
    score = test_decisiontree_classifier(clf, X_test, y_test)
    print("Decisiontree, score: ", score)
```

K-Nearest-Neighbors - code:

```
from sklearn.neighbors import KNeighborsClassifier
"""
KNN: we use 26 values of K ranging from K = 1 to
K = |trainset|. We use KNN with Euclidean distance
and Euclidean distance weighted by gain ratio. We
also use distance weighted KNN, and locally weighted
averaging. The kernel widths for locally weighted averaging
vary from 2^0 to 2^10 times the minimum distance
between any two points in the train set.
"""

def build_kNN(X, y, k):
    clf = KNeighborsClassifier(n_neighbors=k)
    clf.fit(X, y)
    return clf

def test_kNN(X, y, clf):
    return clf.score(X, y)

def find_optimal_k(X, y):

    # Split trainingset into 5 validationset
    splitted_X_set = [X[i::5] for i in range(5)]
    splitted_y_set = [y[i::5] for i in range(5)]

    # 26 values of K ranging from K = 1 to K = |trainset|
    k_values = [1, 2, 3, 5, 7, 10, 12, 15, 20, 25, 30, 40, 50, 70, 100, 150, 200, 300, 500, 750, 1000, 125, 1500, 2500,
    optimal_ks = [0 for i in range(5)]
    errors = {k : 0 for k in k_values}
    for i in range(5):
        validation_X = splitted_X_set[i]
        validation_y = splitted_y_set[i]
        training_X = [item for s in splitted_X_set if s is not validation_X for item in s]
        training_y = [item for s in splitted_y_set if s is not validation_y for item in s]
        for k in k_values:
            clf = build_kNN(training_X, training_y, k)
            test_score = test_kNN(validation_X, validation_y, clf)
            errors[k] += 1 - test_score

    optimal_k = min(errors, key=errors.get)
    print(errors)
    print(optimal_k)
    return optimal_k

def run_kNN_sklearn(X_train, y_train, X_test, y_test):
```

```
    optimal_k = find_optimal_k(X_train, y_train)
    clf = build_kNN(X_train, y_train, optimal_k)
    score = test_kNN(X_test, y_test, clf)
    print("kNN ", score)
```

Random Forrest - code:

```
"""
Random Forests (RF): we tried both the BreimanCutler
and Weka implementations; Breiman-Cutler
yielded better results so we report those here. The
forests have 1024 trees. The size of the feature set
considered at each split is 1,2,4,6,8,12,16 or 20
"""

from sklearn.cross_validation import cross_val_score
from sklearn.ensemble import RandomForestClassifier
from utils import *

def build_random_forrest_clf(X, y):
    clf = RandomForestClassifier(n_estimators=200, bootstrap = True)
    clf = clf.fit(X, y)
    return clf

def test_random_forrest_clf(X, y, clf):
    return clf.score(X, y)

def run_random_forrest(X_train, y_train, X_test, y_test):
    clf = build_random_forrest_clf(X_train, y_train)
    score = test_random_forrest_clf(X_test, y_test, clf)
    print("Random forrest, score: ", score)
```

Boosted trees - code:

```
"""
With boosted trees
(BST-DT) we boost each tree type as well. Boosting
can overfit, so we consider boosted trees after
2,4,8,16,32,64,128,256,512,1024 and 2048 steps
of boosting.
"""

from sklearn.ensemble import GradientBoostingRegressor
from sklearn.ensemble import GradientBoostingClassifier
from utils import *

def build_boosted_trees(X, y):
    #clf = GradientBoostingRegressor(n_estimators = 3000, max_depth = 6, learning_rate = 0.04, loss = 'huber')

    #clf.fit(X, y)
    clf = GradientBoostingClassifier(n_estimators = 3000, max_depth = 6, learning_rate = 0.04)
    clf = clf.fit(X, y)
    return clf

def test_boosted_trees(X, y, clf):
    return clf.score(X, y)

def run_boosted_trees(X_train, y_train, X_test, y_test):
    clf = build_boosted_trees(X_train, y_train)
    score = test_boosted_trees(X_test, y_test, clf)
    print("Boosted trees, score: ", score)
```

Boosted Stumps - code:

```
"""

Boosted stumps (BSTSTMP)
we boost single level decision trees generated
with 5 different splitting criteria, each boosted for
2,4,8,16,32,64,128,256,512,1024,2048,4096,8192 steps.

