Module 1 Lab 2 - Machine Learning terms and metrics

Created	@May 31, 2025 7:21 PM
Class	IIITH
Event	Bootcamp
Subject	AI

KNN implementation

Let's see how the `NN` function works.

Training Data and Labels:

library_addcontent_copy

`import numpy as nptraindata = np.array([[1, 2], [3, 4], [5, 6]])trainlabel = np.array([0, 1, 0])` Use code with caution

Test Data:

Now, let's say we have two test data points that we want to classify:

library_addcontent_copy

`testdata = np.array([[2, 3],[4, 5]])` Use code with caution

We will call the `NN` function with this data:

library_addcontent_copy

`predicted_labels = NN(traindata, trainlabel, testdata)` Use code with caution

Now, let's trace what the `NN` function does internally.

The `NN` function has this line:

library_addcontent_copy

`predlabel = np.array([NN1(traindata, trainlabel, i) for i in testdata])` Use code with caution

This is a list comprehension that iterates through each row `i` in the `testdata` array and calls the `NN1` function for each `i`.

Iteration 1: `i` is the first test point `[2, 3]`

- `NN1(traindata, trainlabel, [2, 3])` is called.
- As we saw in the previous example, when the query is `[2, 3]`, `NN1` calculates the distances to the training points `[1, 2]`, `[3, 4]`, and `[5, 6]`.
- The squared distances were `[2, 2, 18]`.

- `np.argmin([2, 2, 18])` returns `0`.
- `trainlabel[0]` is `0`.
- So, `NN1` returns `0` for this first test point.

Iteration 2: `i` is the second test point `[4, 5]`

- `NN1(traindata, trainlabel, [4, 5])` is called.
- Inside `NN1`, the difference (`diff`) between `traindata` and `[4, 5]` is calculated:

library_addcontent_copy

`[[1, 2], [3, 4], [5, 6]] - [4, 5] = [[1-4, 2-5], [3-4, 5-5], [5-4, 6-5]] = [[-3, -3], [-1, 0], [1, 1]]` Use code with caution

- The squared differences (`sq`) are calculated:

library_addcontent_copy

`[[-3 * -3, -3 * -3], [-1 * -1, 0 * 0], [1 * 1, 1 * 1]] = [[9, 9], [1, 0], [1, 1]]` Use code with caution

- The sum of squared differences along axis 1 (`dist`) is calculated:

library_addcontent_copy

`[9 + 9, 1 + 0, 1 + 1] = [18, 1, 2]` Use code with caution

- `np.argmin([18, 1, 2])` is calculated. The minimum value is 1, which is at index 1. `np.argmin()` returns `1`.
- `trainlabel[1]` is calculated. Looking at `trainlabel` (`[0, 1, 0]`), the element at index 1 is `1`.
- So, `NN1` returns `1` for this second test point.

Building `predlabel`:

The list comprehension collects the results from each call to `NN1`: `[0, 1]`. Then, `np.array()` converts this list into a NumPy array.

library_addcontent_copy

`predlabel = np.array([0, 1])` Use code with caution

Output:

When you run `print(predicted_labels)`, the output will be:

library_addcontent_copy

`[0 1]` Use code with caution

This shows that the `NN` function, using the 1-Nearest Neighbor logic implemented in `NN1`, predicted the label `0` for the first test sample `[2, 3]` and the label `1` for the second test sample `[4, 5]`.

Making use of inbuilt sklearn KNeighborsClassifier module.

from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
neigh = KNeighborsClassifier(n_neighbors=3)
neigh.fit(alltraindata, alltrainlabel) #training the data

test_predictions_3nn = neigh.predict(testdata)
accuracy_3nn = accuracy_score(testlabel, test_predictions_3nn)
print(f"Test accuracy using 3-Nearest Neighbors classifier: {accuracy_3nn * 100:.2f}%")

Accuracy Function

This code defines a Python function called `Accuracy` that calculates the accuracy of a classification model's predictions. Accuracy is a common metric used to evaluate how well a model performs.

library_addcontent_copy

def Accuracy(gtlabel, predlabel):
"""
This function takes in the ground-truth labels and predicted labels
and returns the accuracy of the classifier

```
gtlabel: numpy array of shape (n,) where n is the number of samples
predlabel: numpy array of shape (n,) where n is the number of samples

returns: the accuracy of the classifier which is the number of correct predictions divided by the total nu
mber of predictions
"""
assert len(gtlabel) == len(
    predlabel
), "Length of the ground-truth labels and predicted labels should be the same"
correct = (
    gtlabel == predlabel
).sum()  # count the number of times the groundtruth label is equal to the predicted label.
return correct / len(gtlabel)
```

The function takes two arguments:

- `gtlabel`: This is a NumPy array containing the **ground-truth** labels. These are the actual, correct labels for your data.
- `predlabel`: This is also a NumPy array containing the labels that your classification **model predicted**.

Before calculating the accuracy, the code includes an `assert` statement. This is a check to make sure that the `gtlabel` and `predlabel` arrays have the same number of elements. If they don't, it means there's a mismatch between the actual labels and the predicted labels, and the program will stop with an error message.

