ML Project: FASHION-MNIST

Michele Zanatta
**********@**********

Sara Comelli
**********@**********

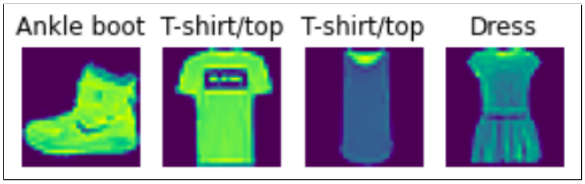

Figure 1. Four images of the dataset with their relative class.

1. Introduction

This project consists in an Image Classification Task.
We have some images of clothes and we have to classify them depending on the type of clothes that they represent. After analysing the dataset (see section 2), we worked with Scikit-Learn to train six models. The procedures of training are explained in section 3. In this section we present each model we trained, how we did train it and which results we got. In section 4 we will select our best two models and see how they work on the test set. It is important to note that the values we write here may vary a little due to randomness. Finally we do some error analysis in order to better understand why they are mispredicting some samples.

2. Dataset

We are using the dataset Fashion-Mnist, that one can find on Kaggle or Github. We loaded the data using the function given by the professor, which, besides loading the data, already flats them into vectors.
The data we are working with are a collection of images of clothes, each belonging to one out of ten categories, labelled as {'T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot'}. Our aim is to build a model that classify the type of cloth correctly. An example is shown in Figure 1.

The size of our data is pretty big: in total we have 70000 samples with 784 features each.
We splitted the dataset into three sets: X_{train} (48000 samples), X_{val} (12000 samples) and X_{test} (10000 samples). Analogously the vector Y containing the predictions was splitted in Y_{train}, Y_{val} and Y_{test}. To do that we have used the function `train_test_split()`.
We have used the training and validation set to train the models and decide which models work better. Finally we have used the test set to compute the performance and to do some error analysis.
Before proceeding with the training part, we checked if the data are "regular" enough. The classes are balanced, so it's not necessary to apply any oversampling or undersampling method. The ranges of the features of X are similar so we didn't use any scaler, except for the Neural Network 3.6. This because we noticed that there weren't significant improvements on the other methods using scaled examples, while there were using the Neural Network.
Thus, without any preprocessing operation we were able to implement the first five models.
For the Neural Network meodel, instead, the heaviness of the data was seriously compromising the performance. For this reason we have preprocessed the data using an hog transformer and scaled them with a standard scaler.

3. Training

After the analysis of the dataset, we trained the following models:

1. Decision tree
2. Random Forest
3. SGDClassifier
4. One vs Rest with Perceptron
5. K Nearest Neighbours
6. Neural Network

We proceeded in this way: after defining the model, we fitted it on X_{train} and Y_{train} and predicted the values of X_{train} and X_{val}, with the function `fit_predict()` that we built "ad hoc". Then we computed the `f1_score` performance on both the training and validation set, in order to discover possible overfitting problems and to have another criteria to compare the models one with each other. We could have used also the `accuracy_score` to measure the performance, since the classes are balanced, but we preferred to use the same metric as we did in class for image classification.

Finally we have also computed the processing time of each model: since the fitting operation for the models is long, it becomes relevant to see how long it is when we choose the best models. In order to do that we built the very basic function `ptime()` that gives us the (approximate) execution time.
Let us now see model by model some useful information and the results we obtained.

3.1. Decision Tree

We implemented a Decision Tree, using GridSearchCV to look for the best parameters between some given. Since we used a GridSearchCV, we could have kept together X_{train} and X_{val}, but we decided to work anyways with just X_{train} (as we did in Lab 5) since we have a lot of data. In particular we reached a good result with `min_samples_leaf` $= 2$ and `max_depth` $= 10$ (see the code for more details). The `f1_score` performance that this Decision Tree achieved is rather satisfactory and the processing time is low:

```
Decision Tree
Train f1_score: 0.8532
Val f1_score: 0.8114
Pr. time in seconds: 80.2406
```

3.2. Random Forest

We implemented a Random Forest, using GridSearchCV to look for the best parameters between some given. We have also implemented a "for loop" to look for a good value for `min_samples_leaf` and we decided to use 10. In this way we lost some performance on the training set but we fixed a little bit the problem of overfitting. The results we obtained are:

```
Random Forest
Train f1_score: 0.9325
Val f1_score: 0.8685
Pr. time in seconds: 45.2129
```

As one can see, some overfitting still remains. To drastically reduce this problem, we could have used `max_depth` $= 5$, instead of $None$ (which is the best parameter found with the GridSearchCV), but in this way we would have lost performance (going from a `f1_score` of 0.93 on training set and 0.86 on validation to 0.76 on both).

