Machine Learning Lab 2

Implementing your first classifier in WEKA

In lab sheet 1 you have:

1. created a new Java project with Weka included as a source;
2. seen how data is stored in the Weka ARFF format, and created test data of your own;
3. loaded ARFF data into your project and used Weka methods to inspect/edit it;
4. built two provided Weka classifiers and used them;
5. written and built your own histogram classifier from the lecture

By the end of lab sheet 2 you will:

1. Have some basic tools to help train and evaluate classifiers;
2. Be able to form a contingency table for test predictions;
3. Have implemented a basic majority class classifier;
4. become familiar with the inner workings of a basic Classifier in Weka, ZeroR;
5. Evaluated some classifiers on the Mosquito problem.

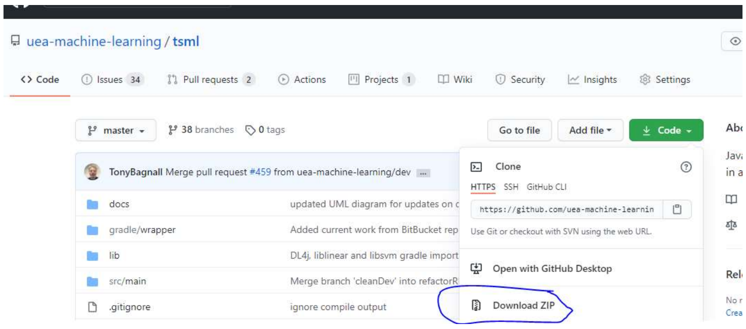
There are also some optional extra sklearn exercises.

Prerequisite:

Last week you used the Weka Jar provided. To complete this lab sheet, you will need to download the UEA TSC repository, a project that extends the Weka library. There will be differences

1. The project is on GitHub. https://github.com/uea-machine-learning/tsml

   If you are not familiar with GitHub, this is a good time to get on top of it. Set up a GitHub account, fork the tsml repo, then clone it locally. Alternatively, just download the zip of the code



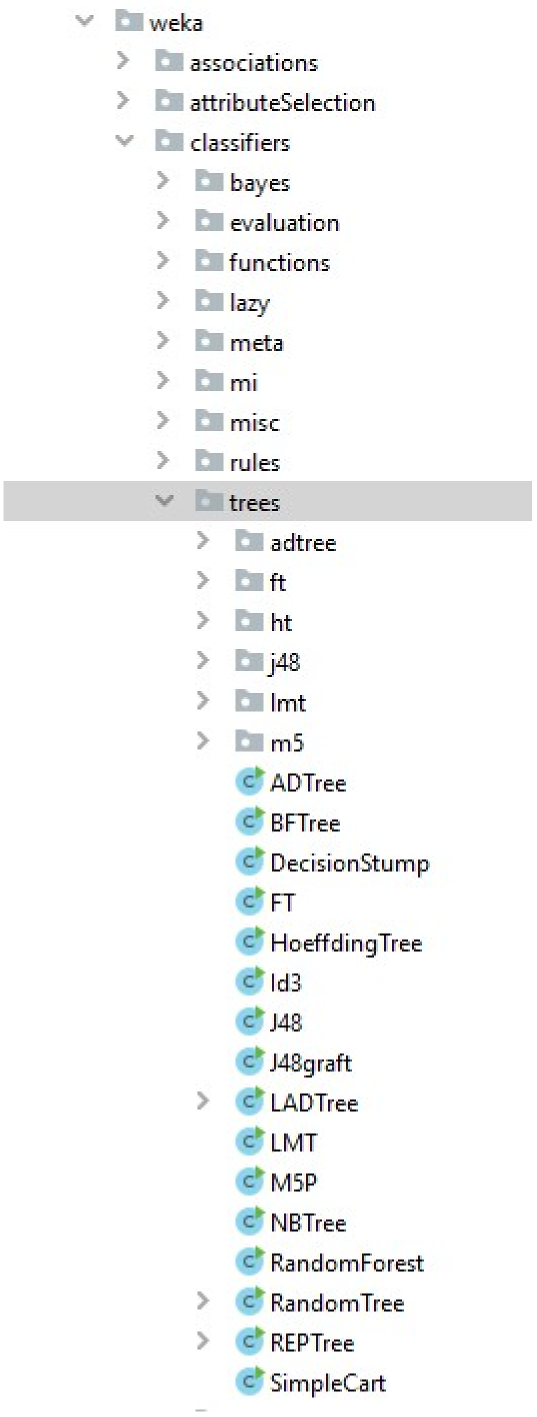
TonyBagnall Merge pull request #459 from uea-machine-learning/dev

docs	updated UML diagram for updates on c
gradle/wrapper	Added current work from BitBucket rep
lib	DL4j, liblinear and libsvm gradle import
src/main	Merge branch 'cleanDev' into refactorR
.gitignore	ignore compile output

```
https://github.com/uea-machine-learnin
```

Use Git or checkout with SVN using the web URL.

2. Move the download to somewhere sensible and unzip.
3. In IntelliJ, open the project. It *should* all install automatically. If you have any questions ask the lab support or raise an issue on the github; if it doesn't work properly we would like to know so we can fix it. It takes quite a while and you should be prompted to set the sdk.
4. To make sure it works, open, go the directory src/main/java/examples, read the code in Ex01 and run the code. All these examples help you work your way around Weka.
5. Go the directory src/main/java/weka. This contains all the Weka source code. The first classifiers we will look at in Week 2 are Decision Trees, which are well represented in Weka


weka
associations
attributeSelection
classifiers
bayes
evaluation
functions
lazy
meta
mi
misc
rules
trees
adtree
ft
ht
j48
lmt
m5
ADTree
BFTree
DecisionStump
FT
HoeffdingTree
Id3
J48
J48graft
LADTree
LMT
M5P
NBTree
RandomForest
RandomTree
REPTree
SimpleCart