The core of the accuracy calculation is the line:

`correct = (    gtlabel == predlabel).sum()`

`# count the number of times the groundtruth label is equal to predicted label`

Here, `gtlabel == predlabel` performs an element-wise comparison between the two arrays. It creates a new array of boolean values (`True` where the labels match, `False` where they don't). The `.sum()` method then counts the number of `True` values in this boolean array. This gives you the total number of correct predictions.

Finally, the function returns the accuracy, which is calculated as the number of `correct` predictions divided by the total number of predictions (`len(gtlabel)`). The result is a floating-point number between 0 and 1, where 1 represents 100% accuracy.

in-built Accuracy function

from sklearn.metrics import accuracy_score

accuracy_3nn = accuracy_score(testlabel, test_predictions_3nn)

MULTIPLE SPLITS

```
def AverageAccuracy(alldata, alllabel, splitpercent, iterations, classifier=NN):
    accuracy = 0
        for ii in range(iterations):
            traindata, trainlabel, valdata, vallabel = split(
                alldata, alllabel, splitpercent
            )
            valpred = classifier(traindata, trainlabel, valdata)
            accuracy += Accuracy(vallabel, valpred)
        return accuracy / iterations  # average of all accuracies

avg_acc = AverageAccuracy(alltraindata, alltrainlabel, 75 / 100, 10, classifier=NN)
print("Average validation accuracy:", avg_acc*100, "%")
testpred = NN(alltraindata, alltrainlabel, testdata)

print("Test accuracy:", Accuracy(testlabel, testpred)*100, "%")
```

This line defines the function `AverageAccuracy`. It takes five arguments:

- `alldata`: This is the entire dataset containing the input features.
- `alllabel`: These are the corresponding labels for the entire dataset.
- `splitpercent`: This is a floating-point number representing the percentage of the data to be used for the training set in each split. The remaining data will be used for the validation set.
- `iterations`: This is an integer indicating how many times the data splitting and accuracy calculation process should be repeated.
- `classifier`: This is a function that represents the classification model you want to evaluate. By default, it's set to `NN`, which refers to the Nearest Neighbor classifier defined earlier in the notebook.

The `splitpercent` argument in the `AverageAccuracy` function (and the `split` function it calls) determines the **proportion of the data that will be allocated to the training set** in each individual split.

Think of it like dividing a pie. If `splitpercent` is 50/100 (or 0.5), it means you're trying to cut the pie so that 50% goes to one person (the training set) and the remaining 50% goes to another (the validation set).

Example with `splitpercent = 50 / 100`:

Imagine our small dataset with 6 samples:

library_addcontent_copy

`Sample 1: [1, 2], Label: 0Sample 2: [3, 4], Label: 1Sample 3: [5, 6], Label: 0Sample 4: [7, 8], Label: 1Sample 5: [9, 10], Label: 0Sample 6: [11, 12], Label: 1` Use code with caution

When `split(alldata, alllabel, 50/100)` is called, the function randomly assigns each sample to either the training set or the validation set based on a 50% probability.

In one possible random split with `splitpercent = 50/100`, you might get:

- **Training Set (roughly 50% of data):**
  - Sample 1: `[1, 2]`, Label: `0`
  - Sample 4: `[7, 8]`, Label: `1`
  - Sample 6: `[11, 12]`, Label: `1` (`traindata = [[1, 2], [7, 8], [11, 12]]`, `trainlabel = [0, 1, 1]`)
- **Validation Set (the remaining data, roughly 50%):**
  - Sample 2: `[3, 4]`, Label: `1`
  - Sample 3: `[5, 6]`, Label: `0`
  - Sample 5: `[9, 10]`, Label: `0` (`valdata = [[3, 4], [5, 6], [9, 10]]`, `vallabel = [1, 0, 0]`)

The exact number of samples in each set might not be *exactly* 50/50 due to the random nature and the total number of samples, but the split will aim for that proportion. If you had 10 samples and `splitpercent` was 0.5, you'd expect roughly 5 in training and 5 in validation.

If `splitpercent` was, for example, `75 / 100` (or 0.75), then in each split, about 75% of the data would go to the training set and the remaining 25% would go to the validation set.

`iterations`

The `iterations` argument in the `AverageAccuracy` function determines **how many times the entire process of splitting the data, training the classifier on the training set, predicting on the validation set, and calculating the validation accuracy is repeated.**

Each time the `split` function is called within the loop, it generates a *new* random split of the data according to the `splitpercent`. This means that the specific samples in the training set and the validation set will likely be different in each iteration.

Example with `iterations = 3`:

Continuing from the previous example with our 6 samples and `splitpercent = 50/100`.

- **Iteration 1:**
  - The data is split randomly (e.g., Training = Samples 1, 4, 6; Validation = Samples 2, 3, 5).
  - The classifier is trained on Samples 1, 4, and 6.
  - Predictions are made on Samples 2, 3, and 5.
  - The accuracy for this specific split (let's say it's 0.333) is calculated and added to the total.
- **Iteration 2:**
  - The data is split randomly *again*. This time, the split might be different (e.g., Training = Samples 2, 3, 5; Validation = Samples 1, 4, 6).
  - The classifier is trained on Samples 2, 3, and 5.

