06.4 - Supervised Learning - Classification - K-Nearest Neighbors and Decision Trees

October 26, 2020

1 Programming for Data Science and Artificial Intelligence

1.1 6.4 Supervised Learning - Classification - K-Nearest Neighbors and Decision Trees

1.1.1 Readings:

- [VANDER] Ch5
- [HASTIE] Ch9, 13

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
```

1.2 K-Nearest Neighbors

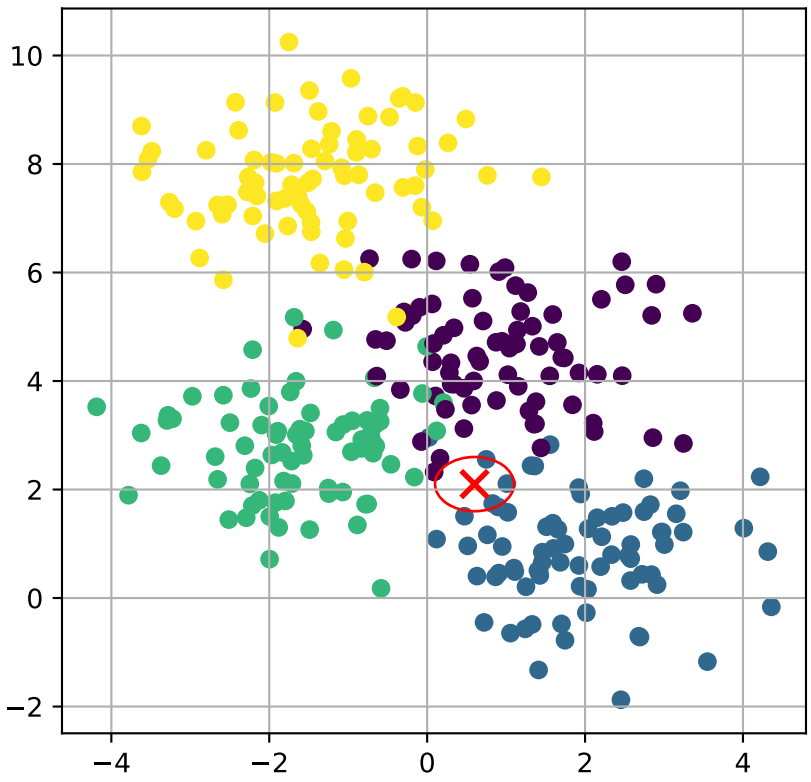
The intuition behind the KNN algorithm is one of the simplest of all the supervised machine learning algorithms. It simply calculates the distance of a new data point to all other training data points. The distance can be of any type e.g Euclidean or Manhattan etc. It then selects the K-nearest data points, where K can be any integer. Finally it assigns the data point to the class to which the majority of the K data points belong.

For example, given the red cross X, it simply get the majority class of neighbors, and assign to its own.

[2]:
```
#let's consider the following 2D data with 4 classes
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

X, y = make_blobs(n_samples=300, centers=4,
                  random_state=0, cluster_std=1.0)

xfit = np.linspace(-1, 3.5)

figure = plt.figure(figsize=(5, 5))
ax = plt.axes()  #get the instance of axes from plt

ax.grid()
```

```
ax.scatter(X[:, 0], X[:, 1], c=y)

#where should this value be classified as?
ax.plot([0.6], [2.1], 'x', color='red', markeredgewidth=2, markersize=10)

#let's say roughly 5 neighbors
circle = plt.Circle((0.6, 2.1), 0.5, color='red', fill=False)
ax.add_artist(circle)
```

[2]: <matplotlib.patches.Circle at 0x21c425a0d48>



1.2.1 Scratch

Implementation steps:

1. Prepare your data
   - X and y in the right shape
     - X -> (m, n)
     - y -> $(m,)$
     - Why no w?
   - train-test split
   - feature scale

- clean out any missing data
- (optional) feature engineering

2. Write a function for computing pairwise distance between every points
3. Then, given set of X_test data, compute their distance to all other points, then argsort the distance matrix, and get the k-nearest indices
4. Get the majority class

1. Prepare your data

[3]:
```
#standardize
scaler = StandardScaler()
X = scaler.fit_transform(X)

#do train test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
```

2. Function for pairwise distance I have written three different ways in numpy assignment answer question 11

[4]:
```
def find_distance(X_train, X_test): #<--look Numpy Assignment Answer Question 11
    #create newaxis simply so that broadcast to all values
    dist = X_test[:, np.newaxis, :] - X_train[np.newaxis, :, :]
    sq_dist = dist ** 2

