tree-methods-sample-solution

July 3, 2023

Ullrich Köthe: Machine Learning Essentials, Summer Semester 2023

[1]:
```
# import modules
import numpy as np
from sklearn.datasets import load_digits
from sklearn.model_selection import KFold
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt
from abc import abstractmethod
```

1 Base Classes

[3]:
```
class Node:
    '''
      this class will later get the following attributes
      all nodes:
          features
          responses
      split nodes additionally:
          left
          right
          split_index
          threshold
      leaf nodes additionally
          prediction
    '''


class Tree:
    '''
      base class for RegressionTree and ClassificationTree
    '''
    def __init__(self, n_min=10):
        '''n_min: minimum required number of instances in leaf nodes
        '''
        self.n_min = n_min
```

```
    def predict(self, x):
        ''' return the prediction for the given 1-D feature vector x
        '''
        # first find the leaf containing the 1-D feature vector x
        node = self.root
        while not hasattr(node, "prediction"):
            j = node.split_index
            if x[j] <= node.threshold:
                node = node.left
            else:
                node = node.right
        # finally, return the leaf's prediction
        return node.prediction

    def train(self, features, responses, D_try=None):
        '''
        features: the feature matrix of the training set
        response: the vector of responses
        '''
        N, D = features.shape
        assert(responses.shape[0] == N)

        if D_try is None:
            D_try = int(np.sqrt(D)) # number of features to consider for each␣
↪split decision

        # initialize the root node
        self.root = Node()
        self.root.features  = features
        self.root.responses = responses

        # build the tree
        stack = [self.root]
        while len(stack):
            node = stack.pop()
            active_indices = self.select_active_indices(D, D_try)
            left, right = self.make_split_node(node, active_indices)
            if left is None: # no split found
                self.make_leaf_node(node)
            else:
                stack.append(left)
                stack.append(right)

    def make_split_node(self, node, indices):
        '''
        node: the node to be split
        indices: a numpy array of length 'D_try', containing the feature
```

```
                        indices to be considered for the present split

        return: None, None -- if no suitable split has been found, or
                left, right -- the children of the split
        '''
        # all responses equal => no improvement possible by any split
        if np.unique(node.responses).shape[0] == 1:
            return None, None

        # find best feature j_min (among 'indices') and best threshold t_min
↪for the split
        l_min = float('inf')  # upper bound for the loss, later the loss of the
↪best split
        j_min, t_min = None, None

        for j in indices:
            thresholds = self.find_thresholds(node, j)

            # compute loss for each threshold
            for t in thresholds:
                loss = self.compute_loss_for_split(node, j, t)

                # remember the best split so far
                # (the condition is never True when loss = float('inf') )
                if loss < l_min:
                    l_min = loss
                    j_min = j
                    t_min = t

        if j_min is None: # no split found
            return None, None

        # create children for the best split
        left, right = self.make_children(node, j_min, t_min)

        # turn the current 'node' into a split node
        # (store children and split condition)
        node.left = left
        node.right = right
        node.split_index = j_min
        node.threshold = t_min

        # return the children (to be placed on the stack)
        return left, right

    def select_active_indices(self, D, D_try):
```

```
        ''' return a 1-D array with D_try randomly selected indices from 0...
↪(D-1).
        '''
        return np.random.choice(range(D), size=D_try, replace=False)

    def find_thresholds(self, node, j):
        ''' return: a 1-D array with all possible thresholds along feature j
        '''
        if node.responses.shape[0] < 2 * self.n_min:
            # no split possible without having less than n_min instances in one␣
↪node
            return []
        # sort features
        sorted_features = np.sort(node.features[:,j])
        # calculate mean between features
        thresholds = 0.5 * (sorted_features[:-1] + sorted_features[1:])
        # remove thresholds with less than n_min instances in one node
        if self.n_min > 1:
            thresholds = thresholds[(self.n_min-1):(-self.n_min+1)]
        return thresholds

    def make_children(self, node, j, t):
        ''' execute the split in feature j at threshold t

            return: left, right -- the children of the split, with features and␣
↪responses
                                    properly assigned according to the split
        '''
        left = Node()
        right = Node()

        # select and store features for each child
        left_mask = node.features[:,j] <= t
        right_mask = np.logical_not(left_mask)
        left.features = node.features[left_mask]
        left.responses = node.responses[left_mask]
        right.features = node.features[right_mask]
        right.responses = node.responses[right_mask]

        return left, right

    @abstractmethod
    def make_leaf_node(self, node):
        ''' Turn node into a leaf by computing and setting `node.prediction`

            (must be implemented in a subclass)
        '''
```

```
        raise NotImplementedError("make_leaf_node() must be implemented in a␣
↪subclass.")

    @abstractmethod
    def compute_loss_for_split(self, node, j, t):
        ''' Return the resulting loss when the data are split along feature j␣
↪at threshold t.
            If the split is not admissible, return float('inf').

            (must be implemented in a subclass)
        '''
        raise NotImplementedError("compute_loss_for_split() must be implemented␣
↪in a subclass.")
```

