10.009 The Digital World

Term 3, 2019
Problem Set 10 (for week 10)
Last Update: 1 April 2019

Due dates:
This problem set only has Cohort problems, which are due Tuesday 2359 hours of Week 11.

Please note:
Attempt this problem set using the Jupyter notebook. It is much more convenient for this problem set.

Objectives

1. Use the matplotlib library to visualize data using a scatter plot, bar chart, box plot and histogram
2. Explain the terms feature, target and record
3. Obtain and interpret the confusion matrix for binary classification
4. Explain the k-Nearest Neighbours (kNN) classification model
5. Using the scikit-learn library and the breast cancer dataset
    a. Implement kNN classification model
    b. Implement linear regression model

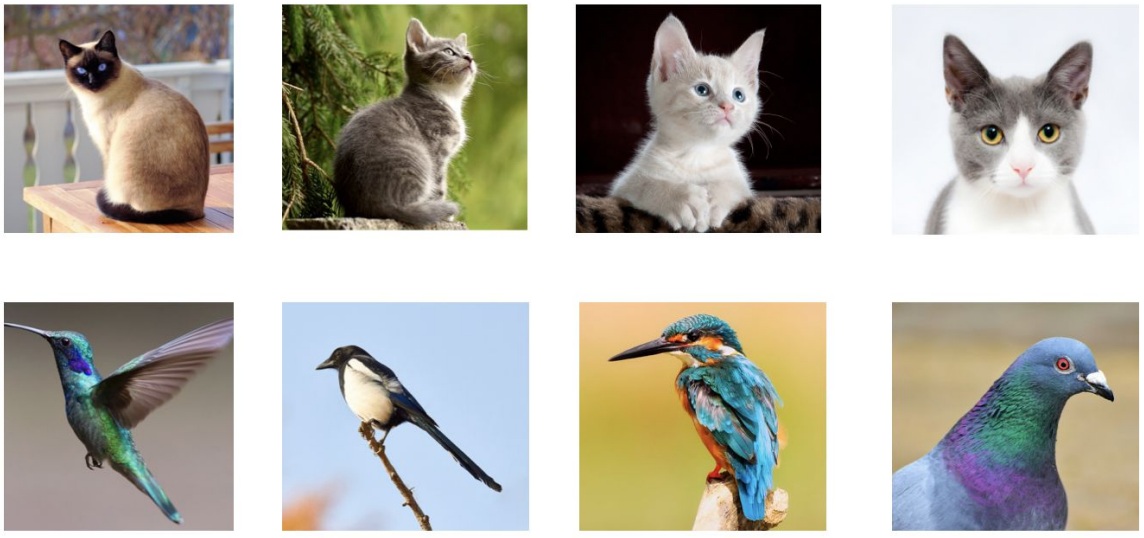
Cohort Session

Attention.

- All numpy arrays that are to be returned by functions are to be *two-dimensional arrays*.
- The order of the questions discussed in during the cohort sessions is decided by your instructor and may be different from the order of the questions below.
- Linear Regression is discussed in Questions 5, 6
- k-Nearest Neighbours is discussed in Questions 1 - 4, 7

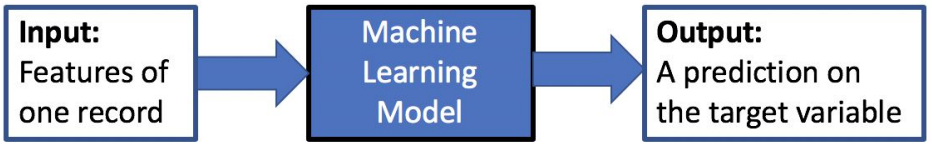
1. **Confusion Matrix.** Before you do any machine learning, you should understand what a **confusion matrix** is.

   For this exercise, we will limit ourselves to categorical target variables containing only two categories.
   Suppose that you had four images of birds and four images of cats. You can think of each image as a record, and the target variable is either **bird** or **cat**.



Images are taken from various free stock photo sites.

Suppose you also had a computer program that takes in data from an image and is able to predict what object is within that image. This is essentially what a machine learning model does. A machine learning model takes in the features in each record and makes a prediction on the target variable.



This prediction is then compared against the actual target variable. The results for all the records in the dataset are summarized in the confusion matrix. From this matrix you can calculate measures such as the accuracy and the sensitivity. These measures tell you how well your model performs in its prediction task.

To keep things simple, let us assume that for our machine learning model

- only images of actual birds or cats are given to it as input
- the model can only tell you whether the image is a bird or cat

a. Let's begin with a pen-and-paper exercise. The actual target variables are shown in the variable named `actual`. The predictions made by a machine learning model are shown in `predicted`. From the data below, complete the confusion matrix.

```
actual = ['cat', 'cat', 'cat', 'cat', 'bird', 'bird','bird','bird']
predicted = ['cat', 'cat', 'bird', 'bird', 'cat', 'bird', 'bird', 'bird'
]
```

	Predicted bird	Predicted cat
Actual bird		
Actual cat		

How many records were predicted correctly?
How many records were predicted wrongly?
How many 'bird' were wrongly classified?
How many 'cat' were wrongly classified?

b. A sample function `get_metrics()` is given below that takes in three inputs, a list of actual targets, a list of predicted targets and the labels in the order you want.

In the data above, there are two categories, 'bird' and 'cat'. One of them would have to be designated as the **positive case**. The positive case is what you really want to predict, or which category is more important.

Suppose you have a dataset of fraudulent and non-fraudulent credit card transactions. It would certainly be more important for you to identify fraudulent transactions. Hence we would treat 'fraudulent' as the positive case.

Let us treat ‘cat’ as the positive case. The `labels` parameter should then specify the negative case, followed by the positive case.

```
labels = ['bird', 'cat' ]
```

Conversely, suppose you are more interested in birds. Then you would treat ‘bird’ as the positive case and the `labels` parameter should be specified as follows.

```
labels = ['cat', 'bird']
```

Run the following script to check your pen-and-paper exercise.

```
from sklearn.metrics import confusion_matrix

def get_metrics(actual_targets, predicted_targets, labels):
    c_matrix = confusion_matrix(actual_targets, predicted_targets,
labels)
    return c_matrix

actual = ['cat', 'cat', 'cat', 'cat', 'bird', 'bird','bird','bird']
predicted = ['cat', 'cat', 'bird', 'bird', 'cat', 'bird', 'bird', 'bird'
]
labels = ['bird', 'cat']
print(get_metrics(actual, predicted, labels) )
```

c. From the confusion matrix, we may calculate the following metrics to help you evaluate the outcome of your machine learning algorithm. The following list is not exhaustive, but are some of the more important ones:
   - **Accuracy** = total correct predictions / total records
   - **Sensitivity** = total correct positive cases / total positive cases. (This metric is also known as **recall**).
   - **False positive rate** = total false positives / total negative cases. The false positives are the actual negative cases that have been predicted to be positive.

   Modify `get_metrics()` to return a dictionary containing the confusion matrix, as well as the results of these metrics above.
   Round the metrics above to three decimal places.
   Submit your code to Vocareum.

   With the bird/cat data as above, the expected output is as follows.

```
{'confusion matrix': array([[3, 1],
       [2, 2]]), 'total records': 8, 'accuracy': 0.625, 'sensitivity':
0.5, 'false positive rate': 0.25}
```

2. **The Five-Number Summary.** A simple summary of each numerical feature in the dataset is the five number summary. It is the numerical version of the box plot. Write a function `five_number_summary()` that takes in a numpy array and returns a list of dictionaries. Each dictionary contains the five number summary for the corresponding column.

   The function definition is given below and some suggested numpy functions are also given. **x** should be a **2D-numpy array**. If **x** is otherwise, return **None**.

```
def five_number_summary(x):
    np.max(x)
    np.min(x)
    np.percentile(x,25)
    #and so on
```

   Test your function using the following script.

```
first_column = bunchobject.data[:, [1] ]
print( five_number_summary(first_column) )
```

   The expected output is as follows. (Your own output may have floating point errors, which is ok. **You are not required to do any rounding in the rest of this problem set**).

```
[{'minimum': 9.71, 'first quartile': 16.17, 'median': 18.84, 'third
quartile': 21.80, 'maximum': 39.28}]
```

   As an exercise, produce the actual boxplot and compare it with the output above.

   This function should also be able to take in more than one column. Hence the following test script should work too.

```
col_no = [0,1,2]
some_columns = bunchobject.data[:,col_no]
print( five_number_summary(some_columns) )
```

3. **Normalization.** Some machine learning models require data to be normalized, so that all features in the dataset have the same order of magnitude. One way is the min/max normalization. The largest value in the feature is assigned a value of 1 and the smallest value in the feature is assigned a value of 0. Intermediate values are calculated by linear interpolation.

   Write a function `normalize_minmax()` that takes in a 2D-numpy array, normalizes it using the min/max normalization and returns the normalized array.

   The function header is given to you. **`data`** should be a **2D-numpy array**. If **`data`** is otherwise, return **`None`**.

```
def normalize_minmax(data):
```

   The following is a test case. We check the five number summary to see that the values have been normalized.

```
first_column = bunchobject.data[:,[1]]
first_column_norm = normalize_minmax(first_column)
print(five_number_summary(first_column_norm))
```

   The expected output is as follows. Your answer may be slightly different due to floating point error.

```
[{'minimum': 0.0, 'first quartile': 0.21846466012850865, 'median':
0.30875887724044637, 'third quartile': 0.40886033141697664, 'maximum':
1.0}]
```

   Your function should also work if more than one column is input. Hence, for the following test script:

```
cols = [1, 7]
some_columns = bunchobject.data[:,cols]
snorm = normalize_minmax(some_columns)
print('normalized', five_number_summary(snorm))
```

   The expected output is as follows.

```
normalized [{'minimum': 0.0, 'first quartile': 0.21846466012850865,
'median': 0.30875887724044637, 'third quartile': 0.40886033141697664,
'maximum': 1.0}, {'minimum': 0.0, 'first quartile': 0.10094433399602387,
'median': 0.1665009940357853, 'third quartile': 0.36779324055666002,
'maximum': 1.0}]
```

4. **k-Nearest Neighbours model.** Having understood what a confusion matrix says, you are ready to build your first classifier using the k-Nearest Neighbours model.

   Plot a Bar Chart for the target variable

   Before doing so, we should plot a bar chart showing the distribution of categories in the target variable of the breast cancer dataset. Recall that it is useful to understand the balance of classes in the target variable. A bar chart is a helpful visualization of this.

   Write a function `display_bar_chart()` that takes in four inputs. For the first two inputs, see the test script below. The third input is the name of each category. The fourth is an optional title. You need not submit this function.

   The function header is given below.

```
def display_bar_chart(positions, counts, names, title_name='default' ):
```

   Test your function using this script.

```
unique, counts = np.unique(bunchobject.target, return_counts = True)
display_bar_chart(unique, counts, bunchobject.target_names)
```

   You do not need to submit this function to vocareum.
   Go on to the next page

Building your k-Nearest Neighbours classifier
Having seen the boxplot, we are ready to build a k-nearest neighbours classifier. The steps are as follows.

Step 1. Obtain the dataset. You have already seen how to do this.

Step 2. Select the features that are to be included in the dataset. The dataset has 30 features, and for our first analysis, let us select the first 20.

```
feature_list = range(20) #features from column 0 to 19
data = bunchobject.data[:, feature_list]
```

Step 3. Each numerical feature selected is normalized using the min/max normalization.

Step 4. The dataset (which includes the target variable) is divided into two sets, the **training set** and the **test set**. The analyst typically decides the percentage and a typical value is to choose the test set from 40% of the records. The performance of the model is checked using the data from the test set.

This is done using the `train_test_split()` method, which conducts a random sampling from the records to give you the two sets. Read the documentation for details.

```
from sklearn.model_selection import train_test_split

data_train, data_test, target_train, target_test = train_test_split(
data , target , test_size = 0.40, random_state = 42 )
```

Step 5. Select a value of *k* to build the classifier. The classifier is built using the data from the training set.
Step 6. The classifier is then used to make predictions on the target variable in the test set.

A partial set of code for these two steps is given below. Read the documentation to find out how to complete it.

```
clf = neighbors.KNeighborsClassifier(pass)
clf.fit(pass)
target_predicted = clf.predict(pass)
```

Step 7. The results of this classification is reported in the confusion matrix and the various metrics. You have already written a method for this.

These steps can be completed in a single function. Write a function `knn_classifier()` that takes in the following inputs:

- The bunchobject that is obtained after loading the dataset
- A list containing the column numbers of the features to be selected
- The size of the test set as a fraction of the total number of records
- A random number seed to ensure that the results can be repeated
- The value of *k* that is selected.

```
def knn_classifier(bunchobject, feature_list, size, seed, k):

    #step 2
    #step 3
    #step 4
    #step 5
    #step 6
    results = get_metrics(pass) #step7

    return results
```

The following is a test case, where the first 20 features are selected.

```
features = range(20)
results = knn_classifier(bunchobject, features, 0.40, 2752, 3)
print(results)
```

The output is

```
{'confusion matrix': array([[141,   5],
       [  9,  73]]), 'total records': 228, 'accuracy': 0.939,
'sensitivity': 0.89, 'false positive rate': 0.034}
```

Notes:

(1) The choice of features in question 4 and 7 is arbitrary. Methods exist to select features systematically, but we are not discussing this in this problem set.

(2) The choice of *k* in this question is arbitrary. In Question 7, we will see how to choose the value of *k* systematically.

5. **Linear Regression.**

   Create a scatter plot

   To determine whether two features have a linear relationship, the first step is to create a scatter plot and examine it.

   Write a function `display_scatter()` that takes in two numpy vectors, together with optional arguments for the x-axis label, y-axis label and title. The scatter plot is then displayed. The function header is given below.

```
def display_scatter(x,y, xlabel='x', ylabel='y',title_name ='default'):
```

   With the breast cancer dataset, you may use the following script to investigate interesting correlations between the features. The following test script assumes that you have already loaded the dataset into `bunchobject`.

```
x_index = 0
y_index = 3
x = bunchobject.data[:,[x_index] ]
y = bunchobject.data[:,[y_index] ]
x_label = bunchobject.feature_names[x_index]
y_label = bunchobject.feature_names[y_index]

display_scatter(x,y,x_label,y_label)
```

Obtaining the linear regression

Your scatter plot likely suggests that two features seem to have a linear relationship. Using linear regression, we are able to determine the extent to which this is true. We are also able to make predictions of the value of one feature from another. The steps are as follows.

Step 1. Obtain the dataset. You have already seen how to do this.

Step 2. Select the feature to go on the x-axis and on the y-axis. The feature that is on the y-axis can be thought of as the target. You may also use the terms **independent variable** and **dependent variable**. No normalization is needed.

Step 3. The dataset is divided into two sets at random, the training set and the test set. The analyst typically decides the percentage and a typical value is to choose the test set from 40% of the records.

This is done using the `train_test_split()` method. Read the documentation for details.

```
from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split( x , y , test_size =
0.40, random_state = 42 )
```

Step 4. The linear regression model is built using the data from the training set.
Step 5. The model is then used to make predictions on the target variable in the test set.

A partial set of code for these two steps is given below. Read the documentation to find out how to complete it.

```
from sklearn import linear_model
regr = linear_model.LinearRegression()
regr.fit(pass)
y_pred = regr.predict(pass)
```

Step 6. Obtain the coefficients, intercept, mean-squared error and the r2 score. You will need to import the following functions. Read the documentation on how to use them.

```
from sklearn.metrics import mean_squared_error, r2_score
```

These steps can be completed in a single function. Write a function `linear_regression()` that takes in the following inputs:

- The bunchobject that is obtained after loading the dataset
- An integer that represents the column number of the x-variable
- An integer that represents the column number of the target variable
- The size of the test set as a fraction of the total number of records
- A random number seed to ensure that the results can be repeated

The function returns the data in the training set, the predictions made on the test set and a dictionary showing the results of the model (see output below). Submit this function `linear_regression()` to vocareum.

```
def linear_regression(bunchobject, x_index, y_index, size, seed):

    #step 2
    #step 3
    #step 4
    #step 5
    #step 6
    return x_train, y_train, x_test, y_pred, results
```

Also, complete the following function to plot the data that you obtained from your linear regression. You need not submit this function.

```
def plot_linear_regression(x1, y1, x2, y2, x_label='', y_label=''):
    plt.scatter(x1,y1, color='black')
    pass
```

The following is a test case, where column 3 is the target variable and column 0 are considered.

```
x_train, y_train, x_test, y_pred, results =
linear_regression(bunchobject,0,3,0.4,2752)
print(results)
plot_linear_regression(x_train, y_train, x_test, y_pred,
                           bunchobject.feature_names[0],
                           bunchobject.feature_names[3])
```

The output is as follows. Remember to also produce the plot.

```
{'coefficients': array([[ 100.16755386]]),
 'intercept': array([-760.52027342]),
 'mean squared error': 2631.2988797244757,
 'r2 score': 0.97772539335215169}
```

This question continues on the next page.

Interpreting the results

The last step is to think about the results that you have obtained.

- From the r2 score, what can you say about the extent to which both variables are correlated?
- Can you rely on the r2 score alone to make this judgement? (Read about anscombe's quartet).
- Hence, what did you see from the scatter plot?

6. **Multiple Linear Regression.** In the previous question, you noticed that a certain trend could not be reproduced by a pure linear regression, i.e a $y = a_1 x + a_2$ model is not sufficient.

   We can try to improve the fit by including higher order variables e.g. a second order model will look like this: $y = a_0 x^2 + a_1 x + a_2$. Including higher orders of the same independent variable is called **Polynomial Regression** and is a special case of multiple linear regression.

   Modify the function you wrote in Question 5. It will now take in an additional parameter called `order`. `order = 2` means you want to try fitting the data to a second order model like the one above.

   The function header is given below.

```
def multiple_linear_regression(bunchobject, x_index, y_index, order,
size, seed):
```

   Previously, your x-values are contained in a numpy array with one column. With this function, your model will now have multiple inputs. You now need to ensure that you now have a numpy array with the same number of columns as the order specified. Each column will then correspond to data for x, x^2, x^3 and so on.

   Part of the code needed to achieve this is as follows. Complete the missing parts by reading the documentation.

```
from sklearn.preprocessing import PolynomialFeatures

poly = PolynomialFeatures(pass,include_bias=False)
c_data = poly.fit_transform(pass)
```

   The function returns the data in the training set, the predictions made on the test set and a dictionary showing the results of the model (see output below). Please note that for the x-values in each set, you are only required to return the column for x^1 as we need just this data to produce the scatter plot. The higher orders of x are not needed. A reminder that you should return x^1 as a **2D numpy array**.

Submit the function `multiple_linear_regression()` to vocareum.

The following is a test script.

```
x_train, y_train, x_test, y_pred, results =
multiple_linear_regression(bunchobject,0,3,4,0.4,2752)
print(results)
plot_linear_regression(x_train, y_train, x_test, y_pred,
                           bunchobject.feature_names[0],
                           bunchobject.feature_names[3])
```

With this test script, the output is as follows.

```
{'coefficients': array([[ -1.28141031e+02,   1.57502508e+01,
-5.29186793e-01, 7.97220165e-03]]), 'intercept': array([
459.72265999]), 'mean squared error': 145.64415629863078, 'r2
score': 0.998767085595213 99}
```

Run your function for orders from 1 to 4 and tabulate the r2 values and mean-squared error.
Is a higher order always better? What is the best order to choose?

7. **k-Nearest Neighbours (full).** In Question 4, you had your first exposure to a k-Nearest Neighbours classifier, using one value of *k* to make predictions on the test set.

   Before we actually deploy a model to make predictions, we will have to go through a validation process. In Question 4, we arbitrarily selected the value of *k*. How do we know the value of *k* we used in Question 4 is the value that gives the best accuracy?

   The solution is to divide the model into three sets, the (1) Training set, the (2) Validation set, and (3) the Test set.

   The idea is that we make predictions with the classifier on the **validation set** with different values of *k*. For each value of *k*, we record the metric that is important to us (e.g. accuracy). We select the smallest value of *k* that gives us the best performance on this metric.

   You now have the best value of *k*. With this best value of *k*, the classifier is then used to make predictions on the **test set.** This then gives you an idea of how the model will perform on new data.

Steps 1 to 3 remain the same as in Question 4.

Step 4. Divide the model into the three sets, i.e. training, validation and test set. There is no hard and fast rule on the proportions, but a typical split is 60%: 20%: 20%, which we will use in this question. You will have to apply the `train_test_split()` function twice.

```
data_train, data_part2, target_train, target_part2 = train_test_split(
data , target , test_size = 0.40, random_state = 42 )
#now data_part2 and target_part2 contains 40% of your records.
#call train_test_split() to split this into two sets of 20% each
```

Step 5. The classifier is built using the data from the training set.
Step 6. The classifier is then used to make predictions on the target variable in the **validation set**. This is repeated for values of *k* from 1 to a certain number, say 20. This helps us to decide which is the best value of *k* to choose. The pseudocode below shows you how it is done.

```
    for k in range(1,21):
       #get an instance of the classifier for a particular value of k
       #fit the model to the training set (step 5)
       #make a prediction on the validation set (step 6)
       #get the accuracy of the prediction
       #store the accuracy in a list
```

After this, you would have information on the accuracy for each value of *k* stored in a list. You then choose the smallest value of *k* that gives you the best accuracy. Explore the following example code for some hints on how to do this with a list.

```
acc = [1, 3, 5, 5, 5, 4, 2, 2, 3]
max_acc = max(acc)
value = acc.index( max_acc )
```

Step 7. With the best value of *k*, the classifier is used to make predictions on the test set.

Thus far, the test set has not been involved in building the model, nor has it been involved in the validation process.

Hence, the test set is a good proxy for data that has not been 'seen' by the model. Using the model on the test set thus gives you an idea of the model's performance on new data.

This question continues on the next page.

Complete the following function to carry out this process. Submit this function to vocareum.

```
def knn_classifier_full(bunchobject, feature_list, size, seed):
      #step2
      #step3
      #step4
      #step5
      #step6
      #step7
    return out_results
```

Note: The input parameter `size` is the proportion of records that are not in the training set. You may assume that this set is then split equally between the validation set and test set.

`out_results` is a dictionary with three key-value pairs:

- '`best_k`' stores the best value of *k* found
- '`validation set`' stores the dictionary returned by `get_metrics()` for the predictions made on the **validation set** for this best value of *k*
- '`test set`' stores the dictionary returned by `get_metrics()` for the predictions made on the **test set** for this best value of *k*

The following is a test script.

```
features = range(20) #select features in cols 0 to 19
results = knn_classifier_full(bunchobject, features, 0.40, 2752)
print(results)
```

The output of this test script is shown below.

```
{'best k': 4, 'validation set': {'confusion matrix': array([[71,  2],
[ 2, 39]]), 'total records': 114, 'accuracy': 0.965, 'sensitivity': 0.951,
'false positive rate': 0.027}, 'test set': {'confusion matrix': array([[69,
4],        [ 1, 40]]), 'total records': 114, 'accuracy': 0.956,
'sensitivity': 0.976, 'false positive rate': 0.055}}
```

End Of Problem Set