    #sum across feature dimension, thus axis = 2
    summed_dist = sq_dist.sum(axis=2)
    sq_dist = np.sqrt(summed_dist)
    return sq_dist
```

3. Argsort the pairwise distance matrix

[5]:
```
def find_neighbors(X_train, X_test, k=3):
    dist = find_distance(X_train, X_test)
    #return the first k neighbors
    neighbors_ix = np.argsort(dist)[:, 0:k]
    return neighbors_ix
```

4. Get the majority class

[6]:
```
#https://numpy.org/doc/stable/reference/generated/numpy.bincount.html
def get_most_common(y):
    return np.bincount(y).argmax()
```

Let's write a wrapper function that links all function and use it

[7]:
```
from sklearn.metrics import average_precision_score, classification_report
from sklearn.preprocessing import label_binarize
```

```
def predict(X_train, X_test, y_train, k=3):
    neighbors_ix = find_neighbors(X_train, X_test, k)
    pred = np.zeros(X_test.shape[0])
    for ix, y in enumerate(y_train[neighbors_ix]):
        pred[ix] = get_most_common(y)
    return pred

yhat = predict(X_train, X_test, y_train, k=3)

n_classes = len(np.unique(y_test))

print("Accuracy: ", np.sum(yhat == y_test)/len(y_test))

print("=========Average precision score=======")
y_test_binarized = label_binarize(y_test, classes=[0, 1, 2, 3])
yhat_binarized = label_binarize(yhat, classes=[0, 1, 2, 3])

for i in range(n_classes):
    class_score = average_precision_score(y_test_binarized[:, i],␣
 ↪yhat_binarized[:, i])
    print(f"Class {i} score: ", class_score)

print("=========Classification report=======")
print("Report: ", classification_report(y_test, yhat))
```

```
Accuracy:  0.9222222222222223
=========Average precision score=======
Class 0 score:  0.7409090909090909
Class 1 score:  0.929513888888889
Class 2 score:  0.8866666666666666
Class 3 score:  0.926984126984127
=========Classification report=======
Report:                precision    recall  f1-score   support

           0       0.77      0.94      0.85        18
           1       0.96      0.96      0.96        24
           2       0.96      0.89      0.92        27
           3       1.00      0.90      0.95        21

    accuracy                           0.92        90
   macro avg       0.92      0.92      0.92        90
weighted avg       0.93      0.92      0.92        90
```

1.2.2 Sklearn

[8]:
```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import StratifiedShuffleSplit, GridSearchCV

model = KNeighborsClassifier()
param_grid = {"n_neighbors": np.arange(2, 10)}

cv = StratifiedShuffleSplit(n_splits=5, test_size=0.2, random_state=42)
grid = GridSearchCV(model, param_grid=param_grid, cv=cv)
grid.fit(X_train, y_train)

print(f"The best parameters are {grid.best_params_} with" +
          f"a score of {grid.best_score_:.2f}")

model = grid.best_estimator_

model.fit(X_train, y_train)
yhat = model.predict(X_test)

print("Accuracy: ", np.sum(yhat == y_test)/len(y_test))

print("=========Average precision score=======")
y_test_binarized = label_binarize(y_test, classes=[0, 1, 2, 3])
yhat_binarized = label_binarize(yhat, classes=[0, 1, 2, 3])

for i in range(n_classes):
    class_score = average_precision_score(y_test_binarized[:, i],␣
 ↪yhat_binarized[:, i])
    print(f"Class {i} score: ", class_score)

print("=========Classification report=======")
print("Report: ", classification_report(y_test, yhat))
```

```
The best parameters are {'n_neighbors': 3} witha score of 0.95
Accuracy:  0.9222222222222223
=========Average precision score=======
Class 0 score:  0.7409090909090909
Class 1 score:  0.929513888888889
Class 2 score:  0.8866666666666666
Class 3 score:  0.926984126984127
=========Classification report=======
Report:                precision    recall  f1-score   support

           0       0.77      0.94      0.85        18
           1       0.96      0.96      0.96        24
           2       0.96      0.89      0.92        27
           3       1.00      0.90      0.95        21
```