"""
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier

def build_boosted_stumps(X, y):
    # Create a single level decision tree
    dt = DecisionTreeClassifier(max_depth=1, min_samples_leaf=1)
    dt = dt.fit(X, y)

    # Boost the tree using the AdaBoost classifier
    clf = AdaBoostClassifier(base_estimator=dt, n_estimators=3000, learning_rate=0.04)
```

```
    clf = clf.fit(X, y)
    return clf

def test_boosted_stumps(X, y, clf):
    return clf.score(X, y)

def run_boosted_stumps(X_train, y_train, X_test, y_test):
    clf = build_boosted_stumps(X_train, y_train)
    score = test_boosted_stumps(X_test, y_test, clf)
    print("Boosted stumps, score: ", score)
```

Adult data-generation - code:

```
import numpy as np
from utils import save_generated_data

# Adult
adult = "datasets/adult/adult.data.txt"
adult_debug = "datasets/adult/adult_debug.data.txt"

# All possible values

values = {1: ["Private", "Self-emp-not-inc", "Self-emp-inc", "Federal-gov", "Local-gov", "State-gov", "Without-pay", "N
          3: ["Bachelors", "Some-college", "11th", "HS-grad", "Prof-school", "Assoc-acdm", "Assoc-voc", "9th", "7th-8th
          5: ["Married-civ-spouse", "Divorced", "Never-married", "Separated", "Widowed", "Married-spouse-absent", "Marr
          6: ["Tech-support", "Craft-repair", "Other-service", "Sales", "Exec-managerial", "Prof-specialty", "Handlers-
          7: ["Wife", "Own-child", "Husband", "Not-in-family", "Other-relative", "Unmarried"],
          8: ["White", "Asian-Pac-Islander", "Amer-Indian-Eskimo", "Other", "Black"],
          9: ["Female", "Male"],
          13: ["United-States", "Cambodia", "England", "Puerto-Rico", "Canada", "Germany", "Outlying-US(Guam-USVI-etc)"


# Define the categorial features
c_features = [1, 3, 5, 6, 7, 8, 9, 13]

def convert_features(line):
    for i in values.keys():
        for j in range(len(values[i])):
            if line[i] == values[i][j]:
                line[i] = j
                break
    return line

# Adult contains nominal attributes, converts them to boolean
def generate_adult_data():
    X = []
    y = []
    with open(adult, 'r') as file:
        for line in file:
            words = line.split(',')
            words = map(str.strip, words)
            # If there's a missing value in the data, ignore the line
            if '?' in words:
                continue
            # Reading the values in as a dictionary
            features = convert_features(words[:-1])
            X.append(features)

            # Reading in the last value as a label (-1 og 1)
            if words[-1] == '<=50K':
                y.append(-1)
            else:
                y.append(1)

        file.close()

    X = np.array(X, dtype='int64')
    y = np.array(y, dtype='int64')

    # Encode the categorical features into integers
    from sklearn.preprocessing import OneHotEncoder
    enc = OneHotEncoder(categorical_features = c_features)
    X = enc.fit_transform(X).toarray()
    return X, y

def main():
    X, y = generate_adult_data()
    save_generated_data(X, y, "adult_generated")

if __name__ == "__main__":
    main()
```

Covtype data-generation - code:

```
import numpy as np
from utils import save_generated_data
from collections import defaultdict

# Covertype
covtype = "datasets/covertype/covtype.data"
covtype_debug = "datasets/covertype/covtype_debug.data"

# Convert cov_type into a binary problem with largest class as positive and the rest as negative
def generate_covtype_data():
    # Create a root node for the tree
    X = []
    y = []
    with open(covtype, 'r') as file:
        i = 0
        for line in file:
            words = line.split(',')
            words = map(str.strip, words)
            X.append(words[:-1])
            y.append(words[-1])
        file.close()
    X = np.array(X, dtype='int64')
    y = np.array(y, dtype='int64')
    return X, y

def largest_class(y):
    d = defaultdict(int)
    for i in y:
        d[i] += 1
    result = max(d.iteritems(), key=lambda x: x[1])
    for i in range(len(y)):
        if y[i] == (result[0]):
            y[i] = 1
        else:
            y[i] = -1

    print(result[0])

    return y

def main():
    X, y = generate_covtype_data()
    y = largest_class(y) # largest class as the positive and the rest as negative
    save_generated_data(X, y, "covtype_generated")

if __name__ == "__main__":
    main()
```

Letter data-generation - code:

```
import numpy as np
from utils import save_generated_data

# Letter Recognition
letter = "datasets/letter-recognition/letter-recognition.data"
letter_debug = "datasets/letter-recognition/letter-recognition-debug.data"

# Letter uses letters A-M as positives and the rest as negatives
A_TO_M = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J','K','L','M']

def generate_letter_data():
    X = []
    y_p1 = []
    y_p2 = []
    with open(letter, 'r') as file:
        for line in file:
            words = line.split(',')
            words = map(str.strip, words)
            # Reading the values in as a dictionary
            X.append(words[1:])

            # Treats O as positive and rest as negative, gives unbalanced set
            if words[0] == 'O':
                y_p1.append(1)
            else:
                y_p1.append(-1)

            # Treats A-M as positive and rest as negative, gives a balanced set
            if words[0] in A_TO_M :
                y_p2.append(1)
            else:
                y_p2.append(-1)

        file.close()

    X = np.array(X, dtype='int64')
```

```
    y_p1 = np.array(y_p1, dtype='int64')
    y_p2 = np.array(y_p2, dtype='int64')
    return X, y_p1, y_p2

def main():
    X, y_p1, y_p2 = generate_letter_data()
    save_generated_data(X, y_p1, "letterp1_generated")
    save_generated_data(X, y_p2, "letterp2_generated")

if __name__ == "__main__":
    main()
```