FTML practical session 13: 2023/09/23

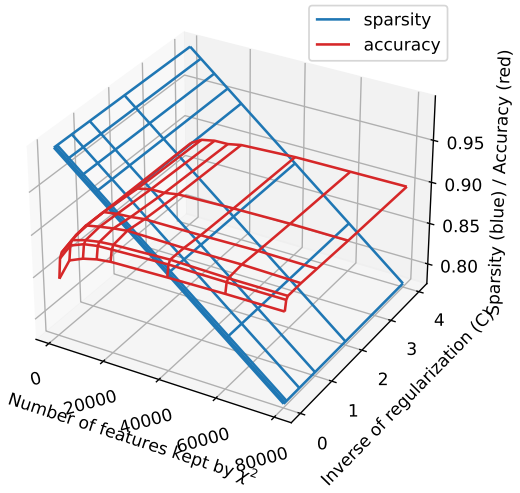

TABLE DES MATIÈRES

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 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()**.

2 VANILLA LOGISTIC REGRESSION

??
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").

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

It is possible to reach a test accuracy of 0.9.

File : **exercice_1_logistic_regression.py**

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.

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

3 UNIVARIATE FILTERING

Univariate filtering consists in statistically evaluating whether each feature carries information about the target. 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. Choose a compromise between sparsity and quality of prediction and save the filtered vocabulary to a different file.

You will need to evaluate the sparsity of the estimator(s). You can measure the degree of sparsity by a number in $[0, 1]$ (1 meaning fully sparse, and 0 full dense). This way, we can plot an image that illustrates this compromise, similar to figure 1.

Now, the pipeline contains the filtering between the one-hot encoding and the scaling. The **SelectKBest** has a **get_support()** method that allows to know which features are kept by the statistical test (this is useful to save the filtered vocabulary).
File : **exercice_2_univariate_filtering.py**
It is possible to have a test score of aroung 89% with a sparsity score of around 83%.

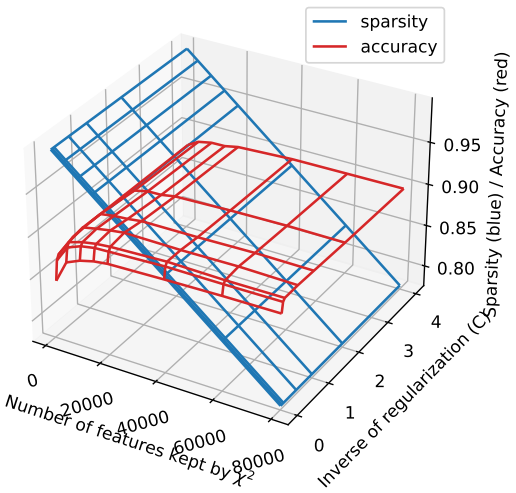

FIGURE 1 – Compromise between sparsity and accuracy. Both sparsity and accuracy are in $[0, 1]$, so they can be represented on the same axis in this image.

3.1 Hyperparameters

Again, the one-hot encoding has some parameters that are important for the statistical test. Indeed, the features could represent the frequency of a n-gram in each document, then it would be a float or int. However, it could also simply represent to presence or absence of the word in each document, and then it is a boolean or an int equal to 0 or 1. Both options make sense. Parameter : **binary** in CountVectorizer.

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

4 EMBEDDED ALGORITHM

Perform the same Pipieline as in **??** , but with a L1 regularization, similar to the Lasso. Again, evaluate the sparsity of the found estimator and search for good regularization hyperparameter values.

File : **exercice_3_l1_regularization.py**