3.3. SGDClassifier

We wanted to implement a SVC as we did in class but it was not working. Searching on the documentation about SVC it is written "The fit time scales at least quadratically with the number of samples and may be impractical beyond tens of thousands of samples. For large datasets consider using LinearSVC or SGDClassifier instead". So, instead of using the model SVC, we tried the two suggested. The LinearSVC was very slow and required a big number of iterations, so we used just the second one.
We implemented it without GridSearchCV, using the default parameters. Here the results:

```
SGDClassifier
Train f1_score: 0.8138
Val f1_score: 0.7916
Pr. time in seconds: 125.8729
```

3.4. One vs Rest with Perceptron

The Perceptron is a binary classification model but we implemented it with the One vs Rest strategy, as we did in class (Notebook Multiclass Classification). We used GridSearchCV to look for the best parameters between some given and we obtained good results setting `estimator_alpha` $= 0.0001$ and `estimator_penalty` $= None$:

```
One vs Rest with Perceptron
Train f1_score: 0.7971
Val f1_score: 0.7824
Pr. time in seconds: 46.2756
```

3.5. K Nearest Neighbours

We implemented a K-nn model and we used a "for loop" for the parameter k, assuming values in [2, 4, 6, 8, 10]. We tried to take more and bigger k values, but the execution time was awfully high. We created the lists `f1score_values_train`, `f1score_values_val` and `process_times`, to save the results of `f1_score` (on train and validation sets respectively) and the (approximated) processing times as k changes.
We have then plotted the learning curves with k on the x-axis and the values of `f1_score` on the y-axis, as shown in Figure 2. The values are high but there is a clear problem of overfitting.
We obtained the best results with $k = 4$:

```
4-Nearest Neighbours
Train f1_score: 1.0000
Val f1_score: 0.8572
Pr. time in seconds: 173.0658
```

3.6. Neural Network

At the beginning we implemented a Neural Network, but with our data X_{train} the performance was very poor: around 0.2 on both training and validation sets. This happened also when we did image classification during the Lab 5. As we said, the reason might be the huge size of the data,

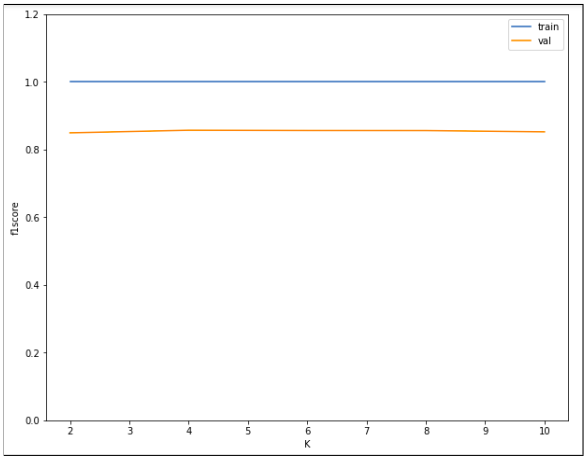

Figure 2. K-nn model. Values of `f1_score` as k changes.

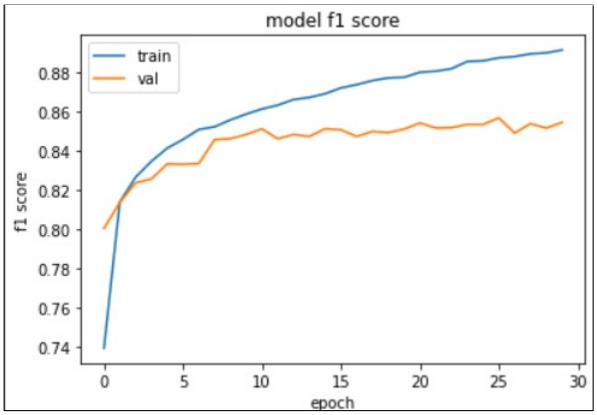

Figure 3. Values of the f1 score on the train and validation set during each epoch of the Neural Network training.

