Developing an automatic text classification model

Student Number: 24885531; this report is submitted in fulfilment of the requirements for course CSMBD16-17, department of Computer Science, University of Reading

Abstract

The aim of this coursework is to create a classification model using KNIME and KNIME Labs that is capable of separating documents from 10 different categories with a high accuracy. Two different pipelines, Baseline and Final, have been created and they both fulfill the task - with 61.8% and 89.3% overall accuracy respectively.

Introduction

The sample data for the task was provided as part of the assignment and is comprised of texts from 10 different newsgroups. Those are effectively emails with From/Subject headers and email bodies. The data is conveniently separated into training and test sets - 4,647 and 2,235 files, respectively, that reside in separate directories according to their categories.

Results

The Baseline and Final pipelines, naturally, lead to different results / overall accuracies.

Baseline Pipeline

The **Document Vector** that the Decision Tree model operates on is composed of 6,972 rows (one per each document) and 62,923 columns (one per each term). This is further split into training (4,674 records) and test (2,325 records) sets. The overall accuracy is 61.8%.

Row ID	TruePositives	FalsePositives	TrueNegatives	FalseNegatives	Recall	Precision	Sensitivity	Specifity	F-meas...	Accuracy
1	152	109	1973	91	0.626	0.582	0.626	0.948	0.603	?
2	132	151	1929	113	0.539	0.466	0.539	0.927	0.5	?
3	150	85	1993	97	0.607	0.638	0.607	0.959	0.622	?
4	244	111	1964	6	0.976	0.687	0.976	0.947	0.807	?
5	178	46	2029	72	0.712	0.795	0.712	0.978	0.751	?
6	104	131	1949	141	0.424	0.443	0.424	0.937	0.433	?
7	140	66	2012	107	0.567	0.68	0.567	0.968	0.618	?
8	146	52	2026	101	0.591	0.737	0.591	0.975	0.656	?
9	109	95	2036	85	0.562	0.534	0.562	0.955	0.548	?
10	82	42	2126	75	0.522	0.661	0.522	0.981	0.584	?
Overall	?	?	?	?	?	?	?	?	?	0.618

Figure 7: Baseline pipeline results: 61.8% overall accuracy

Final Pipeline

The **Document Vector** that the SVM model operates on is composed of 6,972 rows and 6,873 columns (relevant terms). The train/test split is the same as in the Baseline Pipeline - 4,674 train vs. 2,325 test records. The much lower number of columns allows us to run an SVM model on this data set, achieving a higher overall accuracy of 89.3%.

Row ID	TruePositives	FalsePositives	TrueNegatives	FalseNegatives	Recall	Precision	Sensitivity	Specifity	F-meas...	Accuracy
10	130	12	2156	27	0.828	0.915	0.828	0.994	0.87	?
1	221	31	2051	22	0.909	0.877	0.909	0.985	0.893	?
2	214	47	2033	31	0.873	0.82	0.873	0.977	0.846	?
3	227	24	2054	20	0.919	0.904	0.919	0.988	0.912	?
4	234	15	2060	16	0.936	0.94	0.936	0.993	0.938	?
5	244	16	2059	6	0.976	0.938	0.976	0.992	0.957	?
6	195	46	2034	50	0.796	0.809	0.796	0.978	0.802	?
7	224	12	2066	23	0.907	0.949	0.907	0.994	0.928	?
8	218	13	2065	29	0.883	0.944	0.883	0.994	0.912	?
9	169	33	2098	25	0.871	0.837	0.871	0.985	0.854	?
Overall	?	?	?	?	?	?	?	?	?	0.893

Figure 8: Final pipeline results: 89.3% overall accuracy

Methods

A number of traditional Natural Language Processing (NLP), Machine Learning (ML) and Data Mining approaches have been used to preprocess the data, classify the documents and test the respective models' accuracies. The general approach taken was to create a Baseline preprocessing pipeline that would give results with minimum work/transformation - and then create a Final pipeline that would improve on the baseline. The motivation behind the baseline and