Jupyter Notebook Cells Classification – Documentation

Students: Gur Yaniv, Tamir Huber, Yoav Shechter, Omer Edelstein

1. Dataset Description

Our Dataset was collected from scratch using Kaggle API and scraping.

It is consisted of Datasets, Notebooks (for each dataset) and cells (from each notebook) from Kaggle.

- For each dataset we keep – the dataset's Name, Description, Evaluation method (for competitions), Tags (e.g. Finance).
- For each notebook we keep – the related dataset's name, the Notebook's name, the username, score (for competitions).
- For each cell we keep – a unique cell id, the related notebook's name, the cell's source code, Execution count*.

*the cells are numbered by their position in the notebook, it correlates with the real execution count as the cells are executed in-order.

Total of: ~50 different datasets, ~7k Jupyter Notebooks, ~95k code cells.

If necessary for future needs, we can collect more data using the existing infrastructure.

The three datasets that were collected as described were combined and "narrowed" to features that are useful for our purpose (as will be described in section 3).

The final dataset contains a unique cell id, the related notebook's name, the cell's source code, Execution count*, Label (after weak supervision, as will be described).

2. Dataset Analysis Summary

- Mean of 122.8 notebooks per dataset, 16 cells per notebook, 7.2 lines per cell.
- ~3340 empty cells – were deleted.
- ~9160 “useless” cells (contain only comments) – were deleted.

Other possible problems that were noticed:

- Notebooks in the same dataset often have similar cells (some forked notebooks etc…), so datasets with many notebooks might contain sparse data.
- One cell notebooks (all stages in same cell – hard to tag, or just a useless cell).
- Cells with many lines – will be harder to classify since they most probably fit more than one class

We deal with the first problem later, in the second stage (after tagging the data, see section 4) when we shuffle all the data and take a fixed size from each class (also when we tried deleting similar cells in a mini-example the results were worst).

Snorkel (also described in section 4) “deals” with the other problems by creating weights and deciding what is the more “important” tag, or what is the more correct tag for cells that contain code from different stages. Also, the cells code is represented as a fixed size vector in the second stage (after tagging) anyhow, so that there is no need to deal with it in the first stage.

3. Problem Formulation

Given the unlabeled dataset, we want to predict for each cell its data science workflow stage.

The stages (as we defined them for our purpose): Load Data, Data Exploration, Data Preparation and Cleaning, Model Training and Parameter Tuning, Model Evaluation, Imports (cells with only imports).

In order to do so, we need to tag the data using weak supervision methods and then train a multi-class text classification model using the labeled data (see section 4 for full description of solution).

4. Description of Solution

- Weak supervision using Snorkel:

  Snorkel is a system for rapidly creating, modeling, and managing training data. Today's state-of-the-art machine learning models require massive labeled training sets which usually do not exist for real-world applications, as it doesn't exist for our purpose (our initial dataset is unlabeled, as described). Instead, Snorkel is based around the new data programming paradigm, in which the developer focuses on writing a set of labeling functions, which are just scripts that programmatically label data. The resulting labels are noisy, but Snorkel automatically models this process—learning, essentially, which labeling functions are more accurate than others.

  Snorkel provides a general framework for many weak supervision techniques, and as defining a new programming model for weakly-supervised machine learning systems.

  We defined 41 different labeling functions to tag the unlabeled data with the relevant workflow stage, as described in section 3. In addition, we hand-tagged ~1000 cells.

  Snorkel automatically learned a generative model over the labeling functions, estimated their accuracies and correlations, learning from the agreements and disagreements of the labeling functions (Snorkel also provides the user with actionable feedback about labelling function quality, which allowed us to improve the functions accordingly).

  Snorkel generative model (as described in 7.2) outputs the "marginals"- the probability that the cell belongs to each of the classes (stages). We than label the data according to the highest probability.

- Data pre-processing:

  For each cell source code, we removed all comments, as comments may refer to actions that weren't really done or to what was done previously to the current cell, so that it just interferes in our task to classify the current cell correctly.

  The data (source code) was normalized and tokenized- we turned the code to-lower, converted special chars and dots to spaces, than split what's left to tokens by spaces.

- Data Vector Representation:

  Each cell code (after being normalized and tokenized) is represented as a vector of ints that represent the tokens, when only the 8,000 most common tokens gets to be represented, as the less common ones are probably useless for our purpose (meaningless names etc.).

  The vectors are than padded to a fixed max_len of 120 (we looked at the common code length and set the max_length accordingly, so that only outliers will be cut).

  There is a further embedding (see 7.9) of these vectors in the model itself (in the embedding layer).

  -We also tried to train a Doc2Vec Model (see 7.11) over the cells code (using genism) and use this model embeddings as input to the LSTM, but the results were worst. But this could be useful for other similar purposes (to find similar cells).

- End Model – LSTM:

  For our task we need persistence – to consider not only the latest input but also the previous inputs when tagging, we cannot only process single data points, but entire sequences of data. Thus, we need a recurrent neural network.

  There is also a long-term dependency between "words" within a cell's source code and between different cells – we need more context. That's why LSTM is the preferred model for our purpose (simple RNN is uncapable of learning long-term dependencies).

  LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior (as described in 7.6).

  Our LSTM model was built using Keras Deep Learning library.

  The model layers are:

```
                Layer (type)             Output Shape           Param #
=================================================================
           embedding_10 (Embedding)     (None, 120, 512)         4096000
_________________________________________________________________
              spatial_dropout1d_10 (Spatia (None, 120, 512)         0
_________________________________________________________________
              lstm_10 (LSTM)            (None, 64)            147712
_________________________________________________________________
               dense_14 (Dense)          (None, 6)             390
```

  As mentioned, the training set contains vectors of integers that represent the code and one-hot encoding of the label.

  The Embedding layer is used as the first layer. It turns positive integers into dense vectors of fixed size. It resembles the word2vec operation (as described in 7.9) by creating word embeddings. Next layers use the embedding matrix to keep the size of each vector much smaller.

  The second layer is a Spatial Dropout layer. It drops entire 1D feature maps (instead of individual elements in a regular dropout layer). If adjacent frames within feature maps are strongly correlated (as is normally the case in early convolution layers) then regular dropout will not regularize the activations and will otherwise just result in an effective learning rate decrease. In this case, SpatialDropout1D will help promote independence between feature maps and should be used instead. This layer helps preventing overfitting.

  The third layer is an LSTM layer (as described before, see 7.6, 7.7 for a deeper look).

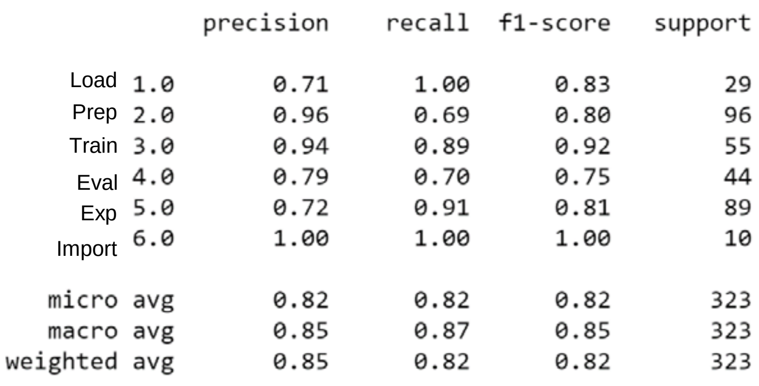
  The fourth layer is a Softmax layer (implemented by keras dense layer). It outputs the Softmax transformation results which are the outputs of our model – probability of the cell to belong to each of the classes. We consider the model's prediction as the class with the highest probability.

5. Findings and Statistical Evaluation

- Weak Supervision evaluation:

  We check the Categorical Accuracy over the test set – how many cells where tagged right (according to the hand-tagged data). We achieved an accuracy of 82.3%.

  The classification report:

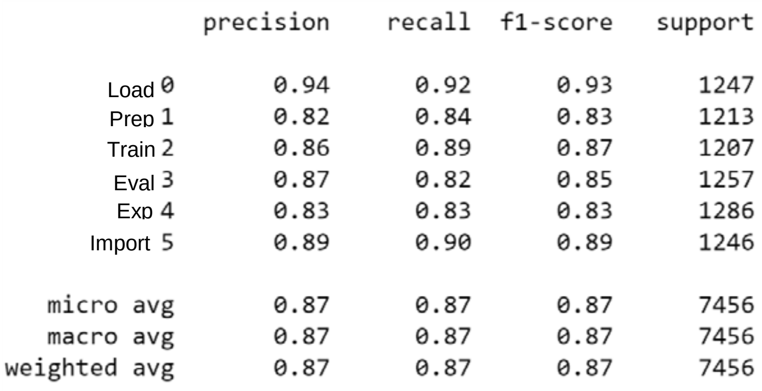


```
                 precision    recall  f1-score   support

   Load  1.0          0.71      1.00      0.83        29
   Prep  2.0          0.96      0.69      0.80        96
  Train  3.0          0.94      0.89      0.92        55
   Eval  4.0          0.79      0.70      0.75        44
    Exp  5.0          0.72      0.91      0.81        89
 Import  6.0          1.00      1.00      1.00        10

    micro avg         0.82      0.82      0.82       323
    macro avg         0.85      0.87      0.85       323
 weighted avg         0.85      0.82      0.82       323
```

  We can see that we label some of the data preparation cells as data exploration, as data preparation recall is lower and data exploration precision is lower, but that's a hard task. Overall the results are pretty good.

- End Model evaluation:
  - Again, we look at the Categorical Accuracy over the test set (it's a different test set than before of course, we train-test-split the tagged data). We achieved an accuracy of 86.7%!

    The classification report:



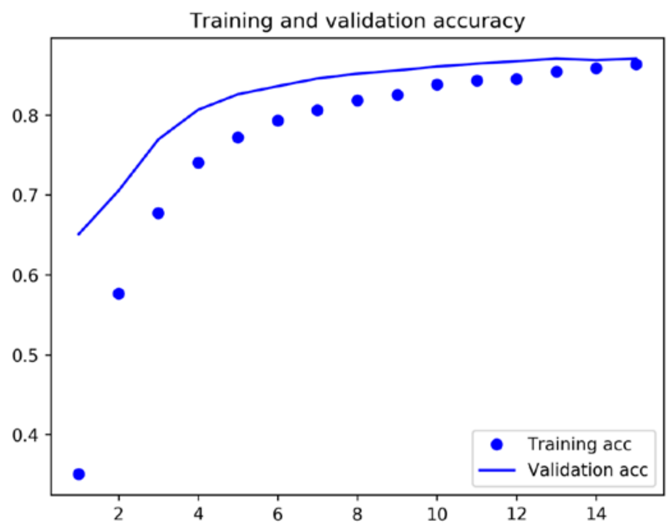
```
                 precision    recall  f1-score   support

     Load 0           0.94      0.92      0.93      1247
     Prep 1           0.82      0.84      0.83      1213
    Train 2           0.86      0.89      0.87      1207
     Eval 3           0.87      0.82      0.85      1257
      Exp 4           0.83      0.83      0.83      1286
   Import 5           0.89      0.90      0.89      1246

    micro avg         0.87      0.87      0.87      7456
    macro avg         0.87      0.87      0.87      7456
 weighted avg         0.87      0.87      0.87      7456
```

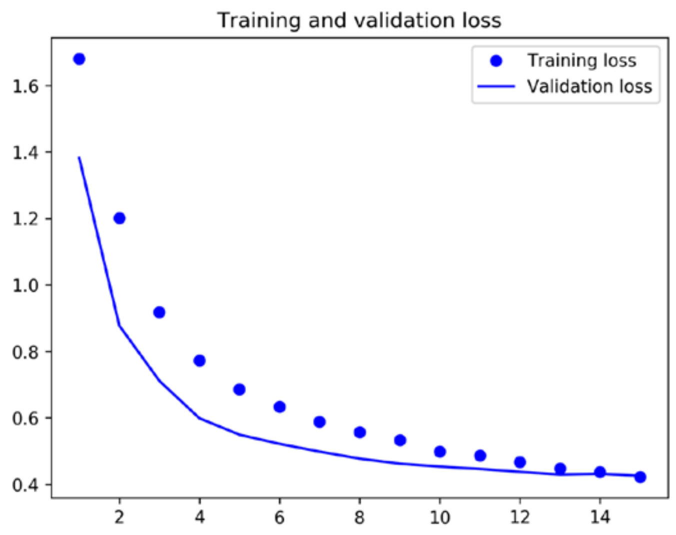
    We can see the model performs well for all labels.
  - Our loss function is Categorical Cross-Entropy. The model is optimized according to this loss function. In addition, when we train the model, we set an EarlyStopping parameter that stops the training if the validation loss stops improving.

    Cross-Entropy is the better option for classification models with a Softmax output node activation (see 7.10).

    We also looked at the loss and accuracy on the train set and the validation set, when the loss is much lower and the accuracy is much higher for the train set, that means we overfitted, so we took that into account when setting the dropout and the number of epochs.

The model convergence graphs:





We stop at 15 epochs to not overfit the model.

We also look at the Mean squared error. We can see that the MSE is low and it is similar between the test set and the validation set.

- Finally, we evaluated "by-hand". Just ran some examples and see that the models classify correctly.

6. Insights and applications

Overall the model works well.

Meaning that for a given data science notebook cell we can know its general purpose with high probability.

This can be useful for many purposes such as Context-Sensitive Recommendations for data scientists (like 7.1). Could be useful for beginners and if improved to create the "perfect notebook" generator. Also can help to "understand code" - interpret a notebook automatically etc.

7. Related Work and References

7.1 REACT: Context-Sensitive Recommendations for Data Analysis by Tova Milo & Amit Somech (TAU)

https://dl.acm.org/citation.cfm?doid=2882903.2899392

7.2 Snorkel: Rapid Training Data Creation with Weak Supervision

https://arxiv.org/pdf/1711.10160.pdf

7.3 Snorkel - A system for rapidly creating training sets with weak supervision

https://hazyresearch.github.io/snorkel/

7.4 Snorkel categorical variables tutorial

https://github.com/HazyResearch/snorkel/blob/master/tutorials/advanced/Categorical_Classes.ipynb

7.5 Text classification guide by Google Developers

https://developers.google.com/machine-learning/guides/text-classification/

7.6 LONG SHORT-TERM MEMORY by Sepp Hochreiter and Jurgen Schmidhuber

http://www.bioinf.jku.at/publications/older/2604.pdf

7.7 Understanding LSTM Networks by Christopher Olah

https://colah.github.io/posts/2015-08-Understanding-LSTMs/

7.8 Keras Documentation

https://keras.io/

7.9 Why You Need to Start Using Embedding Layers by Rutger Ruizendaal

https://towardsdatascience.com/deep-learning-4-embedding-layers-f9a02d55ac12

7.10 Why You Should Use Cross-Entropy Error Instead Of Classification Error Or Mean Squared Error For Neural Network Classifier Training by James D. McCaffrey

https://jamesmccaffrey.wordpress.com/2013/11/05/why-you-should-use-cross-entropy-error-instead-of-classification-error-or-mean-squared-error-for-neural-network-classifier-training/

7.11 Distributed Representations of Sentences and Documents (Doc2Vec) by Quoc Le and Tomas Mikolov (google)

https://cs.stanford.edu/~quocle/paragraph_vector.pdf

7.12 Gensim Doc2Vec tutorial

https://github.com/RaRe-Technologies/gensim/blob/develop/docs/notebooks/doc2vec-IMDB.ipynb