hw7

January 11, 2021

[16]:
```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
from sklearn.model_selection import train_test_split
```

[17]:
```
![ -e spambase.data ] || wget 'https://archive.ics.uci.edu/ml/
↪machine-learning-databases/spambase/spambase.data'
![ -e spambase.names ] || wget 'https://archive.ics.uci.edu/ml/
↪machine-learning-databases/spambase/spambase.names'
```

[18]:
```
data = np.array(pd.read_csv('spambase.data', header=None))

X = data[:,:-1] # features
y = data[:,-1] # Last column is label

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0,␣
↪shuffle=True, stratify=y)
```

[55]:
```
from collections import Counter

def neighbor_means(it):
  first = True
  for i in it:
    if not first:
      yield (i + prev) / 2
    prev = i
    first = False

def mode(it):
  return Counter(it).most_common(1)[0][0]

class Leaf():
  def __init__(self, value=None):
    self.value = value

  def predict(self, X):
    return np.full(X.shape[0], self.value)
```

```python
  def fit(self, X, y, weights):
    self.value = mode(y)

class BinaryNode():
  def __init__(self, stops, min_leaf_size, min_importance, max_depth):
    self.stops = stops
    self.min_leaf_size = min_leaf_size
    self.min_importance = min_importance
    self.max_depth = max_depth

    self.feature = None
    self.stop = None
    self.left = None
    self.right = None

  def loss_function(self, y, weights):
    if len(y) == 0:
      return float('inf')
    return ((y != mode(y)) @ weights).mean()

  def split_condition(self, X):
    return X[:, self.feature] <= self.stop

  def importance(self, X, y, weights):
    cond = self.split_condition(X)
    return self.loss_function(y, weights) - self.loss_function(y[cond], 
↪weights[cond]) - self.loss_function(y[~cond], weights[~cond])

  def list_possible(self):
    for j, fstops in enumerate(self.stops):
      for stop in fstops:
        yield j, stop

  def fit(self, X, y, weights):
    min_loss = float('inf')
    for feature, stop in self.list_possible():
      cond = X[:, feature] <= stop
      cur_loss = self.loss_function(y[cond], weights[cond]) + self.
↪loss_function(y[~cond], weights[~cond])
      if cur_loss <= min_loss:
        min_loss = cur_loss
        self.feature = feature
        self.stop = stop
    if self.feature is None:
      node = self.make_leaf()
      node.fit(X, y, weights)
```

```
#       print("Cannot split")
#       self.__dict__ = node.__dict__  # dark magic
      self.predict = node.predict
      return
    cond = X[:, self.feature] <= self.stop
    not_cond = ~cond
    self.saved_importance = self.importance(X, y, weights)
    self.left = self.create_child(X[cond], y[cond], weights[cond])
    self.right = self.create_child(X[not_cond], y[not_cond], weights[not_cond])

  def make_leaf(self):
    return Leaf()

  def make_non_leaf(self):
    return BinaryNode(self.stops, self.min_leaf_size, self.min_importance, self.
↪max_depth - 1)

  def create_child(self, X, y, weights):
    if self.max_depth == 0 or len(y) < self.min_leaf_size or self.feature is␣
↪not None and self.saved_importance < self.min_importance:
      child = self.make_leaf()
    else:
      child = self.make_non_leaf()
    child.fit(X, y, weights)
    return child

  def predict(self, X):
    cond = X[:, self.feature] <= self.stop
    not_cond = ~cond
    y_pred = np.zeros(X.shape[0])
    y_pred[cond] = self.left.predict(X[cond])
    y_pred[not_cond] = self.right.predict(X[not_cond])
    return y_pred


class DecisionTreeClassifier():
  def __init__(self, min_leaf_size=4, min_importance=0.001, max_depth=-1):
    self.min_leaf_size = min_leaf_size
    self.min_importance = min_importance
    self.max_depth = max_depth

  def fit(self, X, y, weights=None):
    n_features = X.shape[1]
    if weights is None:
        weights = np.ones(n_features)
    feature_stops = tuple(tuple(neighbor_means(sorted(set(X[:, i])))) for i in␣
↪range(n_features))
```

```python
        fake = BinaryNode(feature_stops, self.min_leaf_size, self.min_importance,␣
↪self.max_depth)
        self.root = fake.create_child(X, y, weights)

    def predict(self, X):
        return self.root.predict(X)
```

[53]:
```python
class DecisionStumpClassifier(DecisionTreeClassifier):
    def __init__(self):
        DecisionTreeClassifier.__init__(self, 1, 0, 1)
```

[107]:
```python
class AdaBoostClassifier():
    def __init__(self, models, positive_class=1, negative_class=0):
        self.models = tuple(models)
        self.positive_class = positive_class
        self.negative_class = negative_class

    def class_to_sign(self, y):
        y_sgn = np.ones_like(y)
        y_sgn[y == self.negative_class] = -1
        return y_sgn

    def sign_to_class(self, y_sgn):
        y = np.full_like(y_sgn, self.positive_class)
        y[y_sgn < 0] = self.negative_class
        return y

    def fit(self, X, y):
        n_samples = X.shape[0]
        self.n_samples = n_samples
        y_sgn = self.class_to_sign(y)
        weights = np.full(n_samples, 1 / n_samples)
        self.alphas = []
        for m in self.models:
            m.fit(X, y, weights=weights)
            y_hat = m.predict(X)
            e = weights @ (y != y_hat)
            alpha = float(np.log((1 - e) / e))
            alpha = np.nan_to_num(alpha, True, 0.0, 700.0, -700.0)
            self.alphas.append(alpha)
            weights *= np.exp(alpha * y_sgn * self.class_to_sign(y_hat))
            weights = np.nan_to_num(weights, False, 1.0)
            weights /= weights.sum()
        self.alphas = np.array(self.alphas)

    def predict(self, X):
```

```
        return self.sign_to_class(sum((a * self.class_to_sign(m.predict(X)) for␣
↪a, m in zip(self.alphas, self.models))))
```

[22]:
```
from collections import namedtuple
from functools import lru_cache

class Confusion(namedtuple('Confusion', ['TP', 'FP', 'FN', 'TN'])):
    def calculate(y_true, y_pred):
        y_true = y_true == True
        y_pred = y_pred == True
        return Confusion((y_true & y_pred).sum(), (~y_true & y_pred).sum(),
                                                  (y_true & ~y_pred).sum(), (~y_true &␣
↪~y_pred).sum())

    def summary(self):
        return f"""
Precision: {self.PPV:.3}
Recall: {self.TPR:.3}
F1-score: {self.F1:.3}
Accuracy: {self.accuracy:.3}
"""

    @property
    def P(self):
        return self.TP + self.FN

    @property
    def N(self):
        return self.FP + self.TN

    @property
    def PP(self):
        return self.TP + self.FP

    @property
    def PN(self):
        return self.FN + self.TN

    # Precision
    @property
    @lru_cache()
    def PPV(self):
        return self.TP / self.PP

    # Recall
    @property
    @lru_cache()
```

```
    def TPR(self):
        return self.TP / self.P

    @property
    def F1(self):
        return 2 / (1 / self.TPR + 1 / self.PPV)

    @property
    def accuracy(self):
        return (self.TP + self.TN) / (self.P + self.N)
```

[109]:

```
%%time
model = AdaBoostClassifier(DecisionStumpClassifier() for _ in range(10))
model.fit(X_train, y_train)
confusion = Confusion.calculate(y_test, model.predict(X_test))
print(confusion)
print(confusion.summary())
```

```
<ipython-input-107-cfda11b66637>:27: RuntimeWarning: divide by zero encountered
in double_scalars
  alpha = float(np.log((1 - e) / e))
```

```
Confusion(TP=246, FP=35, FN=208, TN=662)

Precision: 0.875
Recall: 0.542
F1-score: 0.669
Accuracy: 0.789

CPU times: user 1min 31s, sys: 3.79 ms, total: 1min 31s
Wall time: 1min 31s
```

[110]:

```
%%time
model2 = AdaBoostClassifier(DecisionTreeClassifier(1, 0, 2) for _ in range(10))
model2.fit(X_train, y_train)
confusion2 = Confusion.calculate(y_test, model2.predict(X_test))
print(confusion2)
print(confusion2.summary())
```

```
<ipython-input-107-cfda11b66637>:27: RuntimeWarning: divide by zero encountered
in double_scalars
  alpha = float(np.log((1 - e) / e))
```

```
Confusion(TP=327, FP=20, FN=127, TN=677)

Precision: 0.942
```

```
Recall: 0.72
F1-score: 0.816
Accuracy: 0.872

CPU times: user 3min 1s, sys: 39 ms, total: 3min 1s
Wall time: 3min 1s
```

2-depth trees gave a better result, but the difference could be less significant with a greater amount of trees. Increasing the depth also increases learning time significantly.