2 Regression Tree

[4]:
```
class RegressionTree(Tree):
    def __init__(self, n_min=10):
        super(RegressionTree, self).__init__(n_min)

    def compute_loss_for_split(self, node, j, t):
        # return the loss if we would split the instance along feature j at␣
↪threshold t
        # or float('inf') if there is no feasible split

        # select the respective responses
        resp_left = node.responses[node.features[:,j] <= t]
        resp_right = node.responses[node.features[:,j] > t]
        # calculate the means
        y_left = np.mean(resp_left)
        y_right = np.mean(resp_right)
        # calculate the squared loss
        loss = np.sum((resp_left - y_left)**2) + np.sum((resp_right -␣
↪y_right)**2)
        return loss

    def make_leaf_node(self, node):
        # turn node into a leaf node by computing `node.prediction`
        # (note: the prediction of a regression tree is a real number)
        node.prediction = np.mean(node.responses)
```

3 Classification Tree

[5]:
```
class ClassificationTree(Tree):
    '''implement classification tree so that it can handle arbitrary many␣
↪classes
    '''

    def __init__(self, classes, n_min=10):
        ''' classes: a 1-D array with the permitted class labels
            n_min: minimum required number of instances in leaf nodes
        '''
        super(ClassificationTree, self).__init__(n_min)
        self.classes = classes

    def compute_loss_for_split(self, node, j, t):
        # return the loss if we would split the instance along feature j at␣
↪threshold t
        # or float('inf') if there is no feasible split

        # select the respective responses
        resp_left = node.responses[node.features[:,j] <= t]
        resp_right = node.responses[node.features[:,j] > t]
        # calculate the probabilities
        p_left = [np.mean(resp_left == k) for k in self.classes]
        p_right = [np.mean(resp_right == k) for k in self.classes]

        N_left = resp_left.shape[0]
        N_right = resp_right.shape[0]
        C = len(self.classes)
        # calculate the entropy loss (put in zero if p=0 following L'Hôpital's␣
↪rule, see lecture)
        loss = N_left * np.sum([-p_left[i]*np.log(p_left[i]) for i in range(C)␣
↪if p_left[i] != 0]) + \
               N_right * np.sum([-p_right[i]*np.log(p_right[i]) for i in␣
↪range(C) if p_right[i] != 0])
        return loss

    def make_leaf_node(self, node):
        # turn node into a leaf node by computing `node.prediction`
        # (note: the prediction of a classification tree is a class label)
        p = [np.mean(node.responses == k) for k in self.classes]
        # hard label using argmax
        node.prediction = self.classes[np.argmax(p)]
```

4 Evaluation of Regression and Classification Tree

[6]:
```
# read and prepare the digits data and extract 3s and 9s
digits = load_digits()
print(digits.data.shape, digits.target.shape)

instances = (digits.target == 3) | (digits.target == 9)
features = digits.data[instances, :]
labels = digits.target[instances]

# for regression, we use labels +1 and -1
responses = np.array([1 if l == 3 else -1 for l in labels])

assert(features.shape[0] == labels.shape[0] == responses.shape[0])
```

```
(1797, 64) (1797,)
```

