Assignment 1: Sentiment classification, introduction to computational modeling (non-graded part)

Instructions

- We will be working on this assignment in class and at home, during the first four weeks. AulaGlobal tells you how far you should get prior to each lecture.
- Assignment 1 has an **ungraded** part and two **graded** parts. The current document contains only ungraded exercises. None of the ungraded part needs to be handed in. Doing these exercises serves as preparation for the graded part (and to learn ;).
- The **graded** part will be published in a separate document, on Week 3.
  - It will consist of exercises that build closely on the exercises of the non-graded part.
  - You will need to submit it on Weeks 4 + 5 (we will provide detailed instructions).
  - The graded part will contribute 30% to your final grade (see 'General information' on AulaGlobal for more information about course grading).
- The assignment has both coding exercises and more conceptual questions, and this will be true also of the graded part.
- Highlighted parts will be covered in class (time permitting). Code for the highlighted exercises will be released after each class, for your reference.

A. Getting started

1. Download all files for Assignment 1 and place them in a folder called *assignment1*.
2. Go to this folder in your file explorer of choice, and find the file *data/sentiment_10.csv*. What is a **.csv** file, anyway?
3. The files *data/sentiment_train.csv* and *data/sentiment_val.csv* contain part of the full dataset that we will work with, whereas *data/sentiment_10.csv* contains a smaller sample, which makes it easier (faster) to use initially as we're trying things out. Open the smaller *sentiment_10.csv* file in a spreadsheet software of choice. What do the **columns** and **rows** in this .csv file represent? (You can close that file now.)
4. Which text processing steps have been applied to the texts in *sentiment_10.csv*? (Check the introductory tutorial on Aula Global if you're lost in this question.)
5. What does sentiment have to do with semantics?
6. Launch the program **Spyder**.
7. In Spyder, in the menu, select **Projects**, then **New Project…** and choose 'Existing directory' and the folder *assignment1* that you created.
8. The Spyder interface is built up of *panels*. Make sure you can find the following panels: the **Project** panel, the **Editor**, the **IPython Console**, the **Plots** panel and the **Variable Explorer**.
9. What are the respective functions of the **Editor** and the **IPython Console**?
10. What are the different ways in Spyder to **run** a piece of your code? We recommend you to use *Run* (shortcut F5), activating the option to erase all variables before execution (why?),

and also, for quick checks, marking the portion you want to run (different operating systems have different shortcuts).

B. Loading and inspecting the data

1. Create a file **main.py**, which is where you will put the code for this assignment. Using Python's **pandas** package, load the file *data/sentiment_10.csv* and assign it to a variable named *data*. This variable now contains a pandas *DataFrame*. *Note*: You can use the tutorial/cheatsheet on Aula Global to get you up to speed with this and the following exercises; there are tons of useful cheatsheets online, and much other material.
2. Print the first 3 rows of the DataFrame. Pandas may try to be nice and omit some columns (showing "..." instead) to avoid filling your entire window. To prevent this, you can change panda's *display.max_columns* value: see this answer.
3. Print only the 'sentiment' column of the dataframe. Compare the output to the original *sentiment_10.csv* file
4. Using Pandas commands, answer the following: How many rows and columns does the *data* dataframe have? How many positive/negative examples?
5. Instead of printing to the console, Spyder also offers a convenient **Variable explorer**. Inspect the *data* variable there. *Warning:* relying on the Variable explorer is risky, because it represents the final state of the computer after the last run, which need not correspond to the code shown in the editor (e.g., if you rename a variable in the editor, it won't be renamed in the Variable explorer).
6. Create a new column in the same DataFrame, called *n_characters.* It should contain, for each row, the number of characters (i.e. 'length') of the string in the *text* column.
7. Create a new column (name: *tokens*), containing, for each row, the **list** of elements ('tokens') in its text. (Tokens include words and things like punctuation marks; see tutorial in Aula Global if you're not familiar with the notion of *token*.)
8. Open the original .csv file again. Can you find the columns that you added? Why (not)?
9. Find a way to print a **summary** of the DataFrame, for instance with the means and standard deviations of numerical columns (including the new columns you computed).
10. Now generate the summary for a bigger dataset (*sentiment_train.csv* instead of *sentiment_10.csv*). Running this code may take a bit longer. What can you conclude from the results?
11. Later we will be working with this bigger dataset. However, for testing new bits of code it will still be convenient (faster & use less electricity) to use the smaller dataset (*sentiment_10.csv*). But switching between datasets is a bit risky – you may forget to switch back to the full dataset! To prevent this, define a boolean variable called *TRIAL* at the top of the file (but below the *import* statements). This variable must be set to True or False. Then make sure the full dataset is loaded if *TRIAL is false*, while the smaller dataset is loaded if TRIAL is true. In the latter case, make your program automatically print a warning (something like "Don't forget: trial dataset is used!").

