1st Assignment COMPX523-20A [v3]

Heitor Murilo Gomes, Jacob Montiel, Albert Bifet

February 2020

1 Overall Description

In this assignment you are asked to extend a k Nearest Neighbours (kNN) classifier implementation. In general, this assignment has two parts each of them graded as follows:

1. Coding (18% of the final score);
2. Experimentation and Analysis (12% of the final score).

Each of these parts will be further explained in the following sections. Important information:

- **Deadline: See Moodle**
- You must code your assignment using either MOA or scikit-multiflow.
- You don't need to implement the kNN algorithm from scratch, both MOA and scikit-multiflow already have kNN implementations that you can (and should) extend.
- Your submission must contain a zip file with 3 files:
    1. a file containing the classifier implementation (a single .java or .py);
    2. a pdf report with the experiments and analysis; and
    3. a README file with details about how to run your code and reproduce the experiments. Optionally, the README file can be replaced by a jupyter notebook with the execution of the experiments. Notice that a jupyter notebook doesn't replace the PDF report.

Throughout this assignment you will find **(Optional)** questions and requests. These do not give you extra credit nor they influence your final score. They are challenges that may go beyond what was taught in the course, or require further programming and analysis efforts. In summary, you are encouraged to try them, but they are not required.

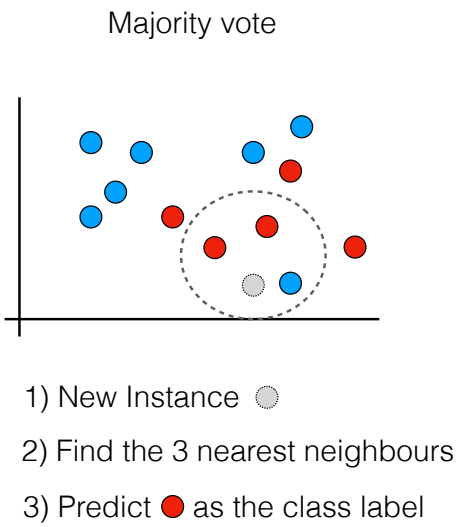

Figure 1: Majority vote using $k = 3$ for a binary classification 2 dimensional problem.

2 Coding

You can create a new classifier (i.e., implement a slidding window kNN from scratch) or you can adapt one of the existing kNN implementations. However, we **strongly recommend** that you extend the existing implementations. Notice that your kNN adaptation must be implemented in either MOA or scikit-multiflow in a way that it can be executed as a stand-alone classifier.

The kNN implementation must contain the implementations to satisfy the requirements described in the following subsections.

2.1 Weighted voting

The standard approach to determine the label of a previously unknown instance x_u on kNN is by employing majority vote among the k neighbours $neighbours(x_u, k)$ of x_u. This process is illustrate in Figure 1.

You are asked to implement a weighted vote based on the distance of each neighbour to x_u. Precisely, each $x_n \in neighbours(x_u, k)$ weights its vote in favor of y_n (i.e. the label associated with x_n) with $1/distance(x_n, x_u)$. Figure 2 extends the example presented on Figure 1.

2.1.1 MOA tips

In MOA, look for the kNN implementation available at **kNN.java**.

You will need to modify the method:

```
public double[] getVotesForInstance(Instance inst)
```

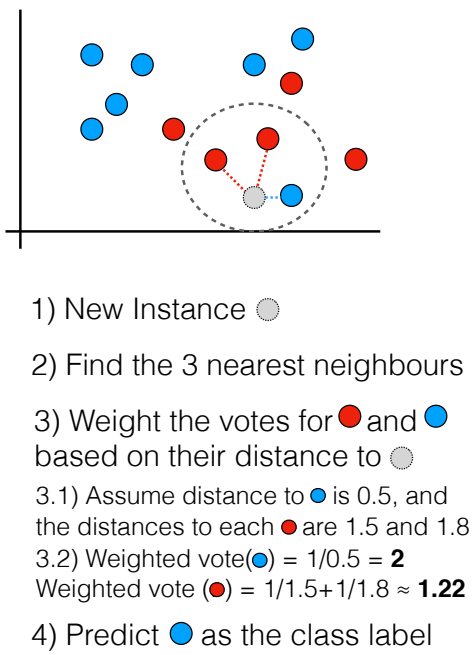

Figure 2: Weighted vote by $1 - distance$ using $k = 3$ for a binary classification 2 dimensional problem.

Add a new **Option**[1] to determine when to use weighted vote or not. This will help you set up and execute your experiments. For example, the parameter k is an option in kNN as shown below:

```
public IntOption kOption = new IntOption("k", 'k',
    "The number of neighbors", 10, 1, Integer.MAX_VALUE);
```

There are many different ways to create Options in MOA, the **IntOption** is just one of them. You may want to use a **FlagOption** or a **MultiChoiceOption**.

Finally, to get started extending MOA code you can follow the tutorial Building MOA from the source.

2.1.2 scikit-multiflow tips

For this assignment, we strongly recommend to install the development version of scikit-multiflow. There are two pre-requirements to have in your system: numpy and Cython. Once these pre-requiremnents are installed, you can install scikit-multiflow by running a single command in the terminal[2]:

```
pip install -U git+https://github.com/scikit-multiflow/scikit-multiflow
```

In scikit-multiflow, look for the kNN implementation available at **knn_classifier.py**. You will need to modify the method:

[1]The Option will show in the GUI and will be available through the command line.
[2]For more details see "Option 1. Install from source code" here.

```
    def predict_proba(self, X)
```

Add a new parameter to kNN to choose between majority vote and weighted vote. This will help you set up and execute your experiments. For example, a flag **weighted_vote** as shown below:

```
class KNNClassifier(BaseNeighbors, ClassifierMixin):
    # ...
    def __init__(self,
                 n_neighbors=5,
                 max_window_size=1000,
                 leaf_size=30,
                 metric='euclidean',
                 weighted_vote=False)
```

In order to easily test your implementation without the need to reinstall the framework, you can extend the existing implementation as follows:

```
class MyKNNClassifier(KNNClassifier):
    # ...
    def __init__(self,
                 n_neighbors=5,
                 max_window_size=1000,
                 leaf_size=30,
                 metric='euclidean',
                 weighted_vote=False)
        super().__init__(n_neighbors=n_neighbors,
                         max_window_size=max_window_size,
                         leaf_size=leaf_size,
                         metric=metric)
```

This way, MyKNNClassifier will inherit all functionality from KNNClassifier and you only need to focus on the methods that need to be overridden for the task at hand. Another benefit from this approach is that the file for your implementation does not need to be placed inside scikit-multiflow.

2.2 Incremental feature standardization

kNN is particularly sensitive to wide variations in the range of possible values each feature can assume. A standard kNN implementation calculates the distance between two data points using the Euclidean distance. A feature with a broad range of values will 'dominate' others when calculating distances. Therefore, the range of all features should be scaled so that each feature contributes proportionately when we sum them up.

You are asked to implement a method that will transform each feature x_i from the input data using **feature standardization**. To accomplish that you

will need the mean $\hat{x_i}$ and the standard deviation σ_i of each feature. The standardization is shown in Eq. 1.

$$x'_i = (x_i - \hat{x_i})/\sigma_i \tag{1}$$

Feature standardization makes the values of each feature in the data have zero mean and unity standard deviation. You must calculate the mean $\hat{x_i}$ and standard deviation σ_i incrementally taking into account every instance ever observed in the stream. Refer to the material from Week 1 and Week 2 precisely, the **StreamAlgorithmics-Slides.pdf**:

You can also find more information about how to calculate the mean and standard deviation for streaming data in Section 2.1.2 of [1], available in here.

2.2.1 MOA tips

Create a method in **kNN.java** to standardize the data as it arrives. You will need to apply it during training and testing, i.e. it will be executed in both methods below:

```
public void trainOnInstanceImpl(Instance inst)

public double[] getVotesForInstance(Instance inst)
```

Notice that even though you need to apply it twice, only during training the method should "update" any counters used for mean and standard deviation calculations.

Add a new **Option** to determine when to standardize the data or not. This will help you set up and execute your experiments.

2.2.2 scikit-multiflow tips

Create a method in **knn_classifier.py** to standardize the data as it arrives. You will need to apply it during training and testing, i.e. it will be executed in both methods below:

```
def partial_fit(self, X, y, classes=None, sample_weight=None)

def predict_proba(self, X) # Or predict(self, X)
```

Notice that even though you need to apply it twice, only during training the method should "update" any counters used for mean and standard deviation calculations.

Add a new parameter to kNN to select whether to use feature standardization or not. This will help you set up and execute your experiments. For example, a flag **standardize** as shown below:

```
class MyKNNClassifier(KNNClassifier):
    # ...
    def __init__(self,
                 n_neighbors=5,
                 max_window_size=1000,
                 leaf_size=30,
                 metric='euclidean',
                 standardize=False)
```

3 Evaluation and Analysis

The results should be presented using a table format, followed by a short analysis about them guided by the questions associated with each experiment. More details are presented in the subsections for each experiment. At the end of this section you will find more details about how to prepare, run and report your experiments using either scikit-multiflow or MOA.

Evaluation framework. For all experiments, use a test-then-train evaluation approach and report the final result obtained, i.e. the result obtained after testing using all instances.

Metrics. For all experiments, report **accuracy**, **recall (for each class)**, **kappa M**, **kappa T**, **time**[3].

3.1 Experiment 1: kNN variations and other algorithms

In the first part of the evaluation, you are asked to perform experiments using **3 learning algorithms**: kNN, Hoeffding Tree and Naive Bayes; and **1 baseline algorithm**: Majority Classifier[4] (always predict the majority class).

For kNN, also report results using the methods implemented in this assignment, i.e. weighted by 1/distance vote, and feature standardization. In summary, you will end up with 6 rows of results: kNN, hoeffding tree, naive bayes, majority classifier, kNN+weighted by 1/distance, kNN+standardization.

Dataset. The dataset used in this experiment will be "data_n30000". Refer to attached materials to obtain the ".arff" and ".csv" versions.

3.1.1 Guiding questions for the analysis of Experiment 1

These are questions that you must cover in your analysis, you can also discuss other interesting aspects that you observed. Remember that you are not required to answer the **(Optional)** questions.

- Have you notice anything in particular in terms of how the Hoeffding Tree, Naive Bayes and the Majority classifier perform?

[3] In MOA, time can be estimated by the **CPU Time**

[4] Not available in scikit-multiflow, in this case Naive Bayes will be the baseline.

- In other words, suppose you are more interested in correct predictions for the minority class (class 1), even if they come at the expense of more incorrect predictions for the majority class (class 0). Which metric would you focus on and which algorithm would you choose?
- What is a more appropriate general metric for this dataset: accuracy or Kappa M?
- Discuss the results obtained by using kNN with and without standardization. Did standardization improve the results? Base the justification for your answer on the characteristics of the data.
- **(Optional)** Based on the data, do you have any intuition about why some of the algorithms might fail? Tip: Try to visualize the data, there are only 2 dimensions.

3.2 Experiment 2: Hyperparameter tunning for kNN

Try to obtain better results with the kNN model by trying different combinations of hyperparameters, i.e. varying k, varying the width of the slidding window, using feature standardization, using weighted vote. For example, kNN with window length = 100 and k = 5; kNN with window length = 1000 and k=5; kNN with window length = 1000, k = 10 and weighted vote; ...

The goal is to improve upon the results you obtained using kNN in Experiment 1. Remember to report the results for all configurations that you have tried (one row for each). There are infinite possible combinations, but you are only asked to report 10 different configurations of kNN, notice that you are free to report more if you want.

Dataset. The dataset used in this experiment will be "data_n30000". Refer to attached materials to obtain the ".arff" and ".csv" versions.

3.2.1 Guiding questions for the analysis of Experiment 2

These are questions that you must cover in your analysis, you can also discuss other interesting aspects that you observed. Remember that you are not required to answer the **(Optional)** questions.

- What is the best configuration that you obtained? Rationalize why this configuration was better than the others based on the semantics of each hyperparameter and the values. Example: "The best configuration was kNN using k=X, window length=Y and weighted vote and no standardization, because when k k was ..."
- Explain the differences between each configuration, i.e. kind of a justification for why you have tried that configuration. Example: "I tried k = 1,5,10,20, so that I could observe the impact of a small k and a larger k. I observed that when k is small... "

- What is the impact in terms of execution time of the window length? Tip: show at least two experiments with a large different in their window lengths (e.g. a very small one and a large one) to support your answer.
- **(Optional)** Plot some of the instances that were incorrectly predicted by the standard configuration and that were correctly predicted by the best model. To achieve this, you will need to plot the instance and the neighborhood (and maybe distances to the instance) for each model, and then interpret how the standard and the best model behaved in each case. Explain how the best configuration was able to correctly predict those instances based on the configuration (for example: "the standard configuration uses k=10, while the best uses k=3, we can observe that only taking into account the 3 closest is often the right choice to predict the minority class...").

3.3 Experiment 3: Real data

In the last part of the evaluation, you are asked to perform experiments using **3 learning algorithms**: kNN, Hoeffding Tree, Naive Bayes; **1 baseline algorithm**: No-Change Classifier[5] (always predict the previously seen class label); and **2 ensemble algorithms**. You can choose from any of the available ensemble algorithms.

For kNN, also report results using the methods implemented in the other parts of this assignment, i.e. weighted by 1/distance vote, and feature standardization. In summary, you will end up with 8 rows of results: kNN, hoeffding tree, naive bayes, no-change classifier, kNN+weighted by 1/distance, kNN+standardization, ensemble1 and ensemble2.

(Optional) Hyperparameter tunning for the ensemble methods. Choose one (or both) of the ensemble methods to tune, select the hyperparameters to tune (e.g., ensemble size) and report their results in a separate table.

Dataset. The dataset used in this experiment will be "covtype_numeric". Refer to attached materials to obtain the ".arff" and ".csv" versions.

3.3.1 Guiding questions for the analysis of Experiment 3

These are questions that you must cover in your analysis, you can also discuss other interesting aspects that you observed. Remember that you are not required to answer the **(Optional)** questions.

- Which model obtained the best results in terms of accuracy? Does accuracy alone is a good indication that this model is a reasonable solution to this problem? What other metric should be take into account?
- How does the models compare against the no-change classifier in terms of predictive performance? Is it a concern that a model is outperformed by the no-change classifier?

[5]Not available in scikit-multiflow, in this case Naive Bayes will be the baseline.

Algorithm	Accuracy	Recall 0	Recall 1	Kappa M	Kappa T	Time (s)
knn(k10,w1000,maj)	68.02	82.65	38.75	4.05	30.23	3.554754
knn(k5,w1000,maj)	68.31	78.465	**47.99**	4.92	30.86	3.428505
knn(k10,w1000,weighted)	**68.62**	78.94	**47.99**	**5.87**	31.55	3.885672
...	...	...	...	...	...	...
HT	66.74	96.045	8.14	0.23	27.45	0.140724
NB	66.656	**99.98**	0.01	-0.03	27.26	0.07592

Table 1: Results for dataset "XYZ" using Interleaved Test-Then-Train evaluation. Best results are highlighted in bold.

- How did kNN and its variants performed? Are they a reasonable solution to this problem? Take into account their performance against other models in terms of predictive performance and processing time.
- **(Optional)** How would you incorporate a weighted vote into kNN to exploit temporal dependencies in the data? Try to implement this alternative weighted vote and observe the results. Tip: Check [2] for more details about temporal dependencies.

3.4 Preparing your report

Present your results. Create a table to report the results of each experiment, such as Table 3.4.

Organize your experiment. Add meaningful identifiers to each model being tested, so that you can easily refer to them and understand what they are without constantly looking at a list of acronyms. For example, notice how the various executions of knn on Table 3.4 can be interpreted (k = number of neighbors, w = width of the slidding window, maj = majority vote / weighted = weighted vote by 1/distance). Even though it is expected that your identifiers are explicit, also create a list for the acronyms you are using just to make sure there are no acronyms undefined (for example, HT = Hoeffding Tree). You don't have to use exactly the same acronyms as in Table 3.4, but pick something that can be easily interpreted.

Prepare your PDF report. You can use whatever you want as long as you the report is submitted as a PDF file. However, one suggestion is that you use either markdown or LaTeXto prepare your report. For LaTeX, overleaf is a good option.

3.4.1 MOA tips

Use EvaluateTestThenTrain, make sure you have the Evaluator set as: **"BasicClassificationPerformanceEvaluator -o -r"**, otherwise you won't obtain the recall for each class on your output. A sample command that you can copy and paste in MOA GUI can be found below.

```
EvaluatePrequential -l trees.HoeffdingTree
-s (ArffFileStream -f data_n30000.arff)
-e (BasicClassificationPerformanceEvaluator -o -r)
-i 30000 -f 100
```

Notice that **"-f 100"** does not influence the end result, it is just how often the intermediary results are printed. You can also set **"-f 30000"** so that you only get one row of results (the last one). If you are running it on the command line, you might use something like this:

```
java -Xmx4g -Xss10M -cp moa.jar moa.DoTask
"EvaluatePrequential -l trees.HoeffdingTree
-s (ArffFileStream -f data_n30000.arff)
-e (BasicClassificationPerformanceEvaluator -o -r)
-i 30000 -f 100"
```

3.4.2 scikit-multiflow tips

The following code snippet shows how to setup the evaluation with the existing KNNClassifier

```
from skmultiflow.data import FileStream
from skmultiflow.lazy import KNNClassifier
from skmultiflow.evaluation import EvaluatePrequential

stream = FileStream("path/to/data_n30000.csv")

knn = KNNClassifier()

metrics = ['accuracy', 'kappa', 'kappa_m',
           'kappa_t', 'running_time', 'model_size']
evaluator = EvaluatePrequential(max_samples=30000,
                                n_wait=100,
                                show_plot=False,
                                metrics=metrics)
evaluator.evaluate(stream=stream,
                   model=[knn],
                   model_names=['KNN'])
```

Per-class recall is not currently available in scikit-multiflow. However it is easy to calculate it. You just need to add the following **immediately** after the "evaluate" call:

```
cm = evaluator.get_mean_measurements(0).confusion_matrix

print("Recall per class")
for i in range(cm.n_classes):
```

```
recall = cm.data[(i,i)]/cm.sum_col[i] \
if cm.sum_col[i] != 0 else 'Ill-defined'
print("Class {}: {}".format(i, recall))
```

References

[1] H. M. Gomes, J. Read, A. Bifet, J. P. Barddal, and J. Gama. Machine learning for streaming data: state of the art, challenges, and opportunities. *ACM SIGKDD Explorations Newsletter*, 21(2):6–22, 2019.

[2] I. Žliobaitė, A. Bifet, J. Read, B. Pfahringer, and G. Holmes. Evaluation methods and decision theory for classification of streaming data with temporal dependence. *Machine Learning*, 98(3):455–482, 2015.