7. Create a working area package for yourself under Java, create a class called Lab1 and copy over your code from the first lab sheet. I suggest you use this as your working area for the module

Take a look at the Weka API for the interface Classifier, and relate it to the code http://weka.sourceforge.net/doc.dev/

EXERCISES

Task 1. Write some basic manipulation tools

Write some static helper methods that will help you with later exercises and the coursework. Test all these on the toy data used in lab 1.

a) Measure accuracy

Implement a static method in a new class called **`WekaTools`** in your package to find the accuracy of a classifier on a given test split.

```
double accuracy(Classifier c, Instances test)
```

This method simply iterates over the test data, calls **`classifyInstance`** on each instance, then counts how many predictions are correct. The accuracy of the classifier is then the number correct divided by the number of instances.

b) Load data

Write a static method in **WekaTools** to make loading data tidier

```
Instances loadClassificationData(String fullPath)
```

This method should set the last attribute as the class value. Load the football data and find out the data size (number of instances and attributes). Open the arff to get some idea of what the attributes represent. Decide what to do about any Exceptions!

c) Split data

Write a static method in **WekaTools** to split the data into train and test sets, with an argument that gives the proportion of instances that should be in the test set

```
Instances[] splitData(Instances all, double proportion)
```

The first Instances should be train, and the second should be test.

```
Instances[] split=new Instances[2];
//Copy all data to split[0]
split[0]=new Instances(all);
//Copy header data but no instances to split[1]
split[1]=new Instances(all,0);
```

Then move `proportion*all.numInstances()` from `split[0]` to `split[1]` and return the array. Please note that this method of splitting the data is not very good. If the original data is ordered by class value, for example, then the split will not be representative of the class distribution of the original data. One way to deal with this is simply to randomise the data before splitting. Look in the API for a way to do this with `Instances.`

d) Class distribution

Write a static method in WekaTools that finds the class distribution of an Instances object.

```
double[] classDistribution(Instances data)
```

so, for example, a three class problem with 200 class 0, 500 class 1 and 300 class 2 should return a double array of length 3, with the values 0.2, 0.5 and 0.3.

Task 2. Find Contingency Table/Confusion Matrix

A contingency table is a way of representing the predictions of a classifier with more detail than just the accuracy. We cover them in detail in the Evaluation lecture (week 4), but they are simple to understand. Suppose we have an array of actual classes (assume a two class problem)

Actual:	0,0,1,1,1,0,0,1,1,1
Predicted:	0,1,1,1,1,1,1,1,1,1

The accuracy is 0.7 (error is 0.3). A confusion matrix displays the number of predicted conditional on the actual class. So when the actual class is 0, the above classifies one class as zero and three cases as one.

	Actual	
Predicted	0	1
0	1	0
1	3	6

a) **write a method that takes works out a confusion matrix**

```
int[][] confusionMatrix(int[] predicted, int[]
actual)
```

test it on the above data

b) **get all predicted class values. `classifyInstance`** classifies a single instance. Write a method that gets the prediction from a classifier (assumed to be already built) for all the cases in an Instances object

```
int[] classifyInstances(Classifier c, Instances
test)
```

c) **Get all actual class values.** Actual class values are stored with the instances. Write a method that extracts the class values from a set of instances

```
int[] getClassValues(Instances data)
```

d) Form a contingency table. Now use one of your classifiers from last week and the football problem to form a confusion matrix on a single test fold. Then pick any classifier in Weka and do the same for the problem ItalyPowerDemand (look at Ex01 to see how to load it).

Task 3. Implement a majority class classifiers

One problem with just reporting accuracy is that it does not reflect the default class distribution. 90% accurate sounds good, but if 95% of the data is one class it is not very good at all. Because of this, a minimum level “straw man” is the accuracy of a classifier that predicts the most frequent class in the train data for all instances in the test data. So buildClassifier simply works out and stores the most commonly occurring class, and classifyInstance predicts this class for all instances.

Implement a MajorityClass classifier, and test it one of the data sets from a previous lab, using your code to generate a single train test split. Calculate the accuracy. Then generate the confusion matrix for a single test split (all the data should be in one column).

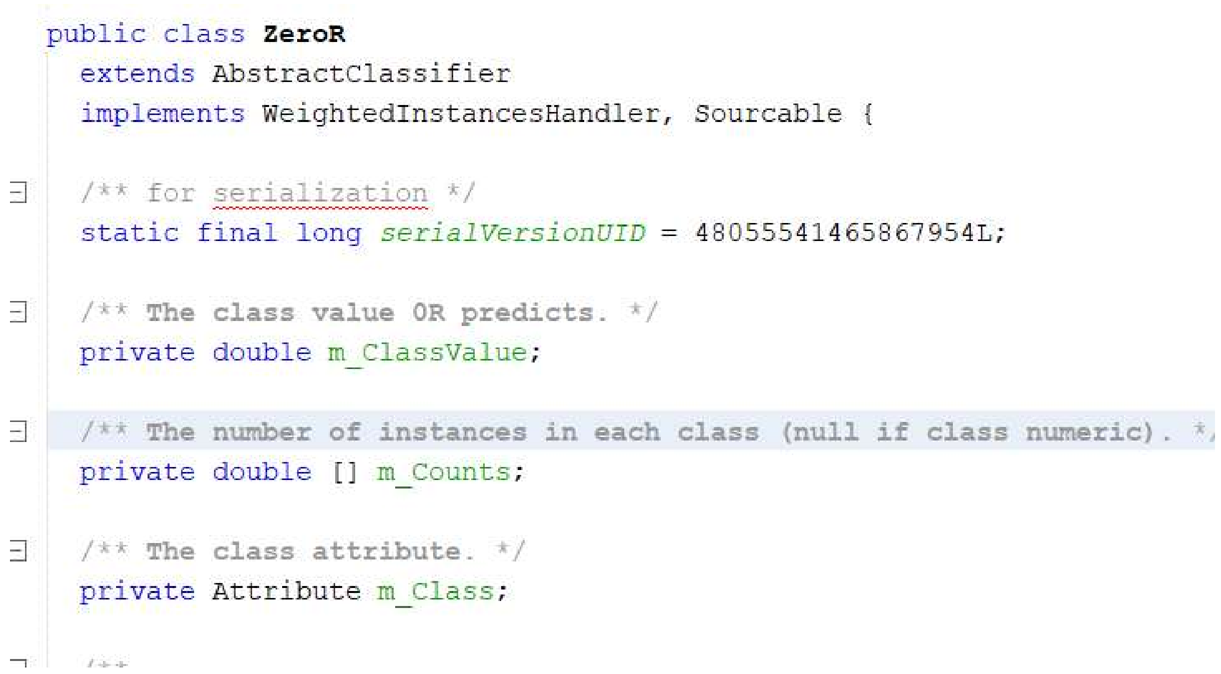
Task 4. Explore the built in majority class classifier ZeroR