C. Computing features for sentiment classification

1. For classification tasks, an input text is usually first transformed into a list of **features**, i.e., numbers representing various aspects of the text. (Silly examples of features: whether or not the text contains the letter 'h'; the number of times it contains the word "kitchen"...) The classification decision is then made based on these features instead of the original, raw text. Why?
2. Try to think of **2 potentially useful features** for sentiment classification. Formulate them in English; no programming required yet.
3. Compute as feature the number of 'positive' words in the text. (You yourself can define simple, not too long lists of words you think are positive/negative.) Computing a feature can be done by taking the *text* column, **applying** an operation to each value in the column, and storing the result in a new column with an appropriate name (use *n_pos_words)*.
4. Compute 3 other features that you think may be useful (note: you may find it difficult to think of these features; look at the data in *data/sentiment_train.csv* for inspiration). Each computed feature can be stored in its own column in the DataFrame.
5. Print some summarizing info about your computed features, for instance, the minimum and maximum, the mean, and the standard deviation.

D. Clean your code

1. Put each portion of code that "does a single thing at a single level of abstraction" into a separate **function**. (For guidance, see the 'SLAP' principle here; see also the 'DRY' principle.)
2. Move any "**magic values**" into constants defined at the top of the file (underneath the *import* statements), conventionally named with full capital letters. For instance, in the data loading code, replace the string 'data/sentiment_*.csv' by the constant PATH_TO_SENTIMENT_CSV, which you define at the top of the file. Why would this be a good approach?
3. For further cleaning, you can refer to the **Python style guide** PEP8, as well as the many summaries that exist (e.g., here).

E. Generating predictions based on the features

1. Define a function *predict_by_n_positive_words* that looks at the feature *n_pos_words* and outputs a prediction depending on a threshold: if the feature value is bigger than a chosen threshold, predict positive sentiment, else predict negative sentiment. I will refer to this rule as "rule 1".
2. Use the prediction rule to **generate predictions**, that is, to determine, for each datapoint, whether it is positive or negative. Store the predictions as a new column in the main DataFrame *data* (using a sensible name for the new column, e.g., *predicted_by_rule1 or pred_rule1*).
3. Rule 1 is quite limited; define other rules that you may think may work better, using the features that you computed earlier (including *n_pos_words* if you want).
4. Generate predictions for these rules and store them as new columns.

F. Evaluating the predictions

1. Check the predictions of rule 1 (from exercise E1) on the small dataframe by hand: what **accuracy** does it give? Recall that accuracy is the proportion of items where the predicted sentiment matches the true sentiment (or "gold label"). How good is it?
2. When developing computational models, it's very important to do **error analysis**, that is, analyze the mistakes of the model (as well as what it does right). Why?
3. Let's get started on error analysis: Compare the predictions of rule 1 with the gold labels for the examples in the small dataframe; see what the current rule gets right/wrong and think of ways to improve over it.
4. Define a function that, given a column of predictions you computed and the original sentiment column, computes **accuracy**. Does it give the same result as the one that you computed by hand? (It should. :)
5. Now is the turn to use the big dataset (for now, file *sentiment_train.csv*). Generate predictions with the different rules that you generated, and compute the accuracies of the different rules.
6. Compare the accuracies of the rules you defined: Was any prediction rule better than the others? Can you tell if any of your rules is any good?
7. **Clean up your code.** There are a number of ways to keep track of multiple prediction rules and apply them to the data one after the other. A simple way is to define a separate function for each prediction rule (*predict1*, *predict2*, …; or better yet use more informative names) and store all of these functions in a list. In this way, you can easily loop over the prediction rules and, on each iteration, apply the given rule and assign it to an appropriately named column. This will come in handy for exercise 9 below.
8. Analyze the predictions of the best model, by inspecting a random sample of 30 texts.
9. Based on the results of the analysis, try to **improve your prediction rules** to obtain higher accuracy scores, by changing the rules, re-computing the predictions, and evaluating them again. *Note*: For now, **do not make any changes to the features** themselves; you can make notes to yourself for ideas about changing the features, as they will come in handy in the next few weeks.
10. Record your best rule and best accuracy, to be discussed in class.

