WK3_Machine_Learning_Task

January 24, 2020

1 Machine Learning Task

In this programming task, we'll try our hand at supervised learning. We'll work with a simple dataset and we'll build, evaluate and use two different machine learning models. The dataset that we're going to be using in this programming task is from the botany domain, so it has to do with plants. Note that in this week's Programming Assignment you'll get to work with medical data!

In [ ]:
```
# Supervised Learning
# - Will work w/ simple dataset and build, evaluate and use two different ML models.
```

In [1]:
```
# RECALL:
# Supervised vs. Unsupervised Learning

# Supervised Learning:
# - The ML task of learning a function that maps an input to output
# based on example input-output pairs.
# - Each example is a pair consisting of an input object (typically a vector)
# and a desired output value (supervisory signal).

# - We teach or train the machine using data which is well labeled
# (already tagged with correct answer).
# Afterwards, the machine is provided a new set of examples (data)
# s.t. supervised learning algorithm analyses the training data
# (set of training examples) and produces a correct ooutcome from labeled data.


# Unsupervised Learning:
# - a type of ML algorithm used to draw inferences from datasets
# consisting of input data w/o labeled responses
# Example: Cluster analysis - used in EDA to find hidden patterns/groupings
# in data.

# - The training of machine using information that is neither classified
# nor labeled and allowing the algorithm to act on that information w/o
# guidance.

# Two types:
```

```
# Clustering - discover inherent groupings in the data (customers by purchasing behavi
# Association - Want to discover rules that describe large portions of your data
# (such as people that buy X also tend to buy Y).
```

1.1 Part 1: Importing *scikit-learn*

scikit-learn is the most widely used Python library for machine learning. We first need to tell Python that we're going to be using scikit-learn, with the use of the import command.

```
In [2]: import sklearn
```

Part 2: Familiarising ourselves with the data

1.1.1 Loading the data

scikit-learn comes with a few small standard datasets that do not require to download and to read data from external websites. In this programming task, we are going to be using the **Iris Plants Dataset**. This dataset contains information about different iris flowers, i.e. sepal length, sepal width, petal length, petal width and species (with three possible values for species: setosa, versicolor and virginica). The iris dataset is typically used for supervised learning tasks, and in particular for classification. The idea is that we have measurements (i.e. sepal length, sepal width, petal length and petal width) for which we know the correct species. So if we go out in nature and find some iris flowers and measure their sepal length, sepal width, petal length and petal width, then we can use the iris dataset to predict which species each flower belongs to. Nice, ha? And since there are three possible values for the iris species, it's a classification task.

Let's load the dataset with the use of the *load_iris* function. We'll call it *iris_dataset* (but you could call it anything you want).

```
In [3]: from sklearn.datasets import load_iris
        iris_dataset = load_iris()
```

1.1.2 Getting a sense of the data

The *iris_dataset* object that is returned by *load_iris* is a *Bunch* object, which contains some information about the dataset, as well as the actual data. Bunch objects are very similar to dictionaries (we were introduced to dictionaries in Week 1) and they contain keys and values.

Run the code below to print the keys.

```
In [4]: print("Keys of iris_dataset: ", iris_dataset.keys())
```

```
Keys of iris_dataset:  dict_keys(['data', 'target', 'target_names', 'DESCR', 'feature_names'])
```

There are five types of information in the dataset: * DESCR * feature_names * target_names * data * target

Let's have a closer look at each one of them.

DESCR is a short description of the dataset. Run the code below to get an extract of the first 200 characters. If you want to get a bigger extract, all you need to do is change *200* to a larger number.

```
In [5]: # DESCR - short description of dataset.
        # Will extract first 200 characters.
        print(iris_dataset['DESCR'][:200] + "\n.......")
```

```
Iris Plants Database
====================

Notes
-----
Data Set Characteristics:
    :Number of Instances: 150 (50 in each of three classes)
    :Number of Attributes: 4 numeric, predictive attributes
...
```

feature_names corresponds to the names of all the features in the dataset, in other words all the variables that we take into account when building our machine learning model. Run the code below to print the names of all features.

```
In [6]: # Feature_names - names of all features in dataset.
        # All the variables taken into account when building our ML model.

        print("Feature names: ", iris_dataset['feature_names'])
```

```
Feature names:  ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm
```

target_names corresponds to the class labels. By running the code below, we can see that there are three class labels: 'setosa', 'versicolor' and 'virginica'.

```
In [7]: # target_names - class labels

        print("Target names: ", iris_dataset['target_names'])
```

```
Target names:  ['setosa' 'versicolor' 'virginica']
```

The actual data is contained in the *data* and *target* fields. *data* contains the values for the different features, e.g. sepal length.