[20]:
```
# perform 5-fold cross-validation (see ex01) with responses +1 and -1 (for 3s␣
 ↪and 9s)
# using RegressionTree()
# and comment on your results
kf = KFold()

train_errors = []
test_errors = []
for i, (train_index, test_index) in enumerate(kf.split(features)):
    print(f"Fold {i+1}")
    rt = RegressionTree(5)
    rt.train(features[train_index], responses[train_index])
    predictions_train = np.sign([rt.predict(features[index]) for index in␣
 ↪train_index])
    predictions_test = np.sign([rt.predict(features[index]) for index in␣
 ↪test_index])
    train_errors.append(np.mean(predictions_train != responses[train_index]))
    test_errors.append(np.mean(predictions_test != responses[test_index]))
    print(f"Train error: {train_errors[-1]:.2f}", end=", ")
    print(f"Test error: {test_errors[-1]:.2f}")
print(f"Mean train error: {np.mean(train_errors):.2f}")
print(f"Mean test error: {np.mean(test_errors):.2f}")
```

```
Fold 1
Train error: 0.08, Test error: 0.07
Fold 2
Train error: 0.06, Test error: 0.27
Fold 3
Train error: 0.06, Test error: 0.10
Fold 4
```

```
Train error: 0.04, Test error: 0.14
Fold 5
Train error: 0.04, Test error: 0.17
Mean train error: 0.06
Mean test error: 0.15
```

[21]:
```python
# perform 5-fold cross-validation with labels 3 and 9
# using ClassificationTree(classes=np.unique(labels))
# and comment on your results

train_errors = []
test_errors = []
for i, (train_index, test_index) in enumerate(kf.split(features)):
    print(f"Fold {i+1}")
    ct = ClassificationTree(classes=np.unique(labels), n_min=5)
    ct.train(features[train_index], labels[train_index])
    predictions_train = np.array([ct.predict(features[index]) for index in␣
 ↪train_index])
    predictions_test = np.array([ct.predict(features[index]) for index in␣
 ↪test_index])
    train_errors.append(np.mean(predictions_train != labels[train_index]))
    test_errors.append(np.mean(predictions_test != labels[test_index]))
    print(f"Train error: {train_errors[-1]:.2f}", end=", ")
    print(f"Test error: {test_errors[-1]:.2f}")
print(f"Mean train error: {np.mean(train_errors):.2f}")
print(f"Mean test error: {np.mean(test_errors):.2f}")
```

```
Fold 1
Train error: 0.04, Test error: 0.12
Fold 2
Train error: 0.02, Test error: 0.30
Fold 3
Train error: 0.04, Test error: 0.14
Fold 4
Train error: 0.03, Test error: 0.10
Fold 5
Train error: 0.06, Test error: 0.26
Mean train error: 0.04
Mean test error: 0.18
```

Both methods perform very similar. The results are not great, we see a lot of overfitting and a high variance in the test error.

5 Regression and Classification Forest

[14]:
```
def bootstrap_sampling(features, responses):
    '''return a bootstrap sample of features and responses
    '''
    N = features.shape[0]
    # sampling indices with replacement
    sample_inds = np.random.choice(range(N), size=N, replace=True)
    return features[sample_inds], responses[sample_inds]
```

[15]:
```
class RegressionForest():
    def __init__(self, n_trees, n_min=10):
        # create ensemble
        self.trees = [RegressionTree(n_min) for i in range(n_trees)]

    def train(self, features, responses):
        for tree in self.trees:
            boostrap_features, bootstrap_responses =␣
 ↪bootstrap_sampling(features, responses)
            tree.train(boostrap_features, bootstrap_responses)

    def predict(self, x):
        # compute the response of the ensemble from the individual responses␣
 ↪and return it
        # ensemble mean
        return np.mean([tree.predict(x) for tree in self.trees])
```

[16]:
```
class ClassificationForest():
    def __init__(self, n_trees, classes, n_min=1):
        self.trees = [ClassificationTree(classes, n_min) for i in␣
 ↪range(n_trees)]
        self.classes = classes

    def train(self, features, labels):
        for tree in self.trees:
            boostrap_features, bootstrap_labels = bootstrap_sampling(features,␣
 ↪labels)
            tree.train(boostrap_features, bootstrap_labels)

    def predict(self, x):
        # compute the response of the ensemble from the individual responses␣
 ↪and return it
        # majority vote
        predictions = np.array([tree.predict(x) for tree in self.trees])
        votes = [np.sum(predictions == k) for k in self.classes]
        return self.classes[np.argmax(votes)]
```