G. More methodology: Baseline, train/val/test partition, confusion matrix

1. [Pencil and paper exercise.] On the small dataset, add the following two types of 'predictions', which will serve as *baselines*: 1. **always positive** sentiment, 2. **always negative** sentiment, 3. **random** assignment to positive or negative. Compute the accuracy of these baselines by hand.
2. The **baselines** don't really generate useful predictions (consider why not). Why is it nevertheless a good idea to compute them? In other words, what purpose does a baseline serve, or what information does it give us?
3. Implement the baselines from exercise G1. MIIS students should also implement a 3rd baseline: **random positive/negative** sentiment with probabilities in accordance with the

proportions of positive/negative in the training data (*sentiment_train.csv*); for instance, if 75% of the items are positive, choose 'positive' with 0.75 probability.

4. Compute the accuracies of the baselines on the data in *sentiment_train.csv*. Does any give a better accuracy than the others? Why / why not?
5. So far we've been finetuning and evaluating our prediction rules **on the same dataset**. If your goal is to know how good the predictions are, why is this problematic? Can you think of a solution? (Explain in ordinary language; no programming required.)
6. Check the performance of the rules you developed in exercise E3 on the development set (file *sentiment_val.csv*). Do they perform approximately as good as in the training set, better, or worse?
7. Call the pandas function *crosstab*() with two arguments: the *sentiment* column and the column containing predictions of rule 1 from exercise E1. Print the resulting table. This is known as **cross-tabulation** or **contingency table**. What do the numbers represent? How does it help evaluate the model?
8. When a contingency table is used to compare a model's predictions to the real data, as we did above, it is known as a **confusion matrix**. Does the name make sense?
9. Confusion matrices help in model development, by giving insight into the kinds of errors the model makes. Print the confusion matrices obtained with the different classifiers that you have built so far (one per prediction rule). Do they differ in the types of errors they predominantly make?

H. Logistic regression

1. We will next develop a Machine Learning-based computational model that does sentiment analysis through logistic regression.
2. In what follows we will only use two features to make predictions: *number of positive words* and *number of negative words.* By convention, in Machine Learning, the data we use to make predictions (the feature representation of the input) is designated with X, and the target outputs (in our case, 'pos' and 'neg' labels) with y. Rename your variables according to this convention: take the two feature columns from *data* and assign them to a new variable called *X*; next, take the target outputs (i.e., the column *sentiment*) and assign these to a new variable called *y*.
3. Create a **logistic regression** model using the SciKit-Learn package, assign it to a variable called *model*, and fit it to the data stored in *X* and *y*. *Note:* https://scikit-learn.org provides a lot of learning materials (see 'Fitting and predicting' under 'Getting started' for this exercise and the next); and we find this cheatsheet useful.
4. The logistic regression model has a method called *predict.* Use it to generate **predictions** and assign them to an appropriately named column in the original DataFrame *data*. For the moment, use the training set to generate predictions; in exercises below, we will use the validation set instead.
5. The logistic regression model provides two methods for generating **predictions**: *predict* and *predict_proba*. Without going into technical details, can you try to explain the difference?
6. Evaluate the predictions of the logistic regression model, using the *accuracy* function you defined before.

7. How does the logistic regression model compare to your manually tweaked prediction rule(s) from above? And the baselines?
8. The **coefficients** of your fitted model indicate how each feature contributes to a 'positive' decision. Print the model's coefficients. What do the coefficients tell you about the features you used?
9. Generate predictions for the validation set. Hint: What do you need to do before being able to use the predict method on these data?
10. Compute the accuracy of the model on the validation set. Does it perform approximately as good as in the training set, better, or worse?
11. Compute accuracy, precision, and recall of the model's predictions on the validation set using **functions from SciKit-Learn**. Double-check that your implementation of accuracy gives the same result as the one from SciKit-Learn.
12. The logistic regression model itself also comes with a method *score()*. What does it do?

I. Error analysis

1. Using the accuracy, precision, and recall values obtained in exercise H1, try to understand which kinds of errors are more prevalent in the model. Using the notions of 'true positive', 'true negative', 'false positive' and 'false negative' will help (as explained in Jurafsky & Martin chapter 4).
2. Print confusion matrices (recall: use the *crosstab()* function) for the **baselines**, and try to interpret them. Do the baselines differ in the types of errors they predominantly make? Can you see why?
3. Print the confusion matrix of the logistic regression classifier. Does it differ from the rule-based classifiers in the types of errors it predominantly makes?
4. SciKit-Learn also provides a function for creating a confusion matrix. Does this result in the same table as pandas' *crosstab*? What are differences? Which function do you find more convenient, pandas' or SciKit-Learn's?
5. [Note: difficult exercise!] Do you get the same insights about model behavior from accuracy, precision, and recall scores as you get from the confusion matrix, or do they yield different insights?
6. Is accuracy a good measure for the current dataset? Why?
7. Select 20 random false positives and 20 random false negatives from the training set. Can you identify patterns in the cases where the model fails?
8. Try to transform these insights into ideas about how to improve the model. Don't implement them – just think. :)