ZeroR is the built in version of the majority class classifier in weka.classifiers.rules package.

a) Use it and verify the confusion matrix is the same as yours on the same data.

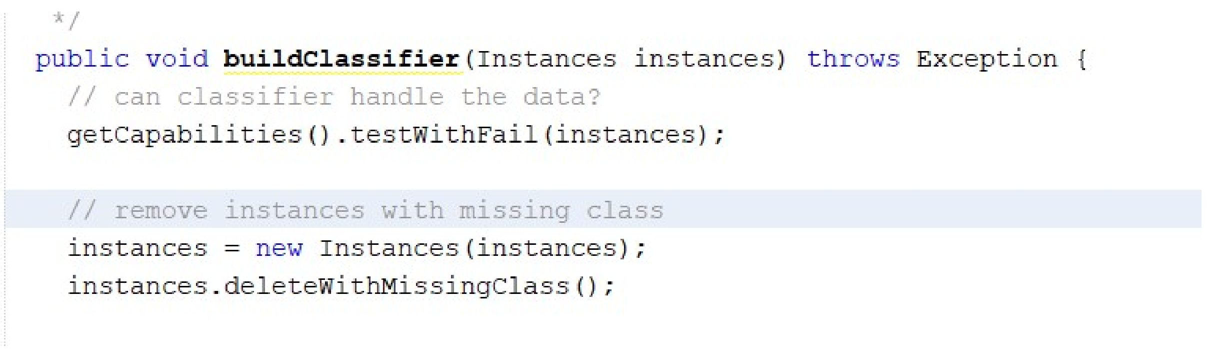
Now we are going to use ZeroR to help you understand the inner workings of built in classifiers.

b) Open the class and find the buildClassifier method. Look at the attributes and see if you can figure out what they store



```
public class ZeroR
  extends AbstractClassifier
  implements WeightedInstancesHandler, Sourcable {

  /** for serialization */
  static final long serialVersionUID = 48055541465867954L;

  /** The class value 0R predicts. */
  private double m_ClassValue;

  /** The number of instances in each class (null if class numeric). */
  private double [] m_Counts;

  /** The class attribute. */
  private Attribute m_Class;

  /**
```

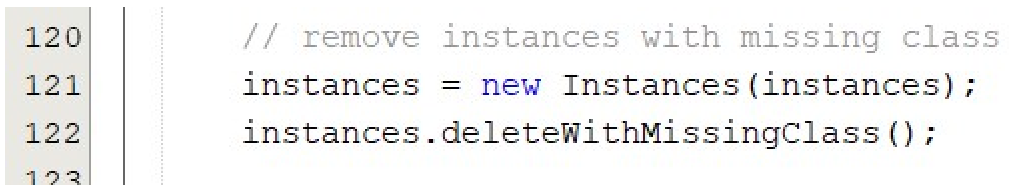
c) Read **`buildClassifier`**. Figure out what **`getCapabilities`** does by reading the method.



```
 */
public void buildClassifier(Instances instances) throws Exception {
  // can classifier handle the data?
  getCapabilities().testWithFail(instances);

  // remove instances with missing class
  instances = new Instances(instances);
  instances.deleteWithMissingClass();
```

The Capability mechanism is complex, but is meant to determine whether the classifier can handle different types of problem. What sort of attributes and class variable can ZeroG handle?

d) Read on in `buildClassifier`



```
    // remove instances with missing class
    instances = new Instances(instances);
    instances.deleteWithMissingClass();
```

Note that ZeroG clones the data. Many Weka classifiers do this, and it is often unnecessary.

```java
double sumOfWeights = 0;

m_Class = instances.classAttribute();
m_ClassValue = 0;
switch (instances.classAttribute().type()) {
  case Attribute.NUMERIC:
    m_Counts = null;
    break;
  case Attribute.NOMINAL:
    m_Counts = new double [instances.numClasses()];
    for (int i = 0; i < m_Counts.length; i++) {
      m_Counts[i] = 1;
    }
    sumOfWeights = instances.numClasses();
    break;
}
```

If the class value is numeric, we have a regression problem, not a classification problem. The classifier is designed to handle both. Refer back to the declaration of m_Counts at the top of the class. We are focussing only on the scenario where class attribute is NOMINAL. You can ignore other code. Read on

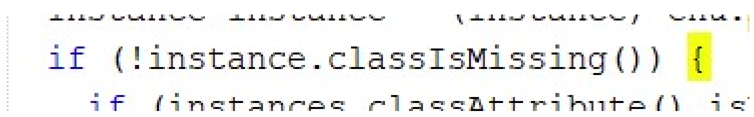
```java
double sumOfWeights = 0;

m_Class = instances.classAttribute();
m_ClassValue = 0;
switch (instances.classAttribute().type()) {
  case Attribute.NUMERIC:
    m_Counts = null;
    break;
  case Attribute.NOMINAL:
    m_Counts = new double [instances.numClasses()];
    for (int i = 0; i < m_Counts.length; i++) {
      m_Counts[i] = 1;
    }
    sumOfWeights = instances.numClasses();
    break;
}
```

Enumeration is an old form of Iterator. It works in a similar way, with **`hasMoreElements`**`()` instead of **`hasNext`**() and **`nextElement`**() instead of **`next`**(). Note this class was written prior to generics. There is a lot of fairly confusing code here. I imagine it is much more complex than your implementation. This is not necessarily a good thing!

