ML for everyone (chapter3 - starting with classification)

April 5, 2022

[2]:
```
from mlwpy import *
%matplotlib inline
```

1 Classification Tasks

If there are only two target classes for output, we can call a learning task binary classification. You can think about {Yes, No},{Red, Black}, or {True, False} targets. Very often, binary problems are described mathematically using $\{-1, +1\}$ or $\{0, 1\}$. Computer scientists love to encode{False, True} into the numbers {0, 1} as the output values, With more than two target classes, we have a multiclass problem.

There is two types of classifiers: * Build a model of how likely the outcomes are - the output will be somthing like we are 90% sure that the patient is sick * pick the most likely outcome - the output will be somthing like {sick or healthy}

2 Simple calssification dataset {Iris dataset}

Each row describes one iris (flower) in terms of the length and width of that flower's sepals and petals . So, we have four total measurements per iris. Each of the measurements is a length of one aspect of that iris. The final column, our classification target, is the particular species—one of three—of that iris: `setosa, versicolor, or virginica`.

[3]:
```
iris = datasets.load_iris()
```

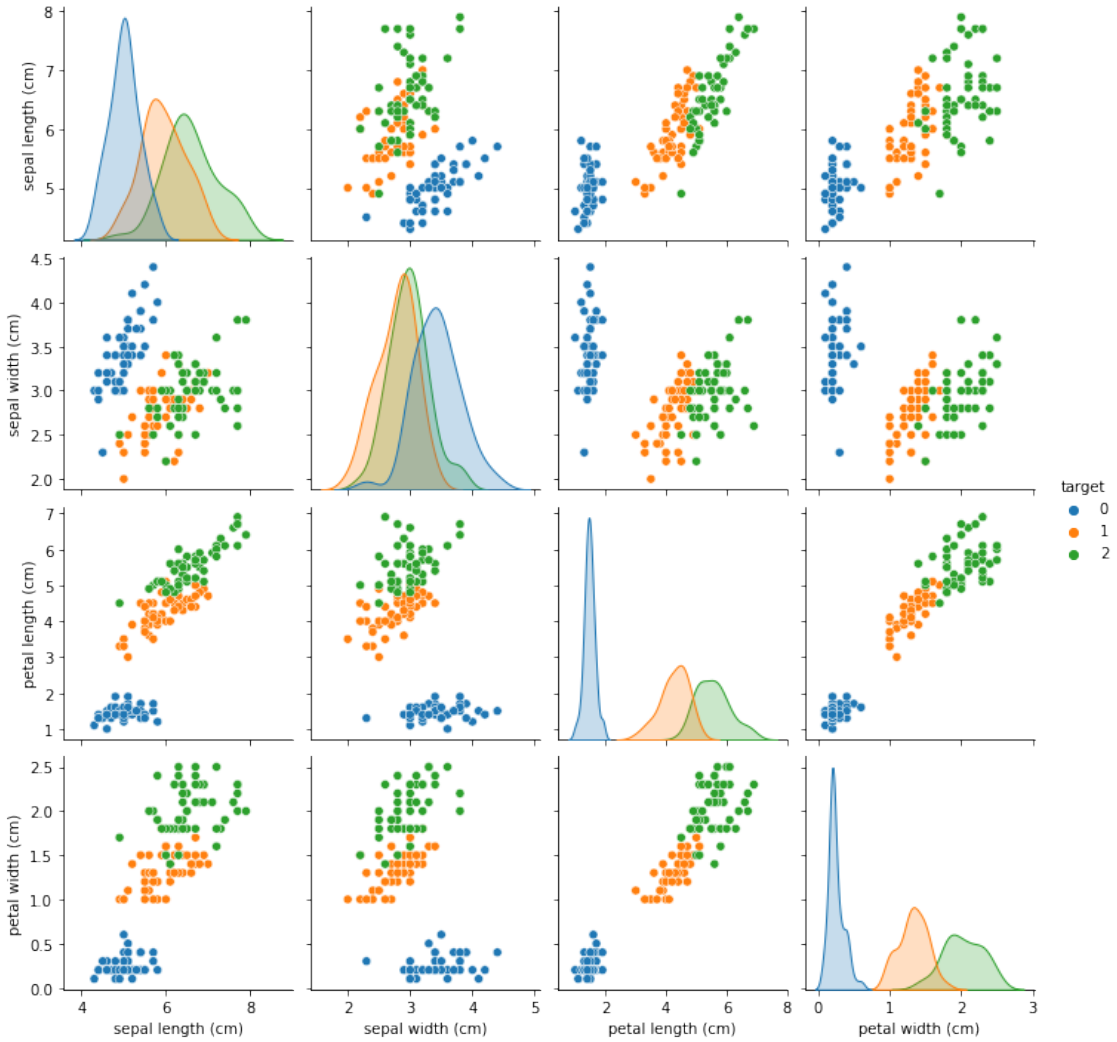
[4]:
```
iris.keys()
```

[4]:
```
dict_keys(['data', 'target', 'frame', 'target_names', 'DESCR', 'feature_names',
'filename'])
```

[5]:
```
iris_df = pd.DataFrame(iris.data, columns=iris.feature_names)
iris_df['target'] = iris.target
display(pd.concat([iris_df.head(3),iris_df.tail(3)]))
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
0	5.1000	3.5000	1.4000	0.2000	0
1	4.9000	3.0000	1.4000	0.2000	0
2	4.7000	3.2000	1.3000	0.2000	0
147	6.5000	3.0000	5.2000	2.0000	2
148	6.2000	3.4000	5.4000	2.3000	2
149	5.9000	3.0000	5.1000	1.8000	2

[6]:
```
sns.pairplot(iris_df, hue='target', palette = "tab10");
```



In several of the plots, the blue group (target 0) seems to stand apart from the other two groups, Which will probably easy to seprate vs the other two, Which species is this?

[7]:
```
print('targets: {}'.format(iris.target_names), iris.target_names[0], sep="\n")