```
    accuracy                           0.92        90
   macro avg       0.92      0.92      0.92        90
weighted avg       0.93      0.92      0.92        90
```

1.2.3 ===Task===

Your work: Let's modify the above scratch code to - If the majority class of the first place is equal to the second place, then ask the algorithm to pick the next nearest neighbors as the decider - Modify the code so it outputs the probability of the decision, where the probability is simply the class probability based on all the nearest neighbors - Write a function which allows the program to receive a range of k, and output the cross validation score. Last, it shall inform us which k is the best to use from a predefined range - Put everything into a class KNN(k=3). It should have at least one method, predict(X_train, X_test, y_train)

1.2.4 When to use KNN

I guess the only good thing about it is that KNN is super easy to implement, and generally work quite well on simple classification problems. However, it also comes with a price:

- Computational expense as feature grows, since it requires computing the distance for each feature, where for each feature, we have to compute the input points with every single points, then perform sort (which can be expensive), and then get the majority class from the nearest nth-neighbors. Very expensive!
- Can't work with categorical features since it is difficult to formulate distance formulas for categorial features
- Of course, it takes even more time to find the right n_neighbors (or commonly known as k)

1.3 Decision Trees

Decision trees are extremely intuitive ways to classify or label objects: you simply ask a series of questions designed to zero-in on the classification.

How is a Decision Tree fit?

- Which variables to include on the tree?
- How to choose the threshold?
- When to stop the tree?

Key idea is that we want to choose the feature that has the lowest "impurity" to split our tree, thus our tree can reach the decision as fast as possible with smallest height possible

One way to measure impurity is using **Gini index** (another one is entropy but which measure very similar thing) with the following formula:

$$I_G = 1 - \sum_{i=1}^{c} p_i^2$$

where c is number of classes, and p_i is the probability of each class. For example, let's say our X is [[2],[3],[10],[19]] and y is [0, 0, 1, 1]. That is, if a node has 4 samples, and 2 samples are of class cancer, and 2 samples are of no cancer, then the probability of each class is

$$p_{cancer} = (2/4)^2 = 0.25$$

and

$$p_{no-cancer} = (2/4)^2 = 0.25$$

Thus the gini index of this node is

$$I_G = 1 - (0.25 + 0.25) = 0.5$$

Then we need to decide how to best split this node so we can get the lowest gini (highest purity) children.

For example, if we split this sample with **x1 < 3**: we will get left node X as [[2]] and y as [0] and the right node X as [[3],[10],[19]] and y as [0, 1, 1]. The weighted gini of the children are

$$1/4 * I_{leftG} + 3/4 * I_{rightG} =$$
$$1/4 * (1 - (1/1)^2) + 3/4 * (1 - (1/3)^2 - (2/3)^2) = 0.33$$

Hmm...but we know we can split better, right? Let's try **x1 < 4**: we will get left node X as [[2],[3]] and y as [0, 0] and the right node X as [[10],[19]] and y as [1, 1]. If you do the math right, the gini is 0!

$$2/4 * (1 - (2/2)^2) + 2/4 * (1 - (2/2)^2) = 0$$

Thus, in conclusion, we can say that spliting **x1<4** is a much better split than **x1<3**. However, to really find the best split, it is an exhaustive and greedy algorithm, in which we have to iterate and check every value on each feature as a candidate split, find the gini index.

How do we find all threshold for continuous values? We can sort all features. Then we are identify critical value using the midpoint between all consecutive values. For example, given X is [[2],[3],[10],[19]], the critical value to compare is 2.5, 6.5 and 14.5.

The code can be implemented in several ways. Example are shown below:

[9]:
```
#Credit: https://github.com/joachimvalente/decision-tree-cart/blob/master/cart.
 ↪py
#Edited to make it more readable and precise for students
"""
Idea is simple.  Simply loop through all possible threshold:
2.5, 6.5, 14.5.

2.5 threshold will give
[0]         [0, 1, 1]
```

```
6.5 threshold will give
[0, 0]      [1, 1]
14.5 threshold will give
[0, 0, 1]   [1]

Then we simply calculate the best gini.

This approach work best if we first sort
our sample to be in order, since we will have fast way
to tell what are the feature value used to split that particular
way.