Technical points to note about Weka and ZeroR:

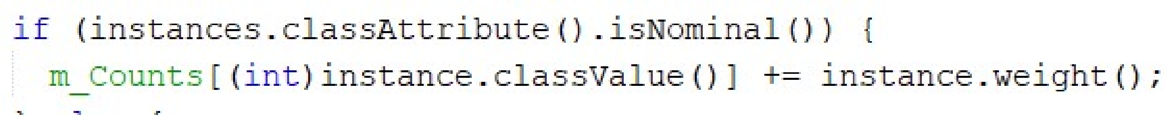
a) Instances can have weights. By default, they all have weight 1. We will come back to these weights with ensembles.

b) This code in ZeroR is unnecessary I think, because it has already removed all cases with missing classes. Cloning the data is wasteful of memory



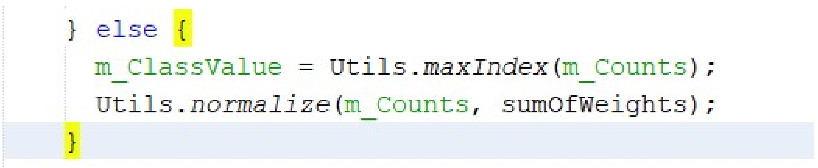
```
[illegible]
if (!instance.classIsMissing()) {
  if (instances.classAttribute().is
```

c) We are only interested in nominal classes, so this is the bit that counts the number of each class. This code basically counts how many cases there are of each class. Using the class value to index into an array is a common pattern. Did you use it for forming the confusion matrix? If not, go back and adapt your code.



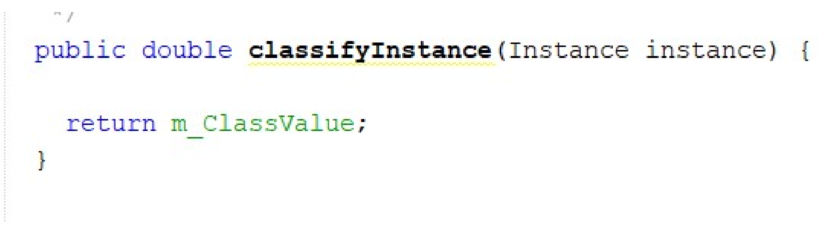
```
if (instances.classAttribute().isNominal()) {
  m_Counts[(int)instance.classValue()] += instance.weight();
```

d) This sets the majority class, and normalises the counts to be probabilities. Check the output is the same as yours if you have not already.

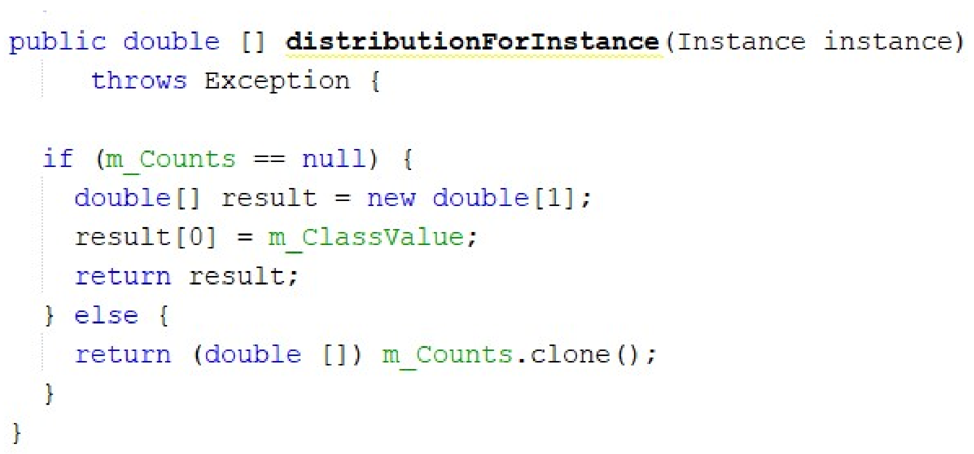


```
} else {
  m_ClassValue = Utils.maxIndex(m_Counts);
  Utils.normalize(m_Counts, sumOfWeights);
}
```

e) This is the prediction stage



```
public double classifyInstance(Instance instance) {

  return m_ClassValue;
}
```



```
public double [] distributionForInstance(Instance instance)
     throws Exception {

  if (m_Counts == null) {
    double[] result = new double[1];
    result[0] = m_ClassValue;
    return result;
  } else {
    return (double []) m_Counts.clone();
  }
}
```

Task 5: Analyse the Mosquito vs Fly Problem

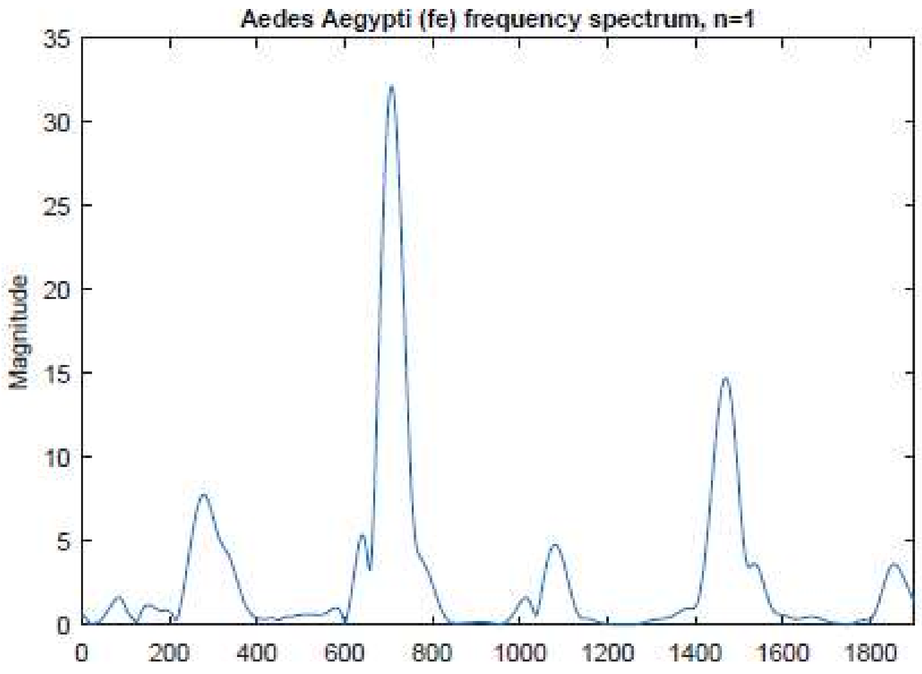
This week's real world application is from work Michael Flynn is doing for his PhD. The problem is to identify the type of insect based on the sound it makes passing through a sensor. The data was generated by researchers at the University of California

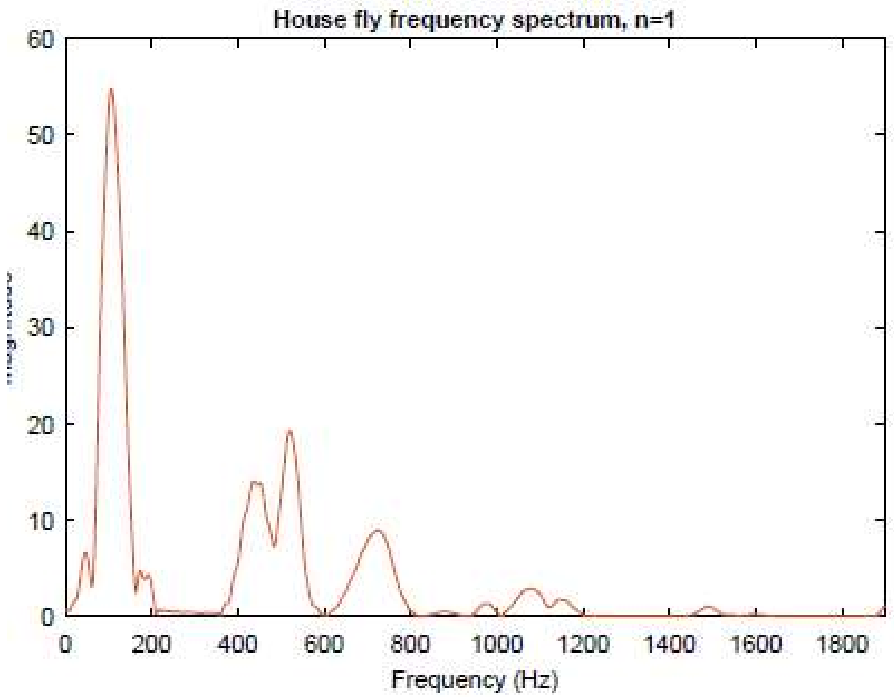
https://www.youtube.com/watch?v=7hj56v6gT34&hd=1

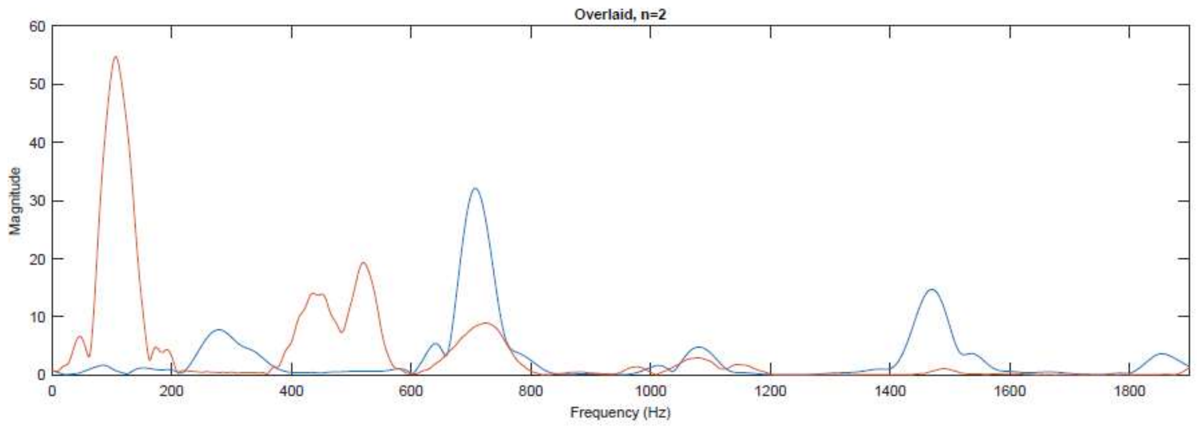
Michael has created an example of the type of problem. The data is in

Aedes_Female_VS_House_Fly_POWER

The problem is to identify whether the insect flying through the sensor is a mosquito or a fly.







I want you to explore using different classifiers for this problem, with the evaluation tools you have developed. Use the following classifiers: IB1, IBk, J48, Logistic and one of the ones you have implemented. If Logistic is too slow, just ditch it. Feel free to find out more about how these classifiers work, but more will be revealed as the module continues.

a) For a single random train test split (using 70% train and 30% test), build the classifiers, measure accuracy and construct a confusion matrix. Using this information, make an informal decision on which classifier is best.
b) Now measure the accuracy of the classifiers over 30 separate different random train test splits. Align the accuracies in a table and store it in a spreadsheet.

We will use these results in a later lab

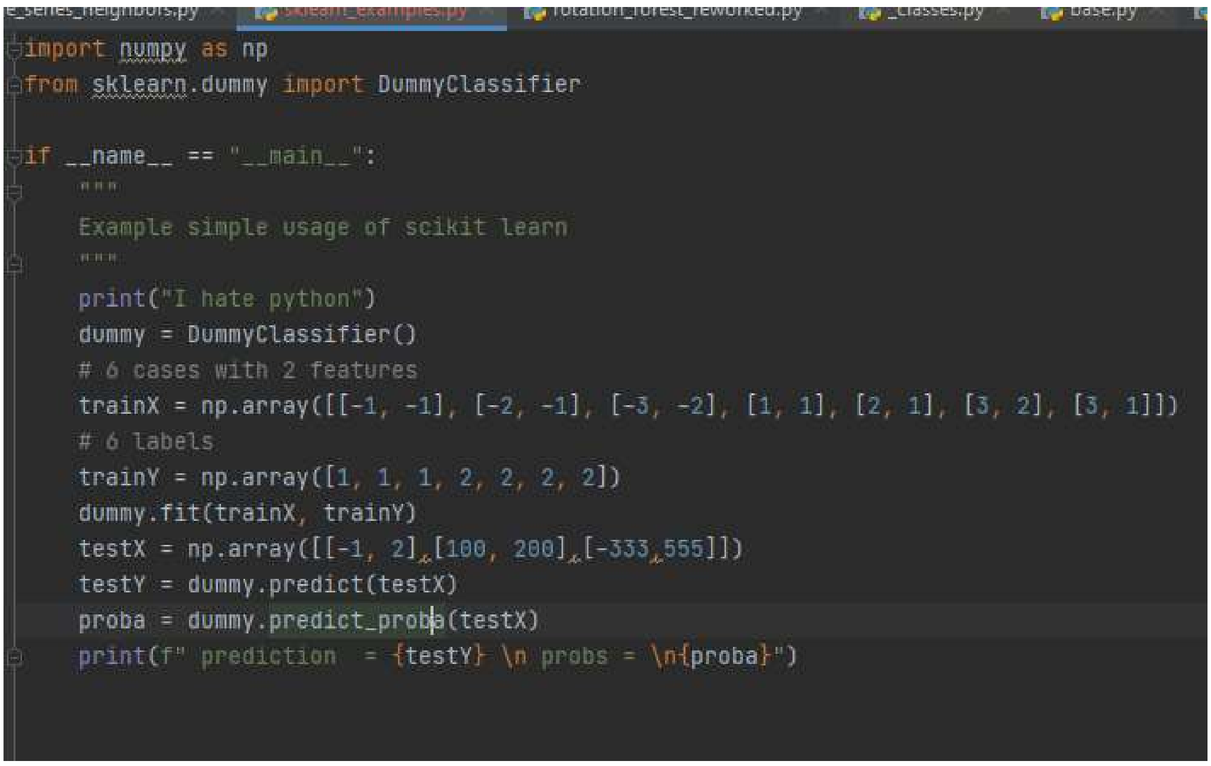
	Classifier1,	Classifier2,	Classifier3,	Classifier4
1	a11,	a12,	a13,	a14
2				
3				
.				
.				
.				
30	a30,1,		…	

Optional Topic

Machine learning in python with sci-kit learn and sktime

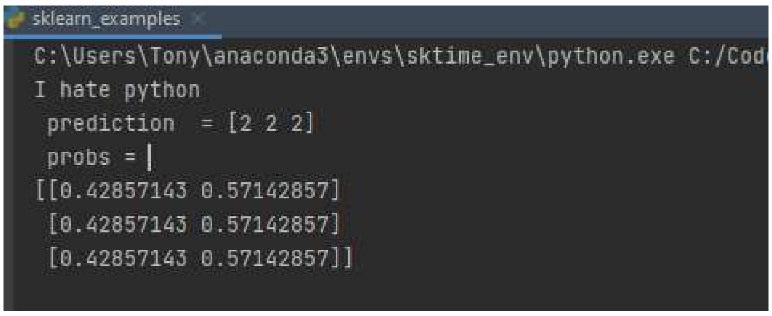
In a proper python style, we will use built in tools for scikit examples. I want you to understand topics via Java and Weka, but also be able to use the code available in toolkits. Weka also has such tools, you can scope them yourself.

DummyClassifier. Last lab we ended with the dummy classifier. DummyClassifier is like ZeroR. By default it predicts the majority class, and predict_proba give the class frequency.



```python
import numpy as np
from sklearn.dummy import DummyClassifier

if __name__ == "__main__":
    """
    Example simple usage of scikit learn
    """
    print("I hate python")
    dummy = DummyClassifier()
    # 6 cases with 2 features
    trainX = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2], [3, 1]])
    # 6 labels
    trainY = np.array([1, 1, 1, 2, 2, 2, 2])
    dummy.fit(trainX, trainY)
    testX = np.array([[-1, 2],[100, 200],[-333,555]])
    testY = dummy.predict(testX)
    proba = dummy.predict_proba(testX)
    print(f" prediction  = {testY} \n probs = \n{proba}")
```

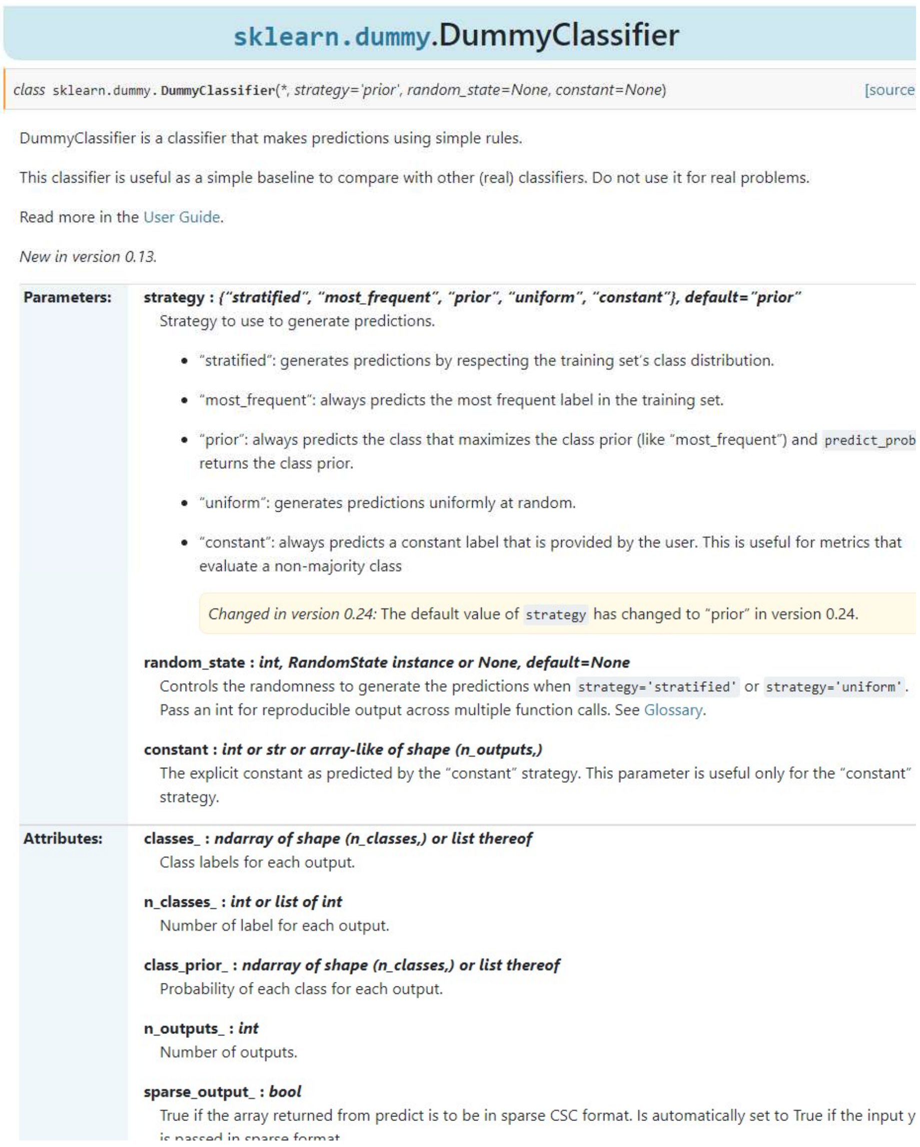
Outputs



```
C:\Users\Tony\anaconda3\envs\sktime_env\python.exe C:/Cod
I hate python
 prediction  = [2 2 2]
 probs = 
[[0.42857143 0.57142857]
 [0.42857143 0.57142857]
 [0.42857143 0.57142857]]
```

https://scikit-learn.org/stable/modules/generated/sklearn.dummy.DummyClassifier.html#sklearn.dummy.DummyClassifier

You can configure it through the constructor (this is the way with all sklearn algorithms)



sklearn.dummy.DummyClassifier

class `sklearn.dummy.DummyClassifier`(**, strategy='prior', random_state=None, constant=None*) [source

DummyClassifier is a classifier that makes predictions using simple rules.

This classifier is useful as a simple baseline to compare with other (real) classifiers. Do not use it for real problems.

Read more in the User Guide.

New in version 0.13.

Parameters:

strategy : *{"stratified", "most_frequent", "prior", "uniform", "constant"}, default="prior"*
Strategy to use to generate predictions.

- "stratified": generates predictions by respecting the training set's class distribution.
- "most_frequent": always predicts the most frequent label in the training set.
- "prior": always predicts the class that maximizes the class prior (like "most_frequent") and `predict_prob` returns the class prior.
- "uniform": generates predictions uniformly at random.
- "constant": always predicts a constant label that is provided by the user. This is useful for metrics that evaluate a non-majority class

Changed in version 0.24: The default value of `strategy` has changed to "prior" in version 0.24.

random_state : *int, RandomState instance or None, default=None*
Controls the randomness to generate the predictions when `strategy='stratified'` or `strategy='uniform'`. Pass an int for reproducible output across multiple function calls. See Glossary.

constant : *int or str or array-like of shape (n_outputs,)*
The explicit constant as predicted by the "constant" strategy. This parameter is useful only for the "constant" strategy.

Attributes:

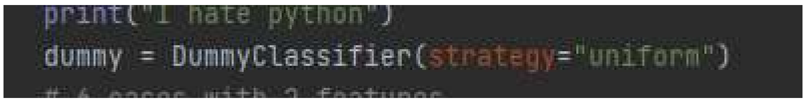
classes_ : *ndarray of shape (n_classes,) or list thereof*
Class labels for each output.

n_classes_ : *int or list of int*
Number of label for each output.

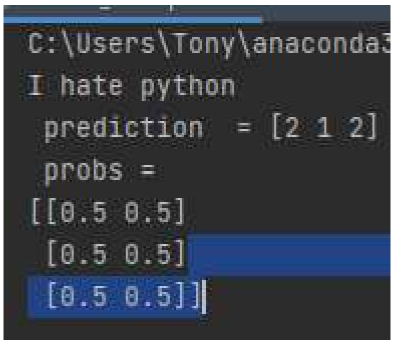
class_prior_ : *ndarray of shape (n_classes,) or list thereof*
Probability of each class for each output.

n_outputs_ : *int*
Number of outputs.

sparse_output_ : *bool*
True if the array returned from predict is to be in sparse CSC format. Is automatically set to True if the input y is passed in sparse format



```
print("I hate python")
dummy = DummyClassifier(strategy="uniform")
```



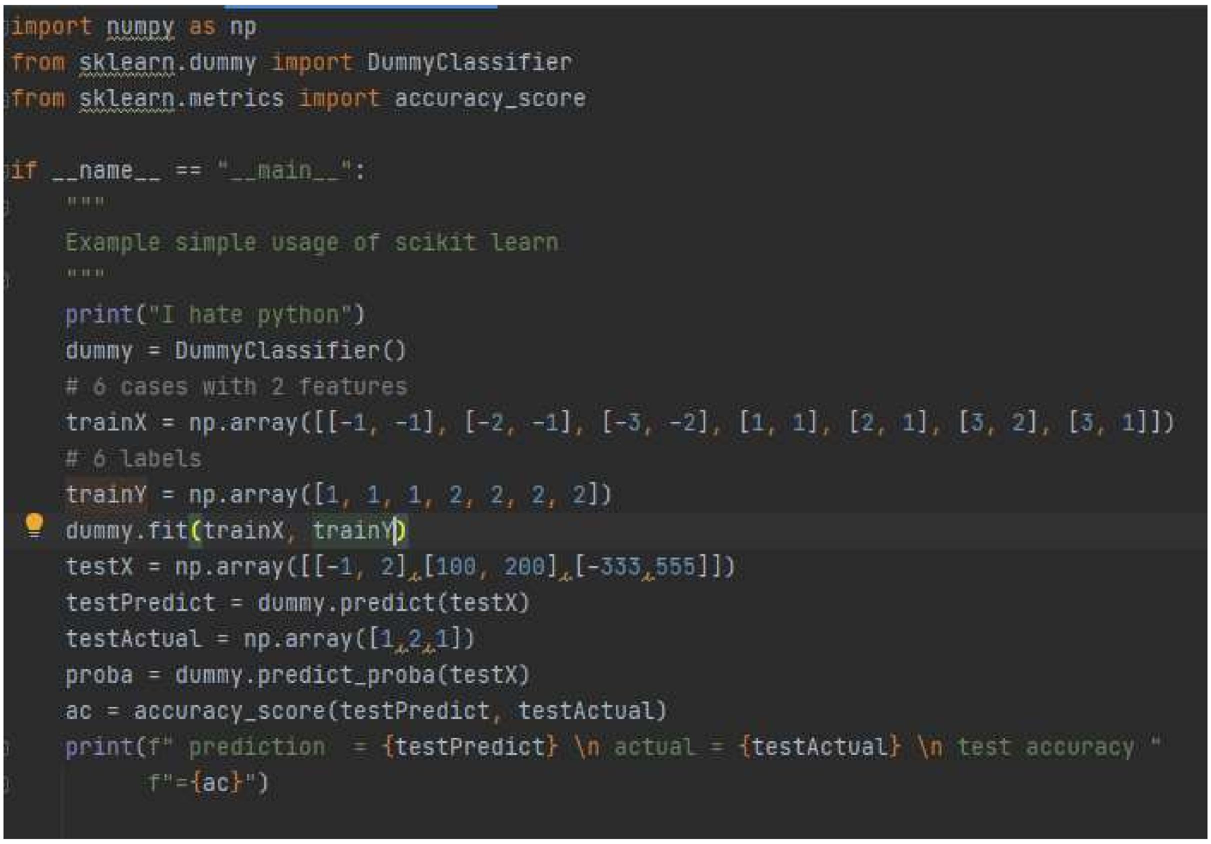
```
C:\Users\Tony\anaconda3
I hate python
 prediction  = [2 1 2]
 probs =
[[0.5 0.5]
 [0.5 0.5]
 [0.5 0.5]]
```

Classifier Metrics

The package sklearn.metrics has tools for performance estimation. For example

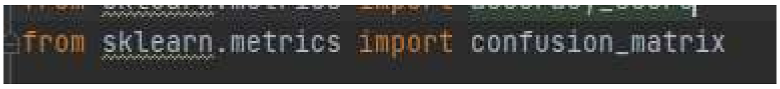
```
from sklearn.metrics import accuracy_score
```

will measure the accuracy for us. Code this example up:

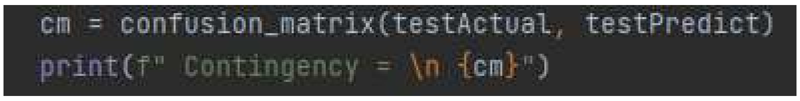


```
import numpy as np
from sklearn.dummy import DummyClassifier
from sklearn.metrics import accuracy_score

if __name__ == "__main__":
    """
    Example simple usage of scikit learn
    """
    print("I hate python")
    dummy = DummyClassifier()
    # 6 cases with 2 features
    trainX = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2], [3, 1]])
    # 6 labels
    trainY = np.array([1, 1, 1, 2, 2, 2, 2])
    dummy.fit(trainX, trainY)
    testX = np.array([[-1, 2],[100, 200],[-333,555]])
    testPredict = dummy.predict(testX)
    testActual = np.array([1,2,1])
    proba = dummy.predict_proba(testX)
    ac = accuracy_score(testPredict, testActual)
    print(f" prediction  = {testPredict} \n actual = {testActual} \n test accuracy "
          f"={ac}")
```

Now work out how to read the examples used for Weka into nparrays, and verify the predictions and accuracy are the same. Try and duplicate the weka exercises in sklearn



```
from sklearn.metrics import confusion_matrix
```



```
cm = confusion_matrix(testActual, testPredict)
print(f" Contingency = \n {cm}")
```