```

```
targets: ['setosa' 'versicolor' 'virginica']
setosa
```

3 Training and Testing: Don't teach to the Test

Let's say we are in a class for machine learning, we want to learn and then apply what we learnt in the real world, one way for evaluating how good we are is Exams to really evalute if we are good the test should be somthing we havn't incounter berfor, That's why when we build a model with our data, we need new data to evaluate how accurate our model is, This is done be splitting the data to `training data` and `test data`

[8]:
```
# simple train-test split (ftrs == features, tgt == target) (test set is 0.25 of the data)
iris_train_ftrs, iris_test_ftrs, iris_train_tgt, iris_test_tgt = skms.train_test_split(iris.data,iris.target,test_size=.25)
print("Train features shape:", iris_train_ftrs.shape)
print("Test features shape:", iris_test_ftrs.shape)
```

```
Train features shape: (112, 4)
Test features shape: (38, 4)
```

4 Evaluation: Grading the Exam

we are simply going to ask, "Is the answer correct?" If the answer is true and we predicted true, then we get a point! If the answer is false and we predicted true, we don't get a point. Every question will count equally for one or zero points. In the end, we want to know the percent we got correct, so we add up the points and divide by the number of questions. This type of evaluation is called accuracy, its formula being $\frac{Numberofcorrectanswers}{Numberofquestions}$. It is very much like scoring a multiple-choice exam.

[9]:
```
answer_key = np.array([True, True, False, True])
student_answers = np.array([True, True, True, True]) # desperate student!
```

We can calculate the accuracy by hand in three steps: 1. Mark each answer right or wrong. 2. Add up the correct answers. 3. Calculate the percent.

[10]:
```
correct = (answer_key == student_answers) # boolean array
num_correct = correct.sum() # True == 1, add them up
print("manual accuracy:", num_correct / len(answer_key))
```

```
manual accuracy: 0.75
```

[11]:
```
# we can use sklearn accuracy_score to do that
print("sklearn accuracy:", metrics.accuracy_score(answer_key, student_answers))
```