    - Predictions are made on Samples 1, 4, and 6.
    - The accuracy for *this new* split (let's say it's 0.667) is calculated and added to the total.
- **Iteration 3:**
    - The data is split randomly *a third time* (e.g., Training = Samples 1, 3, 5; Validation = Samples 2, 4, 6).
    - The classifier is trained on Samples 1, 3, and 5.
    - Predictions are made on Samples 2, 4, and 6.
    - The accuracy for this third split (let's say it's 0.0) is calculated and added to the total.

After 3 iterations, the total accuracy accumulated would be 0.333 + 0.667 + 0.0 = 1.0.

Finally, the function divides this total by the number of iterations (`1.0 / 3 = 0.333`) to get the average accuracy across these 3 random splits.

# `split` function uses a **random process** to decide which samples go into the training set and which go into the validation set.

the `split` function:

rnd = rng.random(len(label))
split1 = rnd < percent
split2 = rnd >= percent

- `rng.random(len(label))` generates an array of random numbers between 0 and 1, one for each sample in your dataset.
- `split1 = rnd < percent` creates a boolean array where `True` indicates that the random number for that sample is less than `percent`, and `False` otherwise. Samples with `True` in `split1` are assigned to the first split (which we use as the training set).
- `split2 = rnd >= percent` creates a boolean array where `True` indicates the random number is greater than or equal to `percent`. Samples with `True` in `split2` are assigned to the second split (which we use as the validation set).

Since the random numbers generated by `rng.random()` are (by default) different each time the `split` function is called, the outcome of the comparison `rnd < percent` will also be different. This means that the indices of the samples that get assigned to `split1` (the training set) and `split2` (the validation set) will change with each call to `split`.

- `rng.random(6)` will generate a NumPy array containing 6 random floating-point numbers, each between 0.0 and 1.0 (exclusive of 1.0). These numbers are generated using the random number generator `rng` which was initialized with a seed (42 in the notebook), making the sequence of random numbers reproducible if you run the notebook again with the same seed.

EXERCISE

How does the accuracy of the 3 nearest neighbour classifier change with the number of splits? How is it affected by the split size? Compare the results with the 1 nearest neighbour classifier.

General Expectation: For many datasets, a k-NN classifier with k > 1 (like 3-NN) often outperforms 1-NN. 1-NN can overfit to the training data and be very sensitive to local noise. 3-NN smooths out the decision boundary by considering more neighbors, which can lead to better generalization on unseen data.

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
import numpy as np


def AverageAccuracySKLearn(alldata, alllabel, splitpercent, iterations, n_neighbors):
    """
    Calculates average accuracy for KNeighborsClassifier over multiple splits.
    """
    accuracy = 0
    for ii in range(iterations):
        # Use the existing split function
        traindata, trainlabel, valdata, vallabel = split(
            alldata, alllabel, splitpercent
        )
        # Create and train the KNeighborsClassifier
        knn_classifier = KNeighborsClassifier(n_neighbors=n_neighbors)
        knn_classifier.fit(traindata, trainlabel)

        # Make predictions on the validation data
        valpred = knn_classifier.predict(valdata)

        # Calculate accuracy using sklearn's accuracy_score
        accuracy += accuracy_score(vallabel, valpred)

    return accuracy / iterations

# --- Experimentation ---

# Parameters to test
split_percentages_to_test = [10/100, 25/100, 50/100, 75/100, 90/100, 99/100] # Vary split size
iterations_to_test = [10, 50] # Vary number of splits

print("Comparing 1-NN and 3-NN Average Validation Accuracy:")

for percent in split_percentages_to_test:
    print(f"\nSplit Percentage (Train Data): {percent*100:.2f}%")
    for num_iterations in iterations_to_test:
        print(f"  Number of Iterations: {num_iterations}")

        # Calculate average accuracy for 1-NN (using the original NN function or SKLearn with n_neighbors
=1)
        # Using the original NN function if it's fast enough, otherwise SKLearn n_neighbors=1
        # avg_acc_1nn = AverageAccuracy(alldata, alllabel, percent, num_iterations, classifier=NN)
        avg_acc_1nn = AverageAccuracySKLearn(alltraindata, alltrainlabel, percent, num_iterations, n_neigh
bors=1)
```

```
        print(f"    1-NN Average Validation Accuracy: {avg_acc_1nn*100:.2f}%")

        # Calculate average accuracy for 3-NN using the new function
        avg_acc_3nn = AverageAccuracySKLearn(alltraindata, alltrainlabel, percent, num_iterations, n_neig
hbors=3)
        print(f"    3-NN Average Validation Accuracy: {avg_acc_3nn*100:.2f}%")

# Remember to use the 'alltraindata' and 'alltrainlabel' for these validation experiments,
# as the 'testdata' and 'testlabel' should be held out completely until final evaluation.
# So, replace alldata, alllabel with alltraindata, alltrainlabel in the AverageAccuracy calls.
```