6 Evaluation of Regression and Decision Forest

[22]:
```
# perform 5-fold cross-validation (see ex01) with responses +1 and -1 (for 3s␣
 ↪and 9s)
# using RegressionForest(n_trees=10)
# and comment on your results

train_errors = []
test_errors = []
for i, (train_index, test_index) in enumerate(kf.split(features)):
    print(f"Fold {i+1}")
    rf = RegressionForest(n_trees=10, n_min=1)
    rf.train(features[train_index], responses[train_index])
    predictions_train = np.sign([rf.predict(features[index]) for index in␣
 ↪train_index])
    predictions_test = np.sign([rf.predict(features[index]) for index in␣
 ↪test_index])
    train_errors.append(np.mean(predictions_train != responses[train_index]))
    test_errors.append(np.mean(predictions_test != responses[test_index]))
    print(f"Train error: {train_errors[-1]:.2f}", end=", ")
    print(f"Test error: {test_errors[-1]:.2f}")
print(f"Mean train error: {np.mean(train_errors):.2f}")
print(f"Mean test error: {np.mean(test_errors):.2f}")
```

```
Fold 1
Train error: 0.00, Test error: 0.08
Fold 2
Train error: 0.00, Test error: 0.18
Fold 3
Train error: 0.00, Test error: 0.03
Fold 4
Train error: 0.00, Test error: 0.04
Fold 5
Train error: 0.00, Test error: 0.14
Mean train error: 0.00
Mean test error: 0.09
```

[23]:
```
# perform 5-fold cross-validation with labels 3 and 9
# using DecisionForest(n_trees=10, classes=np.unique(labels))
# and comment on your results
train_errors = []
test_errors = []
for i, (train_index, test_index) in enumerate(kf.split(features)):
    print(f"Fold {i+1}")
    cf = ClassificationForest(n_trees=10, classes=np.unique(labels), n_min=1)
    cf.train(features[train_index], labels[train_index])
```

```
    predictions_train = np.array([cf.predict(features[index]) for index in␣
 ↪train_index])
    predictions_test = np.array([cf.predict(features[index]) for index in␣
 ↪test_index])
    train_errors.append(np.mean(predictions_train != labels[train_index]))
    test_errors.append(np.mean(predictions_test != labels[test_index]))
    print(f"Train error: {train_errors[-1]:.2f}", end=", ")
    print(f"Test error: {test_errors[-1]:.2f}")
print(f"Mean train error: {np.mean(train_errors):.2f}")
print(f"Mean test error: {np.mean(test_errors):.2f}")
```

```
Fold 1
Train error: 0.00, Test error: 0.05
Fold 2
Train error: 0.00, Test error: 0.14
Fold 3
Train error: 0.00, Test error: 0.03
Fold 4
Train error: 0.00, Test error: 0.01
Fold 5
Train error: 0.00, Test error: 0.04
Mean train error: 0.00
Mean test error: 0.05
```

As `n_min=1` was used, the training data is learned perfectly. We see that the test error is lower than before nonetheless, indicating that using a forest decreases overfitting.