```
sklearn accuracy: 0.75
```

5 Simple Classifier #1:Nearest Neighbors, Long Distance Relationships, and Assumptions

One of the simpler ideas for making predictions from a labeled dataset is: 1. Find a way to describe the similarity of two different examples. 2. When you need to make a prediction on a new, unknown example, simply take the value from the most similar known example.

There are many ways we can modify this basic template. We may consider more than just the single most similar example: 1. Describe similarity between pairs of examples. 2. Pick several of the most-similar examples. 3. Combine those picks to get a single answer.

5.1 Defining similarity

We have complete control over what similar means. We could define it by calculating a distance between pairs of examples: `similarity = distance(example_one, example_two)`.Similar things are close ***small distance apart***. Dissimilar things are far away ***large distance apart***. Let's look at three ways of calculating the similarity of a pair of examples: * Euclidean distance * Minkowski distance * Hamming distance.

5.2 The k in KNN

Instead of considering only the nearest neighbor, we might consider some small number of nearby neighbors, n expanded neighborhood protects us from noise in the data, Common numbers of neighbors are 1, 3, 10, or 20 ***What if our closest neighbors are different how do we combine them?***

5.3 Building a k-NN Classification Model

What we want to achive: 1. We want to use 3-NN—three nearest neighbors—as our model. 2. We want that model to capture the relationship between the iris training features and the iris training target. 3. We want to use that model to predict —on previously unseen test examples—the iris target species. 4. Finally, we want to evaluate the quality of those predictions, using accuracy, by comparing predictions against reality.

[20]:
```
# default n_neighbors = 5
knn = neighbors.KNeighborsClassifier(n_neighbors=3)
fit = knn.fit(iris_train_ftrs, iris_train_tgt)
preds = fit.predict(iris_test_ftrs)

# evaluate our predictions against the held-back testing targets
print("3NN accuracy:", metrics.accuracy_score(iris_test_tgt, preds)*100, "%")
```

```
3NN accuracy: 100.0 %
```

6 Simple Classifier #2: Naive Bayes, Probability, and Broken Promises

The key component of `Naive Bayes` is that it treats the features as if they are conditionally independent of each other given the class. Since independence of probabilities plays out mathematically

as multiplication, we get a very simple description of probabilities in a NB model. The likelihood of features for a given class can be calculated from the training data. From the training data, we store the probabilities of seeing particular features within each target class. For testing, we look up probabilities of feature values associated with a potential target class and multiply them together along with the overall class probability. We do that for each possible class. Then, we choose the class with the highest overall probability. However, when we use NB as our classification technique, we assume that the conditional independence between features holds, and then we run calculations on the data. We could be wrong.

[23]:
```
nb = naive_bayes.GaussianNB()
fit = nb.fit(iris_train_ftrs, iris_train_tgt)
preds = fit.predict(iris_test_ftrs)

print("NB accuracy:", metrics.accuracy_score(iris_test_tgt, preds))
```

```
NB accuracy: 1.0
```

7 Simplistic Evaluation of Classifiers

7.1 Learning Performance

[31]:
```
# stand-alone code
from sklearn import (datasets, metrics, model_selection as skms, naive_bayes,␣
 ↪neighbors)

# we set random_state so the results are reproducible
# otherwise, we get different training and testing sets
# more details in Chapter 5
iris = datasets.load_iris()
iris_train_ftrs, iris_test_ftrs, iris_train_tgt, iris_test_tgt = skms.
 ↪train_test_split(iris.data, iris.target,
                                                                                ␣
 ↪       test_size=.90,
                                                                                ␣
 ↪       random_state=42)

models = {'kNN': neighbors.KNeighborsClassifier(n_neighbors=3),
'NB' : naive_bayes.GaussianNB()}

for name, model in models.items():
    fit = model.fit(iris_train_ftrs, iris_train_tgt)
    predictions = fit.predict(iris_test_ftrs)

    score = metrics.accuracy_score(iris_test_tgt, predictions)
    print("{:>3s}: {:0.2f}".format(name,score))
```