Run the code in the next cell to get the shape of *data*.

```
In [8]: # Shape of data

        print(iris_dataset['data'].shape)
```

```
(150, 4)
```

We can see that we have data for 150 iris flowers. For each flower case we have 4 features.

Run the code in the next cell to get the first three rows in *data*.

```
In [9]: # First 3 rows in data

        print("First three rows of data:\n", iris_dataset['data'][:3])
```

```
First three rows of data:
 [[ 5.1  3.5  1.4  0.2]
 [ 4.9  3.   1.4  0.2]
 [ 4.7  3.2  1.3  0.2]]
```

According to this output, we get the following values for the first flower: * sepal length (cm): 5.1 * sepal width (cm): 3.5 * petal length (cm): 1.4 * petal width (cm): 0.2

Small challenge: What if you wanted to get the first 6 rows of data? Write your code below and run it!

```
In [10]: print("First three rows of data:\n", iris_dataset['data'][:6])
```

```
First three rows of data:
 [[ 5.1  3.5  1.4  0.2]
 [ 4.9  3.   1.4  0.2]
 [ 4.7  3.2  1.3  0.2]
 [ 4.6  3.1  1.5  0.2]
 [ 5.   3.6  1.4  0.2]
 [ 5.4  3.9  1.7  0.4]]
```

Run the code in the next two cells to get the shape of *target* and the first two elements.

```
In [11]: # Shape of target/first 2 elements:

         print("Shape of target: ", iris_dataset['target'].shape)
```

```
Shape of target:  (150,)
```

```
In [12]: print("First two elements in target: ", iris_dataset['target'][:2]) # Contains specie
```

```
First two elements in target:  [0 0]
```

We can see that *target* contains the species for each of the 150 iris flowers in the database. The species of the first two flowers is setosa, as 0 corresponds to setosa, 1 to versicolor and 2 to virginica. (How de we know this? It is a convention that elements in *target_names* appear in an increasing order, starting from 0.)

Part 3: Splitting our dataset into training data and test data

Before using our model for previously unseen iris flowers, we need to know how well it performs. To do this, we split our labelled data in two parts: i) a training dataset that we use for building the model, and ii) a test dataset that we use for testing the accuracy of our model. We do this with the use of the *train_test_split* function, which shuffles the dataset randomly, and by default extracts 75% of the cases as training data and 25% of the cases as test data.

Run the code below to split the iris dataset into training and test data.

```
In [ ]: # Split dataset into training and testing data:

        # Before using our model for previously unseen iris flowers, we need to know how well

        # We must split our labelled data into two parts:
        # i) training dataset we use to BUILD model
        # ii) test dataset that we use for testing ACCURACY of model

        # train_test_split - shuffles dataset randomly and extracts 75% of cases as training a

        # 75 % used to build model (to learn how to predict)
        # 25 % used to test accuracy
```

```
In [14]: from sklearn.cross_validation import train_test_split
         X_train, X_test, y_train, y_test = train_test_split(
             iris_dataset['data'], iris_dataset['target'], random_state=0) # random state used
```

This is standard nomenclature. *X* corresponds to data (as in *data* in *iris_dataset*) and *y* to labels (as in *target* in *iris_dataset*). So, for the training dataset we get *X_train* and *y_train*, while for the test dataset we get *X_test* and *y_test*.

Note that if you wanted to split a different dataset called "my_dataset" into a training and a test dataset, then all you would need to do is substitute "iris_dataset" with "my_dataset" in the code above.

By setting *random_state=0* we are making sure that, even though our dataset is randomly shuffled by the *train_test_split* function, we can reproduce our results by using the same fixed seed for the random number generator (in this case 0). So if in the future you want to reproduce the same training and test data, all you need to do is use *random_state=0*.

Run the code below to get the shape of *X_train* and *y_train*.

```
In [17]: print("X_train shape: ", X_train.shape)
         print("y_train shape: ", y_train.shape)
```

```
X_train shape:  (112, 4)
y_train shape:  (112,)
```

```
In [18]: print("X_test shape: ", X_test.shape)
         print("y_test shape: ", y_test.shape)
```

```
X_test shape:  (38, 4)
y_test shape:  (38,)
```

Discussion prompt: What do the outputs from the two previous cells mean? Post your thoughts in the discussion forums!

