CME 466: Design of a Digital System

Winter 2024

Machine Learning – Classification [part 1]

KHAN A. WAHID, PhD, PEng, SMIEEE

Professor
Electrical and Computer Engineering
University of Saskatchewan
Email: khan.wahid@usask.ca

1. Classification in Machine Learning (ML)

- Classification is a supervised machine learning technique where the model predicts the correct class (or label) of a given multi-class data. The model is fully trained using a set of labeled data, and then it is applied on a test and unseen data to perform prediction.
- Classification is the most widely used ML method. Image or pattern recognition and object detection are two common applications. During the COVID-19 pandemic, researchers used ML classification model to predict whether a person had COVID-19 or not.

1.1 Types of classification

- There are four main classification tasks in Machine learning[1]:
- **Binary classification:** The goal is to classify the data into two categories that are mutually exclusive: true and false; positive and negative. For instance, we can predict whether a given email is spam or not.
- **Multi-class classification:** If there are more than two mutually exclusive class labels, it is multi-class classification. For example, to identify type of vehicles (i.e., car, truck, motorbike, etc.) from an image of a highway.
- **Multi-label classification:** Here the dataset has multiple classes (or labels), but the model predicts 0 or more classes that are not mutually exclusive (data sample can have more than one label). An example would be to auto-tag a given text into multiple topics (English or French) in Natural Language Processing.
- **Imbalanced classification:** In this case, the number of samples is unevenly distributed in each class, meaning that we can have more of one class than the others in the training data. For example, predicting diagnosis of a new disease, detection of fraudulent activity over the internet, etc.

[1] https://www.datacamp.com/blog/classification-machine-learning

1.2 Classification algorithms and evaluation metrics

- There are many classification algorithms used in modern machine leaning. Some examples are: K-Nearest Neighbor (kNN), Logistic Regression, Support Vector Machine (SVM), Random Forest, Gradient Boosting, Naive Bayes, Decision Trees, and Artificial Neural Networks.

- Some common metrics to evaluate the models are: accuracy, precision, recall, F1 score, confusion matrix, AUC (Area Under the Curve), and ROC (Receiver Operating Characteristic).

1.3 Classification example using K-Nearest Neighbor (Iris dataset)

- In this example, we will use the Iris plants dataset[2]. This is perhaps the best known database to be found in the pattern recognition literature.

- There are 150 samples of iris plants that are equally divided into three classes: Iris-Setosa, Iris-Versicolour, and Iris-Virginica. The attributes (or features) are sepal length, sepal width, petal length, and petal width (all in cm).

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns

# import load_iris function from datasets module
from sklearn.datasets import load_iris
```

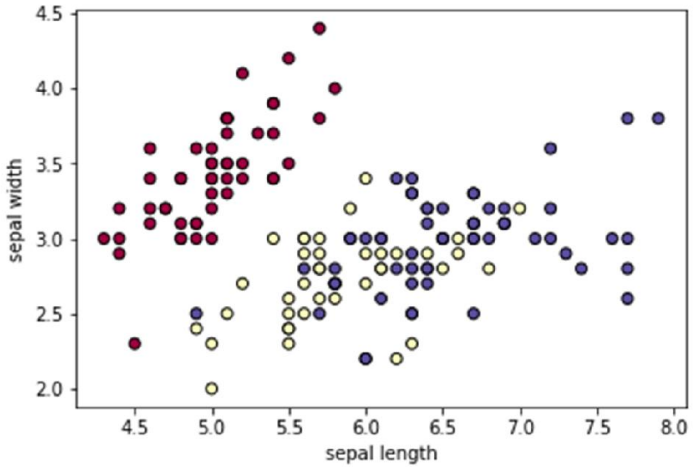
[2] https://scikit-learn.org/stable/datasets/toy_dataset.html

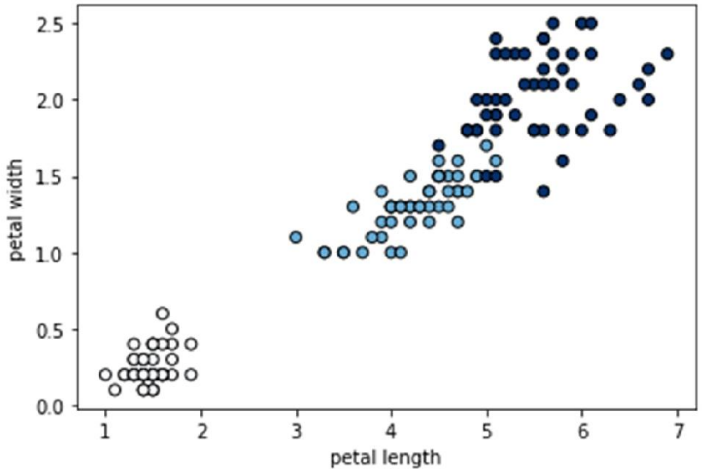
```
irisDS = load_iris()
print(irisDS.data.shape)
type(irisDS)