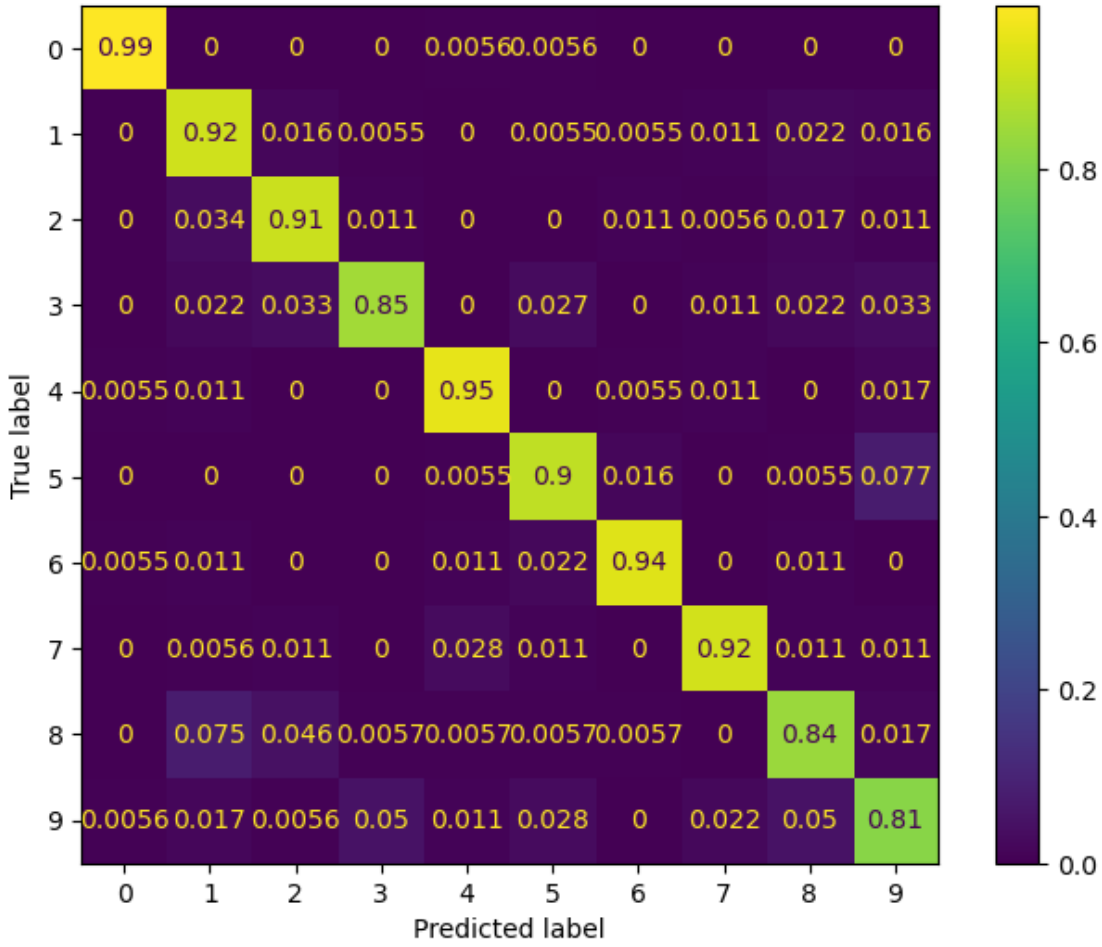
7 Multi-class Classification Forest

[24]:
```
# Train DecisionForest(n_trees=10, classes=np.unique(digits.target))
# for all 10 digits simultaneously.
# Compute and plot the confusion matrix after 5-fold cross-validation and␣
 ↪comment on your results.
train_errors = []
test_errors = []

mccf_pred = np.array([])
mccf_true = np.array([])
for i, (train_index, test_index) in enumerate(kf.split(digits.data)):
    print(f"Fold {i+1}")
    mccf = ClassificationForest(n_trees=10, classes=np.unique(digits.target),␣
 ↪n_min=1)
    mccf.train(digits.data[train_index], digits.target[train_index])
    predictions_train = np.array([mccf.predict(digits.data[index]) for index in␣
 ↪train_index])
    predictions_test = np.array([mccf.predict(digits.data[index]) for index in␣
 ↪test_index])
```

```
    # store predictions and true values to create confusion matrix
    mccf_pred = np.append(mccf_pred, predictions_test)
    mccf_true = np.append(mccf_true, digits.target[test_index])
    train_errors.append(np.mean(predictions_train != digits.
 ↪target[train_index]))
    test_errors.append(np.mean(predictions_test != digits.target[test_index]))
    print(f"Train error: {train_errors[-1]:.2f}", end=", ")
    print(f"Test error: {test_errors[-1]:.2f}")
print(f"Mean train error: {np.mean(train_errors):.2f}")
print(f"Mean test error: {np.mean(test_errors):.2f}")
```

```
Fold 1
Train error: 0.00, Test error: 0.07
Fold 2
Train error: 0.00, Test error: 0.16
Fold 3
Train error: 0.00, Test error: 0.08
Fold 4
Train error: 0.00, Test error: 0.06
Fold 5
Train error: 0.00, Test error: 0.11
Mean train error: 0.00
Mean test error: 0.10
```

[25]:
```
conf_matrix = confusion_matrix(mccf_true, mccf_pred, normalize='true')
fig, ax = plt.subplots(figsize=(8,6))

display = ConfusionMatrixDisplay(conf_matrix, display_labels=mccf.classes)
_ = display.plot(ax=ax)
```



The multi-class classification forest performs ok, but would not be reliable enough for use in production.