```
kNN: 0.96
```

```
NB: 0.81
```

Using only 10% of the data for training the models does fairly well, So, from a learning performance perspective, iris is a fairly easy problem. It is reasonably easy to distinguish the different types of flowers, based on the measurements we have, using very simple classifiers.

8 different dataset (wine dataset)

[35]:
```
# stand-alone code
from sklearn import (datasets, metrics, model_selection as skms, naive_bayes,␣
 ↪neighbors)

wine = datasets.load_wine()
wine.data.shape #
```

[35]:
```
(178, 13)
```

[38]:
```
wine.target_names
```

[38]:
```
array(['class_0', 'class_1', 'class_2'], dtype='<U7')
```

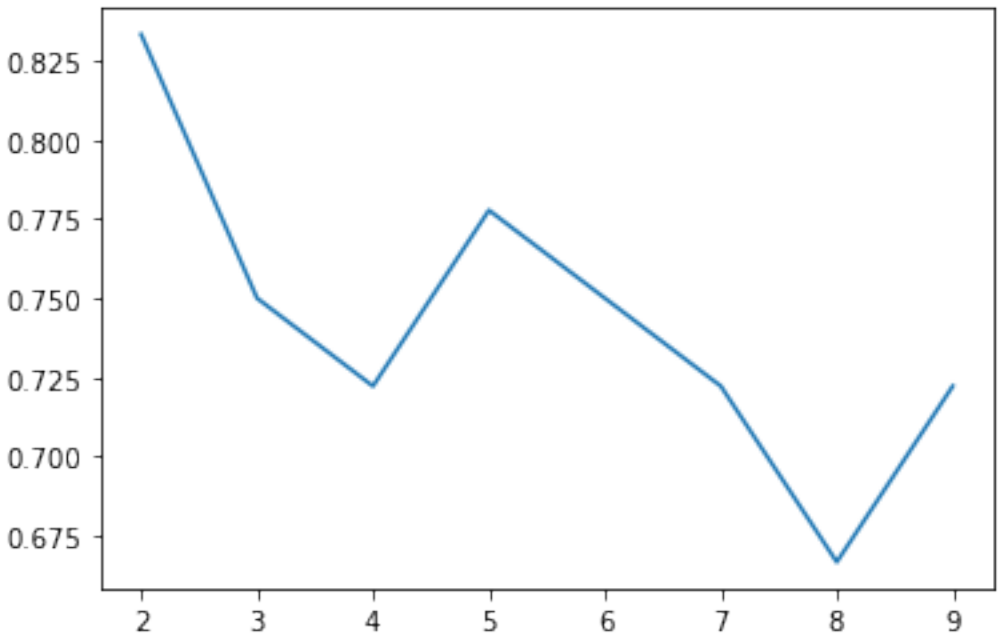
[39]:
```
wine_df = pd.DataFrame(wine.data, columns=wine.feature_names)
wine_df['target'] = wine.target
display(pd.concat([wine_df.head(3),wine_df.tail(3)]))
```

	alcohol	malic_acid	ash	alcalinity_of_ash	magnesium	total_phenols	flavanoids	nonflavano
0	14.2300	1.7100	2.4300	15.6000	127.0000	2.8000	3.0600	
1	13.2000	1.7800	2.1400	11.2000	100.0000	2.6500	2.7600	
2	13.1600	2.3600	2.6700	18.6000	101.0000	2.8000	3.2400	
175	13.2700	4.2800	2.2600	20.0000	120.0000	1.5900	0.6900	
176	13.1700	2.5900	2.3700	20.0000	120.0000	1.6500	0.6800	
177	14.1300	4.1000	2.7400	24.5000	96.0000	2.0500	0.7600	

[64]:
```
X_train, X_test, y_train, y_test = skms.train_test_split(wine.data, wine.
 ↪target, test_size=0.2)

knn = neighbors.KNeighborsClassifier(n_neighbors = 3)
fit = knn.fit(X_train, y_train)
preds = fit.predict(X_test)

# evaluate our predictions against the held-back testing targets
print("3NN accuracy:", metrics.accuracy_score(y_test, preds))
```

```
3NN accuracy: 0.75
```

[62]:
```
# to know which k gives the best accuracy
acc = []
neighbours =  list(range(2, 10))

for k in neighbours:
    knn = neighbors.KNeighborsClassifier(n_neighbors = k)
    fit = knn.fit(X_train, y_train)
    preds = fit.predict(X_test)
    acc.append(metrics.accuracy_score(y_test, preds))

plt.plot(neighbours, acc)
plt.show()
```



[67]:
```
nb = naive_bayes.GaussianNB()
fit = nb.fit(X_train, y_train)
preds = fit.predict(X_test)

print("NB accuracy:", metrics.accuracy_score(y_test, preds))
```

```
NB accuracy: 0.9722222222222222
```