Machine Learning – Multiclass Classifiers – Report

Shlomi Ben-Shushan

General

In this report I'll describe the implementation of K-Nearest-Neighbors, Perceptron, SVM and Passive Aggressive algorithms and I'll explain how I found the most efficient hyper-parameters that able the algorithms to predict the test labels with high accuracy.

Parameter Finder Helper Program

In order to find the best parameters for each algorithm, I programmed another program named $ex2\_param\_finder.py$ that separate the training-set to a new training-set and a test-set which its labels are known (because they were part of the original training-set).

```
def main():
    objects, labels = shuffle(np.genfromtxt(sys.argv[1], delimiter=","), np.genfromtxt(sys.argv[2], dtype=int))
    print("Running tests...\n")
    find_params(objects, labels, "Best params without normalization:")
    find_params(z_score_normalize(objects), labels, "Best params with Z-Score normalization:")
    find_params(min_max_normalize(objects), labels, "Best params with Min-Max normalization:")
    print("Done!\n")
```

The code above is the main function of the parameter finder program which gets the training examples, shuffle them to avoid unwanted deviations, and look for the best parameters using the function $find\_params()$ with different types of normalization methods.

Each $find\_param()$ function looks for the best parameters for each algorithm on the given data, whether it is normalized with Z-Score, with Min-Max or it is not normalized at all. After the parameters were found, it prints them to the screen with the accuracy they yield.

The following code demonstrates that:

```
def find_params(objects, labels, title):
    print(title)
    a = calibrate_knn(objects, labels)
    print(" * KNN: K=" + str(a[0]) + " => accuracy=" + str(a[1]))
    b = calibrate_perceptron(objects, labels)
    print(" * Perceptron: epochs=" + str(b[0]) + " eta=" + str(b[1]) + " => accuracy=" + str(b[2]))
    c = calibrate_svm(objects, labels)
    print(" * SVM: epochs=" + str(c[0]) + " eta=" + str(c[1]) + " lambda=" + str(c[2]) + " => accuracy=" + str(c[3]))
    d = calibrate_pa(objects, labels)
    print(" * PA: epochs=" + str(d[0]) + " => accuracy=" + str(d[1]))
    print()
```

Each $calibrate\_algo()$ function runs the algorithm multiple times using **cross validation** and with different potential parameters in order to find the best ones.

$calibrate\_algo(objects, labels)$ Pseudo-Code:

```
current_max = 0
best = None
for different parameters:
    accuracy = cross_validation(objects, labels, algo, [parameters])
    if current_max < accuracy:
        current_max = accuracy
        best = ([parameters], accuracy)
return best
```

Cross Validation:

The cross validation method works as follows:

1. Get a training-set of objects (features vectors) and correspondent labels.
2. Get an algoithm and its parameters.
3. Divide the training set to k chunks (I devided it to $k = 5$ chunks).
4. For k times:
    a. Determine one chunk as a new test-set an the rest as a new train-set. Each of the new train and test sets are consists of objects and correspondent labels, which means that now we know the labels of the new test objects.
    b. Run the given algorithm with its parameters on the new train-data and the new test objects, and get its predictions.
    c. Compare predictions to the real test labels an calculate accuracy.

The following code describes how did I create the new train and test sets:

```
def create_cross_validation_sets(objects, labels):
    train_x, train_y, test_x, test_y = [], [], [], []
    chunk_size = int(0.2 * len(objects))
    objects_chunks = [objects[i:i + chunk_size] for i in range(len(objects))[::chunk_size]]
    labels_chunks = [labels[i:i + chunk_size] for i in range(len(labels))[::chunk_size]]
    for i in range(5):
        a = i
        b = (i + 1) % 5
        c = (i + 2) % 5
        d = (i + 3) % 5
        e = (i + 4) % 5
        train_x.append(np.concatenate([objects_chunks[a], objects_chunks[b], objects_chunks[c], objects_chunks[d]], axis=0))
        train_y.append(np.concatenate([labels_chunks[a], labels_chunks[b], labels_chunks[c], labels_chunks[d]], axis=0))
        test_x.append(objects_chunks[e])
        test_y.append(labels_chunks[e])
    return train_x, train_y, test_x, test_y
```

Note that it returns sets of k train-sets and test-sets ("sets of sets").

The following code demonstrates how I have implemented the cross validation method as described above:

```
def cross_validation(objects, labels, algo, params):
    train_x, train_y, test_x, test_y = create_cross_validation_sets(objects, labels)
    ratios = []
    for i in range(5):
        if len(params) == 1:
            predictions = algo(train_x[i], train_y[i], params[0], test_x[i])
        elif len(params) == 2:
            predictions = algo(train_x[i], train_y[i], params[0], params[1], test_x[i])
        elif len(params) == 3:
            predictions = algo(train_x[i], train_y[i], params[0], params[1], params[2], test_x[i])
        else:
            raise "Wrong number of params."
        hits = count_hits(predictions, test_y[i])
        ratios.append(round(hits / len(predictions), 4))
    return ratios, round(np.average(ratios), 4)
```

Note that I have tried different division options, means different k's.