# Create a dataframe
df = pd.DataFrame(irisDS.data, columns = irisDS.feature_names)
df['target'] = irisDS.target
df

# see the target or classes
print(irisDS.target_names)
```

- Like previous examples of regression, it is important to observe the characteristics of the dataset to better understand what we are dealing with and which method can be chosen for best results. Let us plot the features in pairs (sepal length vs sepal width and petal length vs petal width).

```
# use four features
X = irisDS.data[:, :4]
y = irisDS.target
print(X.shape, y.shape)

# plot training data for sepal
#plt.subplots(figsize=(10,8))
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Spectral, edgecolor="k")
plt.xlabel("sepal length")
plt.ylabel("sepal width")
```

```
# plot training data for petal
#plt.subplots(figsize=(10,8))
plt.scatter(X[:, 2], X[:, 3], c=y, cmap=plt.cm.Blues, edgecolor="k")
plt.xlabel("petal length")
plt.ylabel("petal width")
```





- It can be seen from the plots that one class is linearly (and easily) separable from the other two classes; however, the other two are not linearly separable from each other, since there are some overlapping samples.

- Also note that, in the *plt.scatter ()* function, making *c = y* enables the scatter() to use the class information in the target array *y* to appropriately colorize the points in different classes, for better visual classification.

- We can also use *pairplot* from seaborn to observe the same characteristics.

```
sns.pairplot(df, hue='target')
```

1.4 Nearest Neighbors Classification[3]: K-Nearest Neighbor (kNN) and Radius Neighbors Classifier (rNN)

- Neighbors-based classification is a type of instance-based learning or non-generalizing learning. It does not construct a model, instead simply stores instances of the training data.

- Classification is computed from a simple majority vote of the nearest neighbors of each point: a query point is assigned the data class which has the most representatives within the nearest neighbors of the point.

- scikit-learn implements two different nearest neighbors classifiers:

    - **KNeighborsClassifier (kNN)** implements learning based on *k* number of nearest neighbors of each query point, where *k* is an integer value specified by the user.
    - **RadiusNeighborsClassifier (rNN)** implements learning based on the number of neighbors within a fixed radius *r* of each training point, where *r* is a floating-point value specified by the user.

- The kNN classifier is the most commonly used technique[4]. The optimal choice of the value *k* is highly data-dependent. In general, a larger *k* suppresses the effects of noise, but makes the classification boundaries less distinct.

```
# import classidier from scikit
from sklearn import metrics
from sklearn.neighbors import KNeighborsClassifier
knn_nn3 = KNeighborsClassifier(n_neighbors=3)
print(knn_nn3)
```

[3] https://scikit-learn.org/stable/modules/neighbors.html#nearest-neighbors-classification
[4] https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

- The number of *n_neighbors* is chosen as 3 (the default is 5). By default, it uses 'uniform' *weights*: that is, the value assigned to a query point is computed from a simple majority vote of the nearest neighbors.

- Sometimes, it is better to weigh the neighbors such that nearer neighbors contribute more to the fit. In that case, change *weights = 'distance'* where weights proportional to the inverse of the distance from the query point is used.

- Alternatively, a user-defined function of the distance can be supplied to compute the weights.