Note that it is good practice to visualise our data to get a sense of how different features are related or to spot any abnormalities. We will not do this in this programming task so as to keep things simple and save time, but it is worth keeping in mind for the future.

```
In [ ]: # This is the dimensions of the test and training data. y should be (38, ) indicating
        # X_test is (38, 4) meaning there are 38 instances and 4 features used to train the da
```

Part 4: Creating our first model: K Nearest Neighbours

We will now learn how to build a classification model for the iris dataset with the use of the k nearest neighbours algorithm.

1.1.3 Building the model

To build a k nearest neighbours model, we will use the *KNeighborsClassifier* class from the *sklearn.neighbors* module.

Run the code below to create a *KNeighborsClassifier* object called *knn* (but we could give it any name we want). Note that *n_neighbors=1* is setting the number of nearest neighbours to 1.

```
In [20]: # Build our first model with KNN:

         # Using KNeighborsClassifier from sklearn.neighbors module.
         from sklearn.neighbors import KNeighborsClassifier
         knn = KNeighborsClassifier(n_neighbors=1)
```

Run the code below to build the model on the training set, i.e. *X_train* and *y_train*.

You can ignore the output for now.

```
In [21]: # Build/Fit a model on training set using .fit().

         knn.fit(X_train, y_train)

Out[21]: KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                    metric_params=None, n_jobs=1, n_neighbors=1, p=2,
                    weights='uniform')
```

1.1.4 Evaluating the model

We will now use the test dataset to evaluate the accuracy of our model. We can do this with the use of the *score* method, as shown below.

```
In [22]: # Use .score() to evaluate accuracy of model:

         print("Test set score: ", knn.score(X_test, y_test))

Test set score:  0.973684210526
```

The code in the next cell contains a variation of the previous code, in case you want to get the value of *knn.score(X_test, y_test)* rounded to three decimal places. If you wanted it rounded to two decimal places, then all you would need to do is change *{:.3f}* to *{:.2f}*.

```
In [23]: # Round to three decimal places using "{:.3f}".format().

         print("Test set score rounded to three decimal places: {:.3f}".format(knn.score(X_tes
```

```
Test set score rounded to three decimal places: 0.974
```

How is the accuracy of our model calculated? Essentially, our model is used to make predictions for *X_test* and the values predicted are compared to the actual labels in *y_test*.

1.1.5 Using the model to make predictions

We will now use our model to make a prediction about a previously unseen iris flower case. We will first import the numpy libary, then we will specify the previously unseen iris flower case (we'll call it *X_unseen*) and finally we will use the *predict* method on *X_unseen* to get the prediction (we'll call the result *prediction*, but we could use any name we want).

```
In [24]: import numpy as np

In [25]: # Use numpy array to create a flower case and predict X_unseen to get prediction.

         X_unseen = np.array([[5.3, 2.7, 1, 0.3]])

In [33]: # Here, we created our model and automatically made predictions of X_unseen to be set

         prediction = knn.predict(X_unseen)

         print("Prediction label: ", prediction)
         print("Predicted target name: ", iris_dataset['target_names'][prediction])

Prediction label:  [0]
Predicted target name:  ['setosa']
```

According to this output, the prediction for case *X_unseen* is setosa.

1.1.6 Tweaking the model

We can play around with the model to try different numbers of k nearest neighbours, e.g. 3 or 4.

Challenge: Provide some code below to build and evaluate such a model. All you need to do is reuse and modify parts of the code above.

```
In [31]: # K = 4
         knn_4 = KNeighborsClassifier(n_neighbors=4)

         knn_4.fit(X_train, y_train)

Out[31]: KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                    metric_params=None, n_jobs=1, n_neighbors=4, p=2,
                    weights='uniform')

In [34]: # Evaluate model:

         print("Test set score: ", knn_4.score(X_test, y_test))
```

```
Test set score:  0.973684210526
```

```
In [35]: # Make prediction with new model after tweaking:

         prediction = knn_4.predict(X_unseen)

         print("Prediction label: ", prediction)
         print("Predicted target name: ", iris_dataset['target_names'][prediction])
```

```
Prediction label:  [0]
Predicted target name:  ['setosa']
```

```
In [ ]: # Still setosa!
```

Part 5: Creating a different model: Decision Trees

We will now learn how to build a classification model for the iris dataset with the use of the decision tree classifier.

Important note: We would normally use the same training and test data as before, so we would reuse *X_train*, *X_test*, *y_train* and *y_test* (so as to compare the results of the K Nearest Neighbours and Decision Tree models). However, in this programming task we will re-split the iris dataset into training and test data, in order to illustrate how we can get a new version by using a different fixed seed (in this case, 7). If you plan to use this notebook as a template for future machine learning projects, then you can delete the next cell.

```
In [36]: # Build classification model using Decision Tree Classifier

         from sklearn.cross_validation import train_test_split
         X_train, X_test, y_train, y_test = train_test_split(
             iris_dataset['data'], iris_dataset['target'], random_state=7)
```

To build a decision tree model, we will use the *DecisionTreeClassifier* class from the *sklearn.tree* module.

Run the code below to create a *DecisionTreeClassifier* object called *tree* and to fit the model on the training set, (i.e. *X_train* and *y_train*). You can ignore the information outputed.

Note that the decision tree classifier algorithm contains some randomness aspects (explaining these is beyond the scope of this course), so by setting *random_state=12* we can reproduce our results by using the same fixed seed for the random number generator (in this case 12).

```
In [37]: # Use DecisionTreeClassifier
         # Create DecisionTreeClassifier object called tree
         # Fit model on training set.

         from sklearn.tree import DecisionTreeClassifier
         tree = DecisionTreeClassifier(random_state=12)
         tree.fit(X_train, y_train)
```

```
Out[37]: DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
                     max_features=None, max_leaf_nodes=None,
                     min_impurity_decrease=0.0, min_impurity_split=None,
                     min_samples_leaf=1, min_samples_split=2,
                     min_weight_fraction_leaf=0.0, presort=False, random_state=12,
                     splitter='best')
```

Run the code below to evaluate the accuracy of the decision tree model that we just built. We'll distinguish between accuracy on the training set and accuracy on the test set.

```
In [38]: # Evaluate accuracy of decision tree model
         print("Accuracy on training set: ", tree.score(X_train, y_train))
         print("Accuracy on test set: ", tree.score(X_test, y_test))

         # - 100% accuracy of training dataset - means our decision tree is overfitting the tr

         # - To avoid overfitting (and hopefully improve accuracy), we can stop before the ent

         # - Can do this by setting the max depth of the tree.
```

```
Accuracy on training set:  1.0
Accuracy on test set:  0.894736842105
```

The decision tree built has accuracy 100% on the training dataset. This means that our decision tree is over-fitting the training data.

In order to avoid overfitting (and hopefully improve the accuracy of the model on test data), we can stop before the entire tree is created. We can do this by setting the maximal depth of the tree.

Run the code below to create a new version of the tree with maximal depth 3. Note that the only difference to the code in the previous cell is *max_depth=3*.

```
In [42]: # Create new version w/ max_depth = 3

         from sklearn.tree import DecisionTreeClassifier
         tree = DecisionTreeClassifier(max_depth=3, random_state=12)
         tree.fit(X_train, y_train)

         print("Accuracy on training set: ", tree.score(X_train, y_train))
         print("Accuracy on test set: ", tree.score(X_test, y_test)) # NOTE: Lower accuracy on
```

```
Accuracy on training set:  0.991071428571
Accuracy on test set:  0.921052631579
```

The new decision tree has lower accuracy on the training dataset, but higher accuracy on the test dataset.

We will now use our decision tree to make a prediction for the previously unseen iris case *X_unseen*, which was defined earlier in this notebook.

```
In [43]: # Use decision tree to make a prediction for previously unseen iris case X_unseen.

         prediction = tree.predict(X_unseen)

         print("Prediction label: ", prediction)
         print("Predicted target name: ", iris_dataset['target_names'][prediction])
```

```
Prediction label:  [0]
Predicted target name:  ['setosa']
```

```
In [ ]: # NOTE: Prediction is in line w/ prediction from KNN classifier.
```

According to this output, the prediction for case *X_unseen* is setosa. This prediction is in line with the prediction that we got using the K Nearest Neighbours classifier.

Part 6 (Optional): Practise further

We highly recommend that you practise further with what you've learnt in this programming task. Here are some ideas to get you started: - Build a K Nearest Neighbours model for a different number of neighbours and evaluate it. - Build a Decision Tree model with a different maximal depth and evaluate it. - Build a Decision Tree model on the original training data (i.e. for the original split of data with random_state=0) and evaluate it.