The Results:

```
Best params without normalization:
 * KNN: K=5 => accuracy=0.9792
 * Perceptron: epochs=100 eta=0.2 => accuracy=0.6083
 * SVM: epochs=140 eta=0.4 lambda=0.9  => accuracy=0.7125
 * PA: epochs=830 => accuracy=0.7083

Best params with Z-Score normalization:
 * KNN: K=5 => accuracy=0.925
 * Perceptron: epochs=90 eta=0.2 => accuracy=0.8125
 * SVM: epochs=80 eta=0.1 lambda=0.8  => accuracy=0.8208
 * PA: epochs=190 => accuracy=0.7875

Best params with Min-Max normalization:
 * KNN: K=9 => accuracy=0.9292
 * Perceptron: epochs=150 eta=0.3 => accuracy=0.7333
 * SVM: epochs=150= eta=0.3 lambda=0.7  => accuracy=0.775
 * PA: epochs=650 => accuracy=0.6833

Done!
```

As we can see, the best hyper-parameters found with Z-Score normalization, except from KNN that seems to perform better without normalization at all.

K-Nearest-Neighbors Implementation Details

Explanation: KNN's implementation was based on the following logic. For every test-object in the test-set, find the k train-objects (from the training-set) closest to it and determine its test-label as the label most common among the train-labels correspondent to the train-objects.

Chosen Hyper-Parameters: $k = 5$.

Helper Code:

```
def calibrate_knn(objects, labels):
    current_max = 0
    best = None
    for k in range(2, int(len(objects) / 2)):
        results, average = cross_validation(objects, labels, knn, [k])
        if current_max < average:
            current_max = average
            best = (k, average)
            # print("Current best: k=" + str(k) + "=> accuracy=" + str(average))
    return best
```

Perceptron Implementation Details

Explanation: Perceptron's implementation was based on the following logic. For $epochs$ iterations, calculate $\hat{y}$, and if $\hat{y} \neq y$, then the algorithm missed so update w by calculating $\eta \cdot x$ to add it to w^{y} and subtract it from $w^{\hat{y}}$, and then divide eta by 2 to reduce the learning rate.

Chosen Hyper-Parameters: $epochs = 90, \quad \eta = 0.2$.

Helper Code:

```
def calibrate_perceptron(objects, labels):
    current_max = 0
    best = None
    for epochs in [e for e in range(10, 160, 10)]:
        eta = round(0.1, 4)
        while eta <= 1.0:
            results, average = cross_validation(objects, labels, perceptron, [epochs, eta])
            if current_max < average:
                current_max = average
                best = (epochs, eta, average)
                # print("Current best: epochs=" + str(epochs) + " eta=" + str(eta) + "=> ac
            eta = round(eta + 0.1, 4)
    return best
```

SVM Implementation Details

Explanation: SVM's implementation was based on the following logic. For $epochs$ iterations, calculate $\hat{y}$, and if $\hat{y} \neq y$, then the algorithm missed so update w according to the instructions, and then divide η and λ by 2 to reduce the learning rate and the regulation.

Chosen Hyper-Parameters: $epochs = 80, \quad \eta = 0.1, \quad \lambda = 0.8$.

Helper Code:

```
def calibrate_svm(objects, labels):
    current_max = 0
    best = None
    for epochs in [e for e in range(10, 160, 10)]:
        eta = round(0.1, 4)
        while eta <= 1.0:
            lamb = round(0.1, 4)
            while lamb <= 1.0:
                results, average = cross_validation(objects, labels, svm, [epochs, eta, lamb])
                if current_max < average:
                    current_max = average
                    best = (epochs, eta, lamb, average)
                    # print("Current best: epochs=" + str(epochs) + " eta=" + str(eta) + " lamb
                lamb = round(lamb + 0.1, 4)
            eta = round(eta + 0.1, 4)
    return best
```

Passive Aggressive Implementation Details

Explanation: PA's implementation was based on the following logic. For $epochs$ iterations, calculate y_{hat} and if $y_{hat} \neq y$, then the algorithm missed so update w by calculating τ as $\frac{loss(w,x,y)}{2\|x\|^2}$ in order to add it to w^y and subtract it from $w^{y_{hat}}$.

Chosen Hyper-Parameters: $epochs = 190$.

Helper Code:

```
def calibrate_pa(objects, labels):
    current_max = 0
    best = None
    for epochs in [e for e in range(10, 1010, 10)]:
        results, average = cross_validation(objects, labels, passive_aggressive, [epochs])
        if current_max < average:
            current_max = average
            best = (epochs, average)
            # print("Current best: epochs=" + str(epochs) + "=> accuracy=" + str(average))
    return best
```

Important Notes:

- I have checked different number of potential hyper-parameter ranges, not only the ranges shown in the code snippets above.
- I ignored good results at low number of epochs in order to avoid underfitting.