```
knn_nn3.fit(X,y)
# 1 seen, 1 unseen
x1 = np.array([6.7, 3.0, 5.2, 2.3])
x2 = np.array([7,3,4,1])
knn_nn3.predict([x1, x2])
# try on two unseen data
x1 = np.array([5,3.7,1.4,0])
x2 = np.array([7,3,4,3])
knn_nn3.predict([x1, x2])

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4,
random_state=0)
print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)

#plot training data for sepal
plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test)
plt.xlabel("sepal length")
plt.ylabel("sepal width")
```

```
#plot training data for petal
plt.scatter(X_test[:, 2], X_test[:, 3], c=y_test)
plt.xlabel("petal length")
plt.ylabel("petal width")

knn_nn3.fit(X_train,y_train)
y_pred = knn_nn3.predict(X_test)
print("accuracy: \n", metrics.accuracy_score(y_test, y_pred))
```

```
accuracy:
     0.9333333333333333
```

- The accuracy of kNN (using *k* = 3) is 93.33%. Can you change k = 5 and see if the improves?

- Now, let us use **RadiusNeighborsClassifier with radius = 3.0** (default is 1.0). Now, the classifier will implement a vote among neighbors within the given radius of 3.0 unit.

```
from sklearn.neighbors import RadiusNeighborsClassifier
rnn = RadiusNeighborsClassifier(radius=3.0)
print(rnn)
rnn.fit(X,y)
rnn.predict([x1, x2])
```

- Immediately, we see there is a miss-classification. This is because of the use of a shorter radius. We can try on the split dataset and see the overall performance.

```
rnn.fit(X_train,y_train)
y_pred = rnn.predict(X_test)
```

```
print("accuracy: \n", metrics.accuracy_score(y_test, y_pred))
      accuracy:
             0.75
```

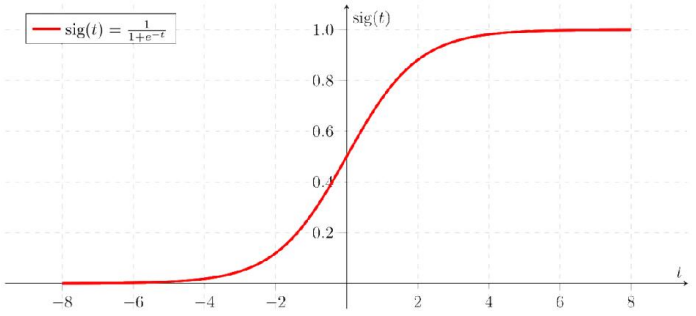
- The accuracy of rNN is much worse than that of kNN. It should be noted that, in cases where the data is not uniformly sampled, rNN can be a better choice than kNN.

1.5 Classification using Logistic Regression

- **Despite the name, Logistic Regression (LR) is not a regressor, but rather a linear model for classification**. It is also known as logit regression, maximum-entropy classification, or the log-linear classifier.

- Unlike kNN (which is instance-based), LR attempts to create a model to predict the class. It can be understood by using a Sigmoid function:

$$\text{sig}(t) = \frac{1}{1 + e^{-t}}$$



- The coefficients in this model is calculated via maximum likelihood estimation (MLE). This method tests different values through multiple iterations to optimize for the best fit curve[5].

- For binary classification, a probability less than 0.5 will predict as “0”, while a probability greater than 0.5 will predict as “1” [6].

