KeLP - Question Classification

Alessio Moretti - alessio.moretti@live.it (0239045)

Goals

Using a previous implementation of the Question Classification task using the KeLP library for kernel-based learning, we should:

- parametrize the kernel function applied to the classifier to adopt a N-fold cross validation approach;
- learn the classifier maximizing the F1 metrics for the `HUM` (human) class;
- re implement explicitly the computation of TP, FP etc. without using the built-in evaluators.

As we will se, in this implementation of mine I have tried to cover all this subtasks in a unique solution.

The application structure

The application was originally intended to be provided via command line the kernel type, the weight for the slack coefficients for the soft-margin SVMs, the input path for the training set, the input path for the validation set. In this implementation, we will use as command line parameters the only kernel type and the soft margin weight. The application is now bundled (using Java Resources helpers) with a training and a validation set from the TREC challenges. The library version used is `KeLP 2.2.0` (the project is available with its documentation at kelp-ml.org) and the project is powered by Maven to be extensible as a KeLP "playground" to implement different optimization tecniques.

The application is composed of two different classes: one is dedicated to implement all the methods to achieve the goals of the exercise and it is the `main` from which the computation is achieved and the execution flow is instantiated, one is the `WmIRClassifier` which is simply an object into which the metrics, the classifier and the learner are stored for further utilization (e.g. for example the comparison between different classifiers).

Algorithm

Let we briefly describe the algorithm that has been implemented to achieve the goals of this track. As we introduced, we will let the user choose amongst different kernel types, a custom base weight for the soft margin SVM and let the algorithm run a grid search optimization over different weights and over a n-fold cross validation schema. The best classifiers for each weight are then validated using a new validation dataset. We will cover all of this in details in the following sections.

Kernel

Very little changes have been made to the original kernels functions by default used in the first version of this application. We will report them briefly now (every of them is built-in in the KeLP support library), as different extensions of the base `Kernel` class:

- `LinearKernel`
- `PolynomialKernel`

Whose represents the similarity between vectors in a feature space which is linear (or polynomial) combination of the input variables. By default in our case the exponent for the polynomial combination is 2.

- `SubSetTreeKernel`

Which is frequently used in Natural Language Processing, where it is often necessary to compare tree structures. Well-designed kernels allow to compute similarity over trees without explicitly computing the features vectors of these trees.

- `LinearKernelCombination`
- `LinearKernelCombination - NormalizationKernel`

When we have linear combination of linear and subtree kernels as well as linear and normalization kernels, for more complex classifiers implementations.

Kernel cache

The kernel-based computation are often resources-intensive, so KeLP provides a cache to speed up computations. This cache is shared amongst the different trained classifiers.

SVM

The very first step of the algorithm, is to choose a solver that will implement the bae kernel function. In this case the application relies (according project's requirements) on the `BinaryCSvmClassification`, where the final decision function is binary: the classifier will categorize the example as positive or negative according the sign of the decision function.

$$h(x) = sign(f(x)) \tag{1}$$

$$f : X \subseteq I\!R^n \rightarrow I\!R^n \tag{2}$$

To achieve that, we will use the Support Vector Machine approach to keep trace of the most relevant labelled examples from the training set In particular, in this implementation we will use a soft-margin SVM approach where classification

errors are allowed, according to a normalization weight C.Greater the weight, the harder the margin: on the other side if the weight is zero, every error is accepted. We can optimize the SVM adjusting the C weight. In formulas, we have to work with the following dual linear programming problem:

$$\min \quad \frac{1}{2}\|\vec{w}\|^2 + C\sum_k \zeta_k \tag{3}$$

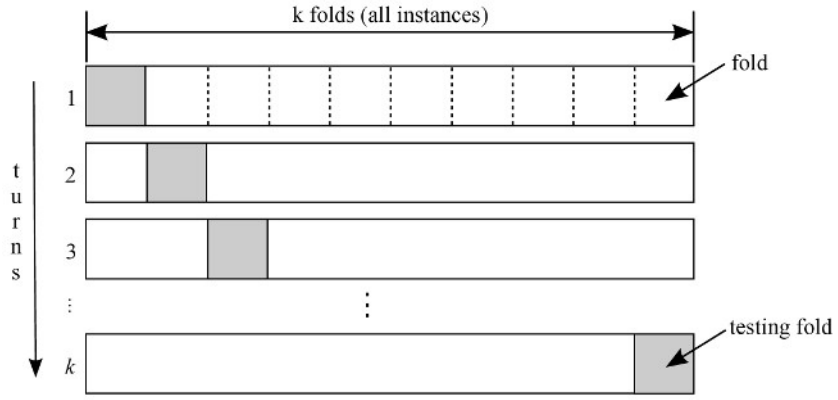
$$\text{subject to} \quad \forall k : y_k[\vec{w} * \vec{w_k} + b] >= 1 - \zeta_k \tag{4}$$

where a sequence of labelled examples (x_k, y_k) is provided.

One vs All

In this Question Classification task we have used the module `OneVsAllLearning` to use the One-vs-All learning schema, where we have a competition between a number of different classifiers: each one is related to a particular class on which it is binary trained and combined to the other classifiers to let the overall classifier identify different classes.

K-fold cross validation



Here comes one of the first tasks to extend the original application. We had to use a K-fold cross validation schema (with a defined $K = 5$) to train the learner over one of the two given datasets (the training set in this case).

1. the training set is shuffled and so randomly partitioned in k different folds;
2. for each step we use one of the folds as a test set and the rest as training;
3. we run training and validation step averaging the metrics.

The K-fold cross validation ensure that every given example will be once in the validation set and a weighted $(\frac{1}{k})$ average of accuracy and other metrics is produced. This technique is particularly useful to avoid overfitting.

Implementation

Let wed dig into how the algorithm has been implemented using the pseudocode to describe it:

```
// splitting in k folds
foldArray = trainingSet.nFolding(k);

// repeating k times
foreach fold in foldArray:
        validation = fold;
        training   = foldArray.remove(fold);
        // learning on the other folds
        learner.learn(training);
        // evaluating on the selected fold
        metrics += learner.evalute(fold);

// averaging the metrics
metrics = 1/k * metrics;
```

Evaluation on `HUM` class

During the evaluation step, we need to evaluate for each classification label, provided via the examples themselves, the following:

- True Positive: the classifier correctly recognize the question label;
- True Negative: the classifier correctly classify the question as non label-specific;
- False Positive: the classifier classify the question as referred to a label when it is not;
- False Negative: the classifier did not classify the question for the given class even when it is right.

Executing the k steps, debugging step by step, it is possible to observe how the learner change its metrics from a fold to another, according the elements present in each validation set, producing an average satisfying classification.
In this case, to be more specific, we are using the following algorithm to calculate the metrics:

```
foreach example in examples:
  foreach label in classificationLabels:
        if example.isExampleOf(label)
                if (predicted == label) TP[label]++;
                else FN[label]++;
```

```
else
        if (predicted == label) FP[label]++;
        else TN[label]++;
```

At the end we have for each label the number of TP, TN, FP, FN whose are required to compute the label specific metrics and the averaged one. In particular the application is now able to compute for each label the specific metric (precision, recall and F1), while it can use later the overall count for the overall metrics. In both the two cases we store in the classifier object the results relative to each label and the classifier averaged precision, recall and F1. The equations adopted are the following:

$$\text{precision} = \frac{TP}{TP + FP} \tag{5}$$

$$\text{recall} = \frac{TP}{TP + FN} \tag{6}$$

$$\text{F1} = 2 * \frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}} \tag{7}$$

We can observe that in case of multiclass classification (as it is in our case due to the use the One-vs-All schema), we have that in case of non-correct classification both of the false positive and false negative values are incremented in the overall results. So in the overall metrics we will have in output:

$$\text{precision} = \text{recall} \rightarrow \text{F1} = 2 * \frac{((precision))^2}{2 * \text{precision}} = \text{precision} = \text{recall} \tag{8}$$

Different observations can be made to each of the labels local metrics that can be produced in output selecting the label to optimize (as we will se in the next section)[1]. To conclude this section we introduce the notion of accuracy, which in this particular implementation is computed as the micro-average of all the accuracies using the overall observations of TP, TN, FP, FN, which are related in the following formula:

$$\text{accuracy} = \frac{TP + TN}{TP + FP + TN + FN} \tag{9}$$

In particular, the true positive and true negative values will be equal due to the simple fact that both are the sum of all true classified examples, regardless their classes.

Slack weights optimization

One last step for the application to satisfy all the requirements was to apply a tuning step to enhance the performances of the learners. I decided to work on two different levels:

[1] every result has been validate using the built-in KeLP evaluators.

1. running the application for all the different kernel types provided from the original application;
2. in each run let the algorithm cycling over a vector of weights for the slack weight to find the optimum one.

While the first step was easy (and capable of automation too), the second one required to use the command line provided base weight and to elevate it to powers in the range $(-2, 2)$, the range was decided after a validation and tuning step. Each weight is the C parameter in the soft margin SVM binary learner used in the One-vs-All classifier. So we obtain this three-step validation:

1. training for each of the chosen C slack variable;
2. evaluation on the training set (metrics produced with the previous method, according the `HUM` class);
3. the one that maximize the F1 measure across the weights is chosen as the best;

As we can see, it is a grid search, where to obtain a certain degree of automation we should provide an API to automate across the different kernel types.

Results

Output

A typical output for this application is the following:

```
Kernel Type: lin
C_SVM -> 2.0
```

OVERALL METRICS	
F1	0.868
precision	0.868
accuracy	0.956
recall	0.868

CLASS 'NUM'	
F1	0.8888889
precision	0.9787234
recall	0.8141593

```
[SYSTEM] time elapsed: 311 seconds
```

KERNEL TYPE	C (SVM)	F1 MEASURE	PREC = REC	ACCURACY
Linear	1	86,4%	86,4%	95,46%
Polynomial	0,25	86,2%	86,2%	95,4%
SubTreeKernel	0,25	84,8%	84,8%	94,9%
Lin. Combination	0,25	88,4%	88,4%	96,1%
Norm. Combination	2.00	80,2%	80,2%	93,4%

Figure 1: comparison table for the classifiers selected by the optimum search algorithm - according the F1 measure over 'HUM' class

KERNEL TYPE	C (SVM)	F1 (HUM)	PREC (HUM)	REC (HUM)
Linear	1	88,9%	100%	80%
Polynomial	0,25	88,1%	98,1%	80%
SubTreeKernel	0,25	88,9%	100%	80%
Lin. Combination	0,25	88,9%	100%	80%
Norm. Combination	2.00	88,9%	100%	80%

Figure 2: results obtained using the validation set bundled with the application, they are pretty similar, we will focus on the previous table for a better comparison

Comparison

All the tests were run using as default base for $C = 2$ and running atop of a MacBook Pro 2015 - i5 using Java 1.8 and KeLP 2.2.0, with the development environment provided by JetBrains IntelliJIdea integrated development environment.
We can now compare the performances of each classifier found by the optimization procedure (given the `HUM` class and its relative F1 measure) in the following table: we will report in the following table only the kernel type, the weight for the SVM, the F1 measure (equal to precision and recall as derived in the previous equations) and the accuracy.

The main goal was to provide one of the best classifiers according the F1 measure. It is clear that the best choice is the `Linear Combination` of the linear and subtree kernels which have the best results amongst the others. It is interesting that a very soft margin is used, mitigating with the subtree kernels features (used in the literature for Natural Language Processing) the linear kernel harder margin required for an interesting classifier. The worst classifier is the `Normalization Combination` which uses the hardest margin to achieve the lowest performances.

Further steps

In this early stage it is possible to find out (for a future in production implementation) some further improvements.

Modularity

We have seen that there are two main classes managing this exercise. It can be considered to increase the number of classes to manage different combinations of grid searches and data structure to handle (with a minor effort on the memory) the K-fold cross validation. Moreover we can provide a subset of classes to handle different scenarios and provide any automation API to the developer itself.

Performances

One of the main issues that arose from this task was to provide an efficient algorithm. While for this particular implementation we relies upon the KeLP support libraries and the Java Virtual Machine, we can think of multithreading to enhance the cross validation schema performances.