8 One-against-the-rest classification with RegressionForest

[26]:
```
def bootstrap_sampling_one_against_rest(features, responses, k):
    '''return a bootstrap sample of features and responses, balanced for 'one'␣
 ↪and 'rest
    '''
    N = features.shape[0]
    # select target responses
    one_mask = responses == k
    features_one = features[one_mask]
    responses_one = responses[one_mask]
    # select rest
    rest_mask = responses != k
```

```
    features_rest = features[rest_mask]
    responses_rest = responses[rest_mask]
    # number of responses for k
    N_k = responses_one.shape[0]
    # We only use 2 * N_k elements to save computation time.
    # Using more might improve performance
    N_per_group = N_k

    sample_inds_one = np.random.choice(range(N_k), size=N_per_group,␣
 ↪replace=True)
    sample_inds_rest = np.random.choice(range(N-N_k), size=N_per_group,␣
 ↪replace=True)
    # use 1 as label for k and -1 as label for rest, concatenate
    return np.concatenate([features_one[sample_inds_one],
                           features_rest[sample_inds_rest]]),\
           np.concatenate([np.repeat(1, N_per_group), np.repeat(-1,␣
 ↪N_per_group)])
```

We specify a new class that takes the class label and uses the new bootstrap sampling function.

[27]:
```
class OneAgainstRestRegressionForest():
    def __init__(self, n_trees, k, n_min=10):
        # create ensemble
        self.k = k
        self.trees = [RegressionTree(n_min) for i in range(n_trees)]

    def train(self, features, responses):
        for tree in self.trees:
            boostrap_features, bootstrap_responses = \
                bootstrap_sampling_one_against_rest(features, responses, self.k)
            tree.train(boostrap_features, bootstrap_responses)

    def predict(self, x):
        # compute the response of the ensemble from the individual responses␣
 ↪and return it
        return np.mean([tree.predict(x) for tree in self.trees])
```

[28]:
```
# Train ten one-against-the-rest regression forests for the 10 digits.
# Make sure that all training sets are balanced between the current digit and␣
 ↪the rest.
# Assign test instances to the digit with highest score,
# or to "unknown" if all scores are negative.
# Compute and plot the confusion matrix after 5-fold cross-validation and␣
 ↪comment on your results.
test_errors = []

mcrf_pred = np.array([])
```

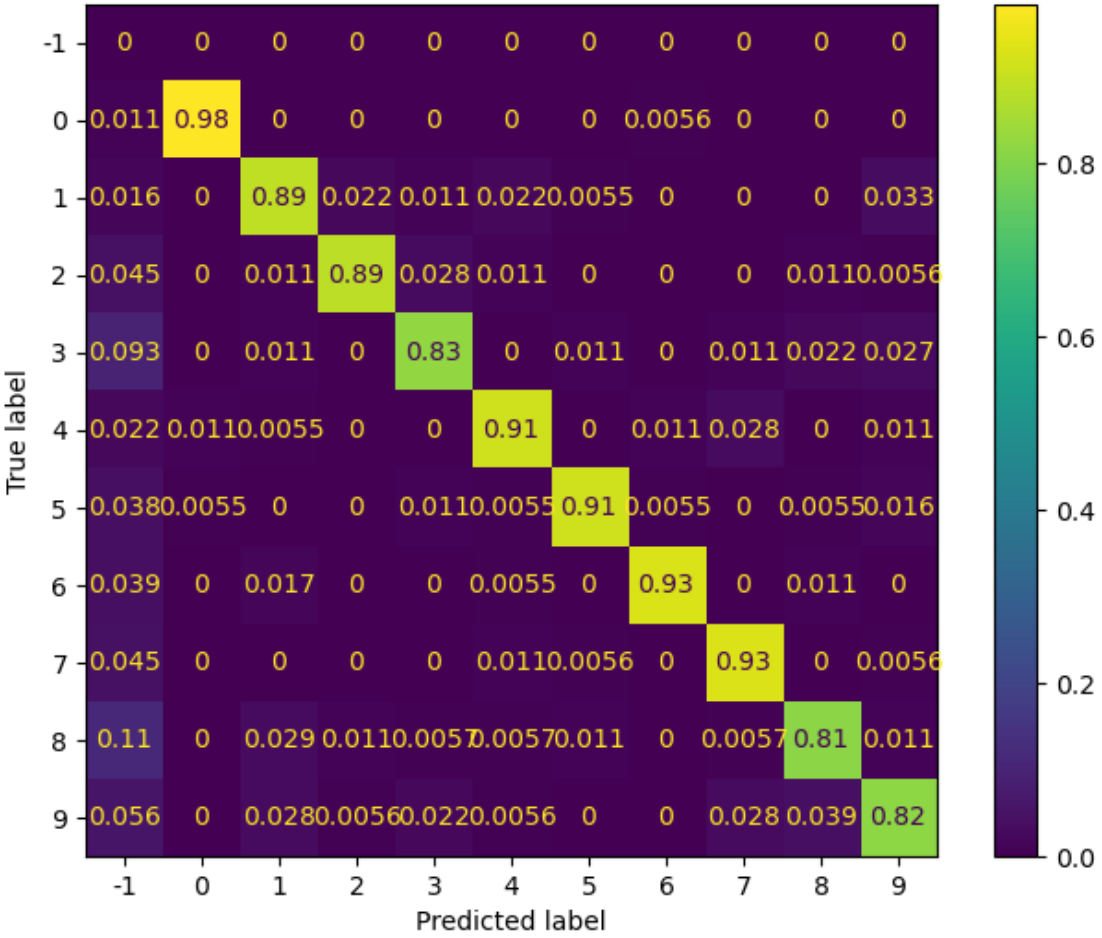
```
mcrf_true = np.array([])
classes = np.unique(digits.target)
for i, (train_index, test_index) in enumerate(kf.split(digits.data)):
    print(f"Fold {i+1}")
    # create one forest for each digit
    forests = [OneAgainstRestRegressionForest(n_trees=10, k=k, n_min=1)
               for k in classes]
    # train all forests
    for forest in forests:
        forest.train(digits.data[train_index], digits.target[train_index])
    # Compute predictions ('unknown' -> -1)
    predictions_test = []
    for index in test_index:
        scores = [forest.predict(digits.data[index]) for forest in forests]
        max_score_ind = np.argmax(scores)
        if scores[max_score_ind] < 0:
            # unknown coded as -1
            predictions_test.append(-1)
        else:
            predictions_test.append(classes[max_score_ind])
    predictions_test = np.array(predictions_test)
    # store predictions and true values to create confusion matrix
    mcrf_pred = np.append(mcrf_pred, predictions_test)
    mcrf_true = np.append(mcrf_true, digits.target[test_index])
    test_errors.append(np.mean(predictions_test != digits.target[test_index]))
    print(f"Test error: {test_errors[-1]:.2f}")
print(f"Mean test error: {np.mean(test_errors):.2f}")
```

```
Fold 1
```

```
/home/valentin/anaconda3/lib/python3.9/site-
packages/numpy/core/fromnumeric.py:3464: RuntimeWarning: Mean of empty slice.
  return _methods._mean(a, axis=axis, dtype=dtype,
/home/valentin/anaconda3/lib/python3.9/site-packages/numpy/core/_methods.py:192:
RuntimeWarning: invalid value encountered in scalar divide
  ret = ret.dtype.type(ret / rcount)
```

```
Test error: 0.09
Fold 2
Test error: 0.16
Fold 3
Test error: 0.12
Fold 4
Test error: 0.07
Fold 5
Test error: 0.12
Mean test error: 0.11
```

[29]:
```
conf_matrix = confusion_matrix(mcrf_true, mcrf_pred, normalize='true')
fig, ax = plt.subplots(figsize=(8,6))

display = ConfusionMatrixDisplay(conf_matrix, display_labels=np.insert(classes,␣
↪0, -1))
_ = display.plot(ax=ax)
```



We can see that the Multi-class Regression Forest doesn't perform that well. The decrease in accuracy compred to the multi-class classification forest mainly comes from the "unknown" label, which depending on the class absorbs some predictions that could have been correct.