```
from sklearn.linear_model import LogisticRegression
```

[5] https://www.ibm.com/topics/logistic-regression
[6] https://web.stanford.edu/~jurafsky/slp3/5.pdf

```
log_reg = LogisticRegression(max_iter=200)
log_reg.fit(X,y)
log_reg.fit(X_train,y_train)
y_pred = log_reg.predict(X_test)
print("accuracy: \n", metrics.accuracy_score(y_test, y_pred))
```

```
accuracy:
     0.9166666666666666
```

- Given the above results, we can say that kNN with *n_neighbors* = 3 has a better generalization capability in learning.

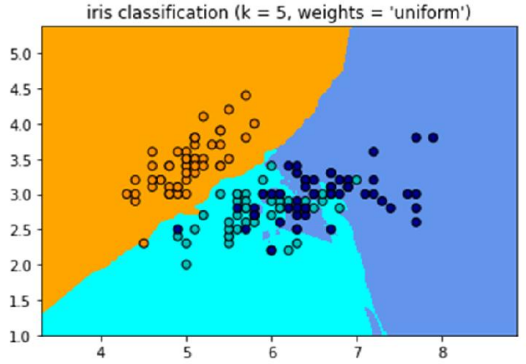
- **Questions:**
  - How do we know what parameter values will be best for a classification task? e.g., what should *n_neighbors* be?
  - How much data should we use for training?
  - Does it matter if the classification scheme is instance-based vs. model-based?
  - What is the implications of over-fitting vs under-fitting (a.k.a, bias-variance tradeoff)?

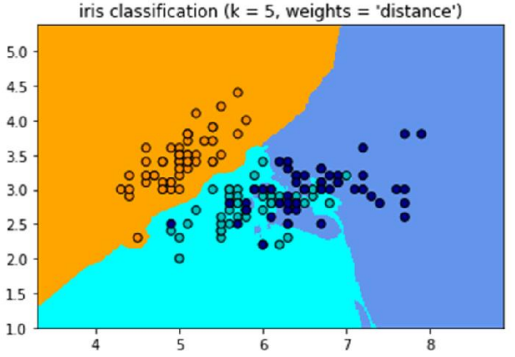
1.6 Decision boundary plot

- In a classification task, it would be nicer to see the decision boundary of the classifier. There are several ways to plot the decision line. We can use the instructions given here:
  https://www.tvhahn.com/posts/beautiful-plots-decision-boundary/

- Alternately, the following code may also be used:

```
from matplotlib.colors import ListedColormap
# Create color maps
cmap_light = ListedColormap(["orange", "cyan", "cornflowerblue"])
cmap_bold = ListedColormap(["darkorange", "c", "darkblue"])

X = irisDS.data[:, :2]
y = irisDS.target
h = .02
n_neighbors = 5

for weights in ['uniform', 'distance']:
    # we create an instance of Neighbours Classifier and fit the data.
    clf = KNeighborsClassifier(n_neighbors, weights=weights)
    clf.fit(X, y)

    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    # plt.scatter(xx, yy, c='r', s=50, cmap='autumn',alpha=0.4)
    # plt.axis('equal')
     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.figure()
    plt.pcolormesh(xx, yy, Z, cmap=cmap_light)

    # Plot also the training points
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap_bold,
edgecolor="k")
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.title("iris classification (k = %i, weights = '%s')" % (n_neighbors, weights))
#plt.show()
```





1.7 Model evaluation metrics[7]

- So far we only used model accuracy in determining the performance of the classification algorithm. Accuracy is the number of correctly predicted data points out of all the data points. More formally, it is defined as:

$$Accuracy = \frac{no.of\ correct\ prediciitons}{total\ no.of\ predictions} = \frac{TP + TN}{TP + TN + FP + FN}$$

Where, *TP* = True Positives, *TN* = True Negatives, *FP* = False Positives, and *FN* = False Negatives.

- An accuracy of 0.9166 or 91.66% means 91.66 correct predictions out of 100 total test samples. Does it mean the classifier is doing a great job of classifying the samples?

- Along with accuracy, *precision* (aka, positive predicted value) and *recall* (aka, sensitivity or true positive value) are two metrics that are used together to evaluate the performance of the classification algorithm.

$$Precision = \frac{TP}{TP + FP} \qquad Recall = \frac{TP}{TP + FN}$$

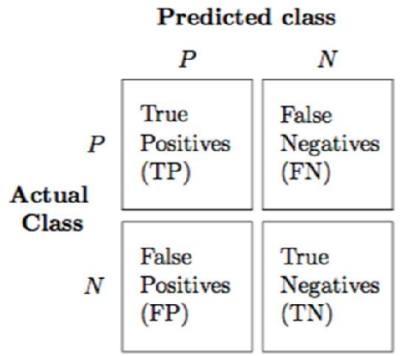
- Precision-Recall metrics are useful when the classes are very imbalanced. In information retrieval, precision is a measure of result relevancy, while recall is a measure of how many truly relevant results are returned[8].

- More information can be found here:

https://amueller.github.io/ml-workshop-3-of-4/slides/04-model-evaluation.html

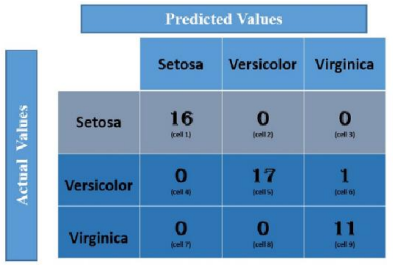
[7] https://amueller.github.io/ml-workshop-3-of-4/slides/04-model-evaluation.html#1
[8] https://scikit-learn.org/stable/auto_examples/model_selection/plot_precision_recall.html#sphx-glr-auto-examples-model-selection-plot-precision-recall-py

- A *confusion matrix* is often used that summarizes the performance in a tabular form. We can see the confusion matrix for the previous classifiers we used so far:




```
from sklearn.metrics import confusion_matrix
confusion_matrix(y_test, knn_nn3.predict(X_test))
```

```
    array([[16,  0,  0],
           [ 0, 22,  1],
           [ 0,  3, 18]], dtype=int64)
```

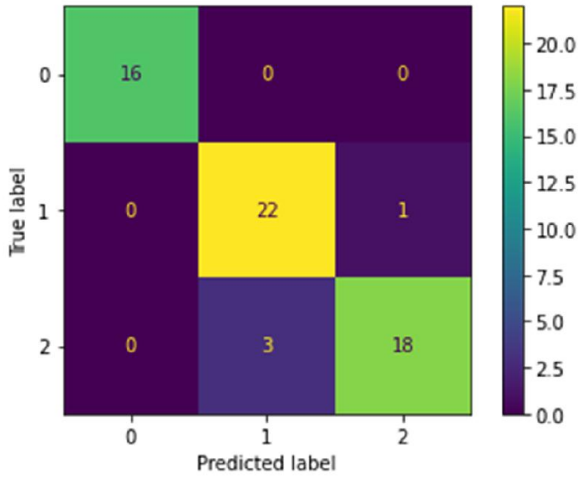



```
confusion_matrix(y_test, rnn.predict(X_test))
```

```
    array([[16,  0,  0],
           [ 5,  9,  9],
           [ 0,  1, 20]], dtype=int64)
```

- We can visualize the confusion matrix using matplotlib.

```
# Confusion Matrix visualization
from sklearn.metrics import ConfusionMatrixDisplay
clf = KNeighborsClassifier(n_neighbors=3)
clf.fit(X_train, y_train)
predictions = clf.predict(X_test)
cm = confusion_matrix(y_test, predictions, labels=clf.classes_)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=clf.classes_)
disp.plot()
plt.show()
```



- Finally, the classification report can be generated and displayed as well.

```
from sklearn.metrics import classification_report
print(classification_report(y_test, knn_nn3.predict(X_test)))
```

	precision	recall	f1-score	support
0	1.00	1.00	1.00	16
1	0.88	0.96	0.92	23
2	0.95	0.86	0.90	21
accuracy			0.93	60
macro avg	0.94	0.94	0.94	60
weighted avg	0.94	0.93	0.93	60