"""
def find_split(X, y, n_classes):
    """ Find split where children has lowest impurity possible
    in condition where the purity should also be less than the parent,
    if not, stop.
    """
    n_samples, n_features = X.shape
    if n_samples <= 1:
        return None, None

    #so it will not have any warning about "referenced before assignments"
    feature_ix, threshold = None, None

    # Count of each class in the current node.
    sample_per_class_parent = [np.sum(y == c) for c in range(n_classes)]

    # Gini of parent node.
    best_gini = 1.0 - sum((n / n_samples) ** 2 for n in sample_per_class_parent)

    # Loop through all features.
    for feature in range(n_features):

        # Sort data along selected feature.
        sample_sorted = sorted(X[:, feature]) #[2, 3, 10, 19]
        sort_idx = np.argsort(X[:, feature])
        y_sorted = y[sort_idx] #[0, 0, 1, 1]

        sample_per_class_left = [0] * n_classes   #[0, 0]

        sample_per_class_right = sample_per_class_parent.copy() #[2, 2]

        # loop through each threshold, 2.5, 6.5, 14.5
        for i in range(1, n_samples):
            #the class of that sample
            c = y_sorted[i - 1]  #[0]
```

```
            #put the sample to the left
            sample_per_class_left[c] += 1  #[1, 0]

            #take the sample out from the right  [1, 2]
            sample_per_class_right[c] -= 1

            gini_left = 1.0 - sum(
                (sample_per_class_left[x] / i) ** 2 for x in range(n_classes)
            )

            #we divided by n_samples - i since we know that the left amount of␣
↪samples
            #since left side has already i samples
            gini_right = 1.0 - sum(
                (sample_per_class_right[x] / (n_samples - i)) ** 2 for x in␣
↪range(n_classes)
            )

            #weighted gini
            weighted_gini = ((i / n_samples) * gini_left) + ( (n_samples - i) /
↪n_samples) * gini_right

            # in case the value are the same, we do not split
            # (both have to end up on the same side of a split).
            if sample_sorted[i] == sample_sorted[i - 1]:
                continue

            if weighted_gini < best_gini:
                best_gini = weighted_gini
                feature_ix = feature
                threshold = (sample_sorted[i] + sample_sorted[i - 1]) / 2  #␣
↪midpoint

    #return the feature number and threshold
    #used to find best split
    return feature_ix, threshold

X = np.array([[2],[3],[10],[19]])
y = np.array([0, 0, 1, 1])
feature, threshold = find_split(X, y, len(set(y)))

#will print 0, 6.5
print("Best feature used for split: ", feature)
print("Best threshold used for split: ", threshold)
```

```
Best feature used for split:  0
```

```
Best threshold used for split:  6.5
```

1.3.1 Scratch

One all value are exhausted, we can then use the best decision note as our split node. Then when we go to the next node, we have to repeat again. This algorithm is called **CART (Classification and Regression Trees)** algorithm, where the recursion keeps on going until certain stop criteria, such as maximum tree depth is reached, or no split can produce two children with lower purity.

Implementation steps:

1. Calculate the purity of the data
2. Select a candidate split
3. Calculate the purity of the data after the split
4. Repeat for all variables
5. Choose the variable with the lowest impurity
6. Repeat for each split until some stop criteria is met

Example could be stop criteria could be max tree depth, or minimum node records.

Here are some snippets of the possible implementation of Decision Tree

[10]:
```
#To help with our implementation, we create a class Node
class Node:
    def __init__(self, gini, num_samples, num_samples_per_class,␣
 ↪predicted_class):
        self.gini = gini
        self.num_samples = num_samples
        self.num_samples_per_class = num_samples_per_class
        self.predicted_class = predicted_class
        self.feature_index = 0
        self.threshold = 0
        self.left = None
        self.right = None
```

Let's try implement Decision Tree to see how it looks

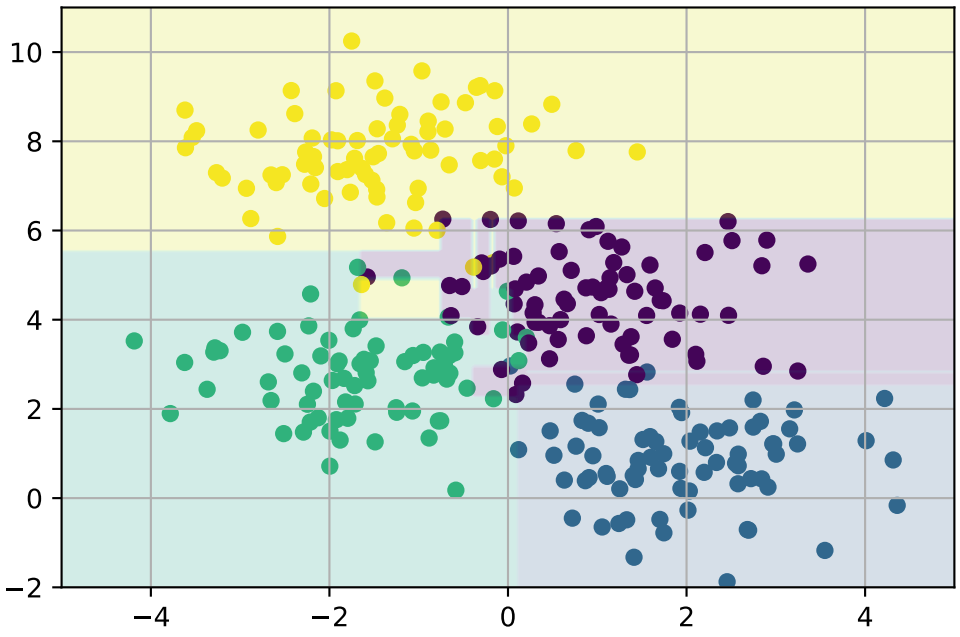
[11]:
```
def fit(Xtrain, ytrain, n_classes, depth=0):
    n_samples, n_features = Xtrain.shape
    num_samples_per_class = [np.sum(ytrain == i) for i in range(n_classes)]
    #predicted class using the majority of sample class
    predicted_class = np.argmax(num_samples_per_class)

    #define the parent node
    node = Node(
        gini = 1 - sum((np.sum(y == c) / n_samples) ** 2 for c in␣
 ↪range(n_classes)),
        predicted_class=predicted_class,
        num_samples = ytrain.size,
        num_samples_per_class = num_samples_per_class,
```

```
        )

    #perform recursion
    feature, threshold = find_split(Xtrain, ytrain, n_classes)
    if feature is not None:
        #take all the indices that is less than threshold
        indices_left = X[:, feature] < threshold
        X_left, y_left = X[indices_left], y[indices_left]

        #tilde for negation
        X_right, y_right = X[~indices_left], y[~indices_left]

        #take note for later decision
        node.feature_index = feature
        node.threshold = threshold
        node.left = fit(X_left, y_left, n_classes, depth + 1)
        node.right = fit(X_right, y_right, n_classes, depth + 1)
    return node

#to predict, it is as simple as moving
#through the tree
def predict(sample, tree):
    while tree.left:
        if sample[tree.feature_index] < tree.threshold:
            tree = tree.left
        else:
            tree = tree.right
    return tree.predicted_class

#fit starting with tree depth = 0
Xtrain = np.array([[2, 5],[3, 5],[10, 5],[19, 5]])
ytrain = np.array([0, 0, 1, 1])
Xtest = np.array(([[4, 6],[6, 9],[9, 2],[12, 8]]))
ytest = np.array([0, 0, 1, 1])

tree = fit(Xtrain, ytrain, len(set(ytrain)))
pred = [predict(x, tree) for x in Xtest]

print("Tree feature ind: ", tree.feature_index)
print("Tree threshold: ", tree.threshold)
print("Pred: ", np.array(pred))
print("ytest: ", ytest)
```

```
Tree feature ind:  0
Tree threshold:  6.5
Pred:  [0 0 1 1]
ytest:  [0 0 1 1]
```

1.3.2 Sklearn

[12]:
```
from sklearn.tree import DecisionTreeClassifier

X, y = make_blobs(n_samples=300, centers=4,
                  random_state=0, cluster_std=1.0)

model = DecisionTreeClassifier().fit(X, y)

def plot_tree(model, X, y):
    plt.grid()
    plt.scatter(X[:, 0], X[:, 1], c=y, s=30)

    xx, yy = np.meshgrid(np.linspace(-5, 5, num=200),
                         np.linspace(-2, 11, num=200))
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)

    # Create a color plot with the results
    n_classes = len(set(y))
    contours = plt.contourf(xx, yy, Z, alpha=0.2)


plot_tree(model, X, y)
```



1.3.3 ===Task===

Let's modify the above scratch code to - Modify the scratch code so it can accept an hyperparameter max_depth, in which it will continue create the tree until max_depth is reached. - Put everything into a class DecisionTree. It should have at least two methods, fit(), and predict() - Load the iris data and try with your class

1.3.4 When to use Decision Trees

Decision Trees are more powerful than other classification in a sense that it can work very well given heterogenous features. However, the downsides is high possibility of over-fitting: it is very easy to go too deep in the tree, and thus to fit details of the particular data rather than the overall properties of the distributions they are drawn from.

However, by using information from multiple decision trees training on subset of data (i.e., random forests), we might expect better results. We shall explore random forests and a general family of ensembles later in our course.

[ ]: