Repository and Directory Structure

The git repository is here: dtecu/kaChallenge and it is public. The provided data file: **data.json** was not added to this repository.

It contains the following files and directories:

- `README.md` – empty file.
- `bert_balancedOversampled.py` – entry point for training and validation with bert; it balances the dataset by oversampling
- `bert_ensemble.py` – entry point for training and validation with bert ensemble; it also uses the ensemble to balance the data
- `bert_simple.py` – entry point for training and validation with bert ignoring the fact that data is imbalanced
- `lib/` – contains a few modules used in the python files of this directory
- `results/` – contains the output as well as the saved results of the runs
- `test/` – contains a brief test of some of the functions implemented inside the modules in `lib/`
- `tf_idf_with_answers.py` – entry point for TF_IFD based training and validation; it uses the answers too
- `tf_idf_without_answers.py` – entry point for TF_IFD based training and validation; it does not use the answers
- `workspaceSetup.py` – convenience file to automate the fetching of the whole repository into a specific location

Each of the subdirectories of `results/` contains at least an **output.txt** file. This file contains the output generated during training and validation. Usually, the final results are among the last lines. All the bert* runs also saved the final prediction on the validation set (probabilities) and the expected categories. In this way we can restore the tensors and we can reproduce the metrics or create new metrics without saving the bert model (one bert is about 420MB) or without training model again. The output tensors are very small in comparison.

The `lib/` directory contains some modules defining various functions that are used by the *"toplevel"* scripts in the root directory.

The **data.json** file was purposely not added to this repository. It belongs to a company and it may contain data that is not meant to be public. So, in order to reproduce these results, the file needs to be manually fetched in the current directory. The current directory is the directory hosting the repository locally.

Data

A small correction has been done on the training data: the case for category 'Off-Topic' was not consistent throughout, so I "normalized" all occurrences to: 'Off-topic'. After this modification all data was labelled consistently.

There are 1185 total entries in the data grouped like this:

```
{'Discovery': 734, 'Troubleshooting': 199, 'Code': 59, 'Comparison': 50, 'Advice': 53, 'Off-topic': 90}
```

We notice that the data is very imbalanced.

By validation, within this document, it is meant the part of the data never shown in training and used only to validate the final performance of the model. Usually, 00% (maximum 20%) of the training data is used for validation.

After reserving 10% of the data for the final validation, we split the rest 90% in training data and test data (usually 90% - 10% split).

Measuring Model Performance

Measuring only accuracy can be misleading for the current dataset, because the data is heavily imbalanced. Accuracy could be very high, and still the model could perform very badly on text belonging to the under-represented classes.

This is why I used the F1 score in addition to accuracy: F1 macro, micro and weighted.

In the final reporting I preferred to report the accuracy for each of the 6 classes, separately.

Bert Simple

In this approach, I use BERT (bert-base-uncased) to classify data. In this simple approach the fact that data in imbalanced is completely ignored. Data is shuffled (I use seeds in order to try to get a reproducible run – which may also depend on the hardware) and 10% (randomly chosen) is reserved for validation. The rest 90% is split into 10% - 90% for test and train.

The model is trained and the validation results are presented at the end. It is very easy for this model to overfit. The training loss can be made arbitrarily small in less than 35 epochs, while test loss seems to "jump" randomly.

In order to avoid overfitting I "played" with some parameters: batch size, number of epochs, learning rate, weight decay, dropout. The most successful was early stopping: stopping (and choosing the best model) after 4 epochs of no-improvement.

Another thing which I didn't try, but would like to, would be to freeze certain layers of BERT and finetune only a few.

The validation results with this approach are:

- Overall: {'Accuracy': 0.83, 'F1Macro': 0.72, 'F1Micro': 0.83, 'F1Weighed': 0.83}
- Category based accuracy: {'Discovery': 0.875, 'Troubleshooting': 0.82, 'Comparison': 1.0, 'Code': 0.33, 'Advice': 0.75, 'Off-topic': 0.62}
- Validation sample size: {'Discovery': 72, 'Troubleshooting': 17, 'Comparison': 10, 'Code': 3, 'Advice': 4, 'Off-topic': 13}

The exact details about which classes were mis-predicted are found in the corresponding **output.txt** file.

Bert Balanced Oversampled

It is very similar to bert simple described above. The difference consists in overloading the DataLoader that the Trainer uses in order to oversample the under-respresented data. The procedure is coded in the **lib\ balancedOversampledTrainer.py** file, using a weighted random sampler. This was performed in order to balance the data, such that under-represented (like 'Code', 'Comparison', 'Advice') classes become "better" represented during the training.

The validation results with this approach are:

- Overall: { Accuracy': 0.80, 'F1Macro': 0.72, 'F1Micro': 0.80, 'test_F1Weighed': 0.80}
- Category based accuracy: { 'Troubleshooting': 0.75, 'Off-topic': 0.4, 'Discovery': 0.92, 'Code': 0.71, 'Comparison': 0.8, 'Advice': 0.5}
- Validation sample size: {'Troubleshooting': 28, 'Off-topic': 10, 'Discovery': 65, 'Code': 7, 'Comparison': 5, 'Advice': 4}

The exact details about which classes were mis-predicted are found in the corresponding **output.txt** file.

Surprisingly, bert balanced oversampled performs worse than simple bert at almost all metrics (except one: accuracy for 'Discovery').

Bert ensemble

In this approach I train 15 models, bert based. Each of them is trained on balanced data. Each of the 6 categories has a number of samples equal to the least representative category (that is 45 = 0.9 * 50). For each model, the train data for the least represented category is the same. But for the other categories (which have more samples), we use the next available data such that finally all the training data is used. The procedure is coded in the **lib\balancedDatasetExtractor.py** file.

Why 15 models? The least under-represented class has 45 available items for training and the most representative has 660 (45*15=675 to cover the 660).

Each of the 15 models is trained in a very similar way as the simple bert. I wanted to avoid storing each of the 15 models (420MB each), so after training the model I already evaluated it on the validation set and stored the result.

At the end of training all 15 models of the ensemble, we have a tensor of shape (15, 120, 6). 120 is the length of the validation data.

I chose 2 ways to predict the final outcome.

One way is based on majority voting. Each model choses its "winner" and the result of the ensemble is the majority category.

The other way is to average (or to sum which is perfectly equivalent for this purpose) the probabilities of all models for one prediction. Then we end up with 6 numbers per prediction and the biggest one wins. This way turned out to be slightly better (but not by much).

The validation results with this approach are:

- Overall accuracy: 0.68
- Category based accuracy: {'Advice': 0.67, 'Discovery': 0.69, 'Comparison': 1.0, 'Off-topic': 0.78, 'Troubleshooting': 0.5, 'Code': 0.83}
- Validation sample size: {'Advice': 6, 'Discovery': 74, 'Comparison': 5, 'Off-topic': 9, 'Troubleshooting': 20, 'Code': 6}

More details are found in the corresponding **output.txt** file.

Bert ensembles seems to be worse than bert oversampled which is worse than simple bert!

TF-IDF without using answers

Just to mention that the none of the bert approaches above used the answer. Probably, that was a mistake!

Neither this TF-IDF approach uses the answers. But the next one does.

For the TF-IDF approach I split the data for train and validation in an 80% - 20% ratio.

Then, from all the training data I form 6 documents by concatenating all questions that belong to the same category. I get in the end 6 big documents; one for each category.

For each of the test questions I measure the TF-IDF based similarity with the 6 documents. The most similar wins. Everything is matrix based and it is very fast, even on CPU.

The validation results with this approach are:

- Accuracy for Discovery : 0.75
- Accuracy for Troubleshooting : 0.575
- Accuracy for Code : 0.5
- Accuracy for Comparison : 0.8
- Accuracy for Advice : 0.73
- Accuracy for Off-topic : 0.39
- Overall accuracy: 0.68

TF-IDF using answers

This approach is very similar to the previous approach with the only difference that the answers are also concatenated to the questions (only in the training set).

The reason they are concatenated only to the training set and not to the validation set is due to the following assumption: I assumed that we need to classify the question before we have an answer. I assumed the classification of questions is needed before the answer is available in order to select the best entity to answer it.

The validation results with this approach are:

- Accuracy for Discovery : 0.46
- Accuracy for Troubleshooting : 0.425
- Accuracy for Code : 0.5
- Accuracy for Comparison : 0.9
- Accuracy for Advice : 0.45
- Accuracy for Off-topic : 0.39
- Overall accuracy: 0.47

It seems that using the answers produces a worse TF-IDF classifier.

Conclusion

The conclusion is that the simple bert model was the most successful. Its overall accuracy is 83%.

I was expecting a better result.

One problem I did not have time to mitigate is related to possible “outliers”. By that I mean some questions that are marginally tagged with a category, but could very well be something else. If I had time, I would try to exclude (clean-up the data set) some of the questions for which the prediction uncertainty is high. They might “confuse” the model.