that's why we decided to preprocess the data to reduce their size.
At the beginning we used the rgb2grayscale transformer (which converts the image into a gray scale). In this way the performance improved a little bit (it was around 0.3) but clearly not enough.
Successively we decided to use the hog transformer. After transforming the data, we scaled them using a standart scaler. The data we got at the end of this preprocessing procedure were $X_{train\_hog\_scaled}$, $X_{val\_hog\_scaled}$ and $X_{test\_hog\_scaled}$: the number of samples remains the same, but we drastically reduced the number of features and the size of the data were much more handleable.
With this new data we implemented (for the third time) a Neural Network. We did a "for loop" to decide how many units to put in the hidden layer and we decided to take 50 units. All the details are in the code, where we also printed the summary of the network and the learning curves representing the `f1_score` with the passing of epochs (Fig. 3).

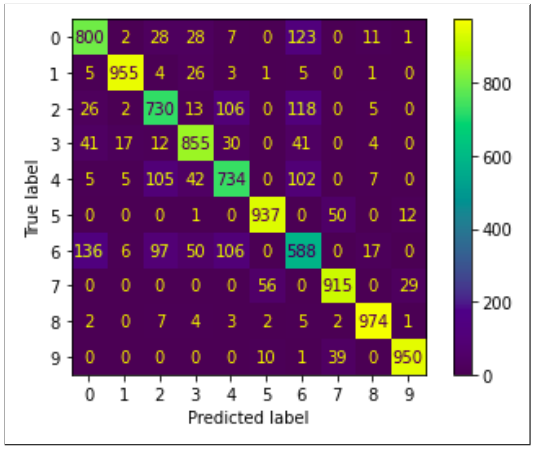

Figure 4. Confusion Matrix for the Neural Network.

We now report the results that we obtained:

```
Neural Network
Train f1_score: 0.8975
Val f1_score: 0.8482
Pr. time in seconds: 144.8699
```

4. Best models and error analysis

4.1. Test Performance

After this training phase we compared the results and we decided to focus on the Decision Tree and the Neural Network. In fact they both have good results in terms of `f1_score` and processing time and they don't seem to present severe problems of overfitting or underfitting.
We computed the performance on the test set, obtaining:

```
Test f1_score Decision Tree: 0.7975
Test f1_score Neural Network: 0.8434
```

that are quite satisfactory results.

4.2. Confusion Matrix

Afterwards we focused on the prediction of the Neural Network for the test set. We printed the confusion matrix (see Figure 4), that represents the classes in which the test samples are classified and their real class.

4.3. Error Analysis

As one can see from the Confusion Matrix, there are some samples of the test set that are misclassified.
For example we have 136 samples of class 6 ('Shirt') mispredicted in class 0 ('T-shirt/Top'). Or 106 samples of class 2 ('Pullover') mispredicted in class 4 ('Coat').
We notice that our main concern is how to classify correctly the examples belonging to class 6 ('shirt'). In fact, that is where our model make the greatest amount of mistakes.

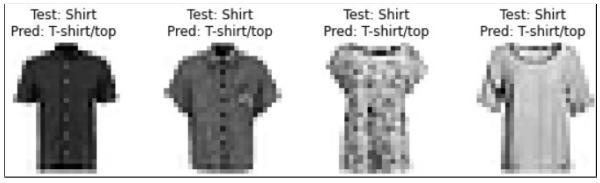

Figure 5. Some mispredicted images.

Thus wanted to see concretely some examples of mispredicted images, that we report in Figure 5. We can see that these mistakes (and also others that are visible in the code) are quite understandable, since they can be easily confused as T-shirt and even humans would argue that, for example, the last two are T-shirts.
So we are overall quite satisfied with the way our model is working and the performance it reached.

References

[1] Scikit-learn guide. `https://scikit-learn.org/stable/user_guide.html`.

[2] Kaggle challenge. `https://www.kaggle.com/c/image-classification-fashion-mnist/`.