Solution

March 17, 2023

[1]:
```
import numpy as np
```

0.1 Preprocessing

First, we load the data:

[2]:
```
train_data = np.loadtxt('./train-data.csv', delimiter=',')

test_data = np.column_stack((
    np.loadtxt('./test-data.csv', delimiter=','),
    np.loadtxt('./ground-truth.csv', delimiter=',')
))
```

The data are pretty unbalanced: 5914 from class 0, and 18506 from class 1:

[3]:
```
np.unique(train_data[:, -1], return_counts=True)
```

[3]:
```
(array([0., 1.]), array([ 5914, 18506]))
```

The simplest way of dealing with this is probably to rebalance:

[4]:
```
def balance(data):
    """Balance a data set to have the same number of examples from each class.
    """
    labels = data[:, -1]
    ix_1 = np.where(labels == 1)[0]
    ix_0 = np.where(labels == 0)[0]

    n = min(len(ix_1), len(ix_0))

    return np.concatenate((
        data[ix_1[:n]],
        data[ix_0[:n]]
    ))
```

This helper function will also be useful:

[5]:
```
def split_labels(data):
    """
    Take an array that has labels in the last column and return
    two arrays: one with features, and one with labels.
    """
    return (
        data[:, :-1],
        data[:, -1]
    )
```

Then we balance the training set and create validations sets:

[6]:
```
# balance the training set
balanced_train_data = balance(train_data)
```

[7]:
```
# create a validation set
shuffled_train_data = np.random.permutation(balanced_train_data)
train_set = shuffled_train_data[:-2_000]
validation_set = shuffled_train_data[-2_000:]
```

[8]:
```
# balance the test set
X_test, y_test = split_labels(balance(test_data))
```

[9]:
```
X_full, y_full = split_labels(balanced_train_data)
X_train, y_train = split_labels(train_set)
X_val, y_val = split_labels(validation_set)
```

We'll also create a standardized version of the data set:

[10]:
```
mu = X_train.mean(axis=0)
sigma = X_train.std(axis=0)
```

[11]:
```
def standardize(X):
    return (X - mu) / sigma
```

[12]:
```
Z_train, Z_val, Z_full, Z_test = [standardize(X) for X in [X_train, X_val,␣
↪X_full, X_test]]
```

0.2 Least Squares (Baseline)

The simplest approach to implement is probably linear least squares. It achieves a pretty high score of 68% – enough to get a point of extra credit:

[13]:
```
w = np.linalg.lstsq(X_full, y_full, rcond=None)[0]
```

[14]:
```
pred = (X_test @ w > 0.5).astype(int)
```

[15]:
```
(pred == y_test).mean()
```

[15]: 0.6891541255838091

0.3 kNN

kNN is also pretty simple to implement. It received close to the highest accuracy.

[16]:
```
import scipy.spatial.distance
```

[17]:
```
def knn_predict(X_train, y_train, X, k):
    distances = scipy.spatial.distance_matrix(X, X_train)
    ix_knn = np.argpartition(distances, k)[:,:k]
    return (np.mean(y_train[ix_knn], axis=1) >= 0.5).astype(int)
```

We need to choose the number of neighbors, k. This can be done with a simple grid search:

[18]:
```
for k in range(5, 200, 10):
    val_score = (knn_predict(Z_train, y_train, Z_val, k) == y_val).mean()
    print(f'k = {k}: {val_score}')
```

```
k = 5: 0.6975
k = 15: 0.7085
k = 25: 0.717
k = 35: 0.7155
k = 45: 0.7205
k = 55: 0.7195
k = 65: 0.7225
k = 75: 0.718
k = 85: 0.717
k = 95: 0.72
k = 105: 0.7215
k = 115: 0.7175
k = 125: 0.719
k = 135: 0.7205
k = 145: 0.7205
k = 155: 0.7205
k = 165: 0.7215
k = 175: 0.7175
k = 185: 0.719
k = 195: 0.719
```

[19]:
```
pred = knn_predict(Z_full, y_full, Z_test, 25)
```

[20]:
```
(pred == y_test).mean()
```

[20]: 0.7143227815256876