FTML practical session 11

5 juin 2025

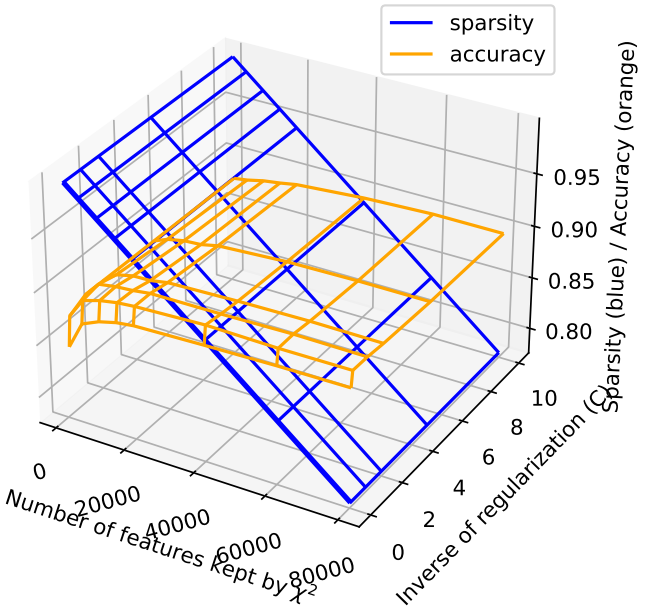

TABLE DES MATIÈRES

1 FEATURE SELECTION

Introduction

In this session we will work with the Stanford sentiment analysis dataset.

https://ai.stanford.edu/~amaas/data/sentiment/

The dataset contains 25000 train samples and 25000 test samples. Each input is a review about a movie. The task is a binary classification : predict whether a review about a movie is positive or negative.

Our objective is to find linear estimators that are sparse but still keep a good prediction performance. Sparsity is nice because the prediction might be faster than a dense estimator at runtime, and also because a sparse estimator is easier to interpret.

Note that for this problem, we intuitively expect that it should be possible to have a good predictin with a sparse estimator. Indeed, the presence of some specific words in the review, like "bad", "good", or "awful" should be very informative about its polarity.

1.1 Dataset loading and preprocessing

The file **fetch_dataset_Stanford_sentiment.py** loads the dataset and puts it in a cache for later use. In the following exercises, the dataset is preprocessed before applying learning algorithms. The text files are cleaned by the function **clean_text()** in **utils_data_processing.py**. To see the result of the cleaning, run **exercice_0_test_preprocessing.py** and read **clean_text()**.

1.2 Vanilla logistic regression

We will start by trying a classical baseline, with a logistic regression on a one-hot encoding of the texts.

Build a baseline estimator, with a pipeline that will contain the following steps :

— a one-hot encoding of the data.

— a scaling of the one-hot representation (optional)

— a vanilla logistic regression. ("vanilla" is a term used to mean "standard", "basic", "default").

You can use the tools and docs listed in 1.2.2.

Explore manually different hyperparameter values (see 1.2.1 just below), and evaluate whether the scaling as an impact on performance or not. Save the vocabulary built by the one-hot encoding stage to a txt file that contains all the n-grams (each line contains a n-gram in the vocabulary).

It is possible to reach a test accuracy of 0.9.

File : **exercice_1_logistic_regression.py**

1.2.1 *Hyperparameters*

The one-hot encoding step as some important parameters. The first one is the size of the **n-grams** used. A n-gram is a sequence of n words, for instance "is good" is a 2 gram. When doing a one-hot encoding, the size of the one-hot representation will directly depend on the range of integers n for which we keep the n-grams. Parameter : **ngram_range** in CountVectorizer.

Another important parameter is the minimum document frequency that is necessary to keep a word in the one-hot representation. Parameter : **min_df** in CountVectorizer.

1.2.2 Ressources

https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html
https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html
https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MaxAbsScaler.html

1.3 Univariate filtering

Univariate filtering consists in statistically evaluating whether each feature carries information about the target, and then keeping some number of most informative features. To do so, several statistical tests are possible. In the case of text documents, the χ_2 test is a relevant option because the features are non-negative, as they represents counts or frequencices.

Implement an univariate filtering in order to remove some features from the data and monitor the change in test accuracy after doing a logistic regression, as in the previous exercise. Observe the compromise between sparsity and quality of prediction, as a function of the number of kept features, and save the filtered vocabularies to different files.

To do so, you will need to evaluate the sparsity of an estimator, an for this, a sparsity scorer might be used. A possibility is to measure the degree of sparsity by a number in $[0, 1]$ (1 meaning fully sparse, and 0 full dense). Here, we can define the sparsity as $1 - f$, where f is the fraction of kept features.

Now, the pipeline contains a filtering step between the one-hot encoding and the scaling. The **SelectKBest** has a **get_support()** method that retrieves the features that are kept by the statistical test (this is useful to save the filtered vocabulary).

We can visualize the compromise between accuracy and sparsity, in different manners.

— In figure 1, the compromise between the fraction of kept feature and the accuracies is shown.

— In figure 2, the regularization constant C of logistic regression is added to the plot.

File : **exercice_2_univariate_filtering.py**

It is possible to have a test score of around 89% with a sparsity score of around 83%.

1.3.1 Hyperparameters

For the statistical test, some hyperparameters of CountVectorizer are especially important. For instance, let us consider the **binary** argument of the class.

— if **binary=True**, then each feature of the vector that represents a document is the presence or absence of the n-gram in this document, hence it is set to 0 or 1.

— if **binary=False**, then each feature represents the frequency or counter of the n-gram in the document.

These two options are both valid in our case!

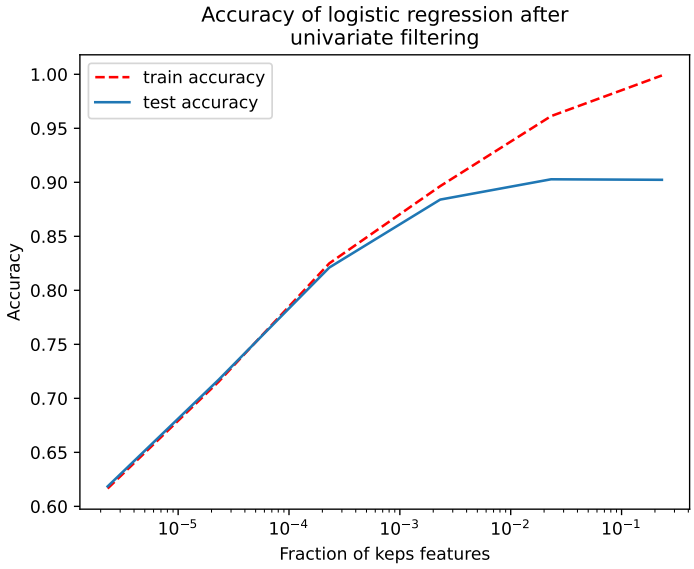

Figure 1 – Univariate filtering : compromise between accuracies and fraction of kept features after applying SelectKBest and logistic regression.

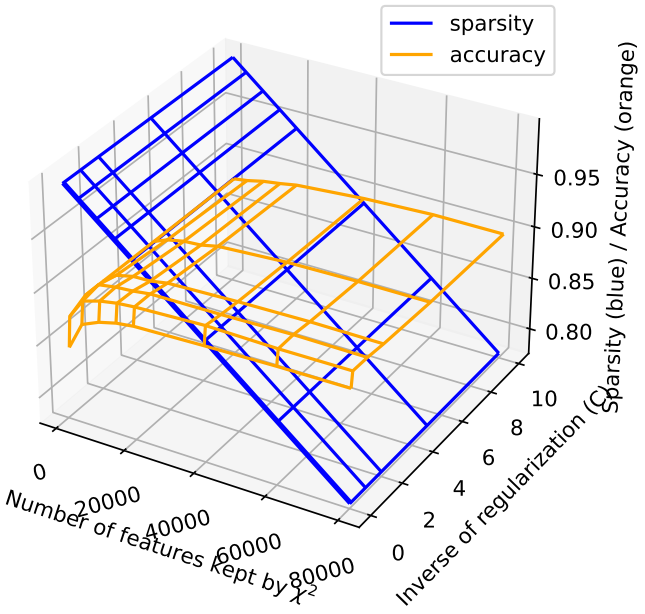

Figure 2 – Compromise between sparsity ($1 - f$, where f is the fraction of kept features) and accuracy. Both sparsity and accuracy are in $[0, 1]$, so they can be represented on the same axis in this image.

1.3.2 *Ressources*

https://scikit-learn.org/stable/modules/feature_selection.html#univariate-feature-selection
https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectKBest.html#sklearn.feature_selection.SelectKBest
https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.chi2.html#sklearn.feature_selection.chi2
https://fr.wikipedia.org/wiki/Test_du_%CF%87%C2%B2

1.4 Embedded algorithm

We will now study whether the sparsity can be enforced in the estimator itsself, similarly to a sparse Lasso estimator (linear regression with L1 regularization).

Perform a new pipieline, similar to the first one (without filtering, section 1.2) but with a L1 regularization in the logistic regression step.

Save the vocabulary effectively used by the obtained estimator, along with the weights corresponding to each word. Rank the used words according to the weight, and interpret the meaning of this ranking. Evaluate the influence of the regularization parameter C (which corresponds to the inverse of the regularization strength) on the sparsity and on the accuracy (e.g. by plotting the cross validated accuracy, like in figure 3).

File : **exercice_3_l1_regularization.py**

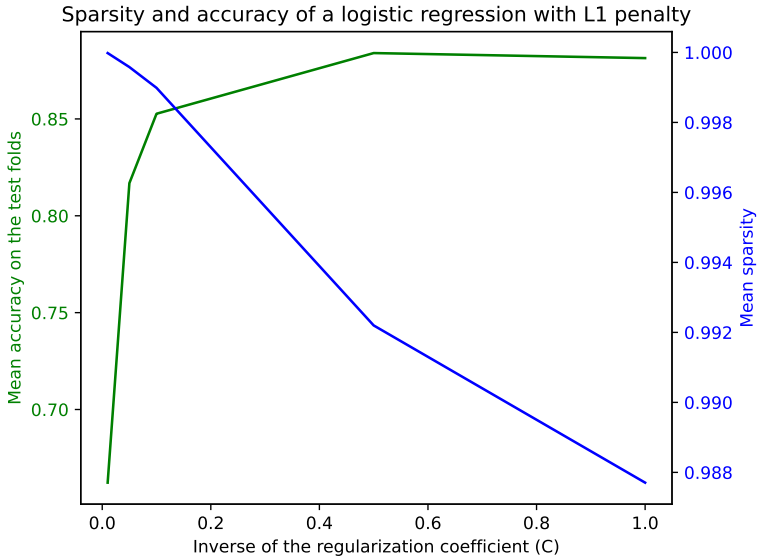

FIGURE 3 – Sparsity and accuracy as a function of the regularization strength.

1.5 Wrappers

As a last method, we will now use a wrapper, in order to reduce the number of features used by the estimators. When using a wrapper, we score feature subsets, based on a previously trained estimator (as opposed to univariate filtering, where we remove features of the inputs without taking an estimator into account). We will use the recursive feature elimination (RFE) method, which is one possibility out of several other available choices. Features are removed iteratively : at each iteration, a chosen number of features is removed (this is a parameter of the algorithm), and the

estimator is trained again. This iteration is performed until the number of features kept is equal to a specified number (which is another parameter of the RFE).

By default, in the case of a linear estimator, the score for each feature used by the scikit implementation of RFE is the squared value of the coefficient that corresponds to this feature. In **utils_data_processing.py**, there is a function **LinearPipeline** that makes it possible to apply RFE to a pipeline directly, with this same feature scoring.

Apply RFE with a chosen number of final features and a step size, and save the final vocabulary used to a file, again sorted according to the weights.

With this method on this example, it is possible to keep the 0.9 test accuracy with a vocabulary of 10000 words.
File : **exercice_5_wrapper.py**

1.5.1 Ressources

`https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFE.html#sklearn.feature_selection.RFE`

1.6 Conclusion

We have seen several methods in order to obtain a sparse estimator for this binary classification task. These methods lead to slightly different vocabularies being used, but all work well as they keep an accuracry that is as high or almost as high as the vanilla logistic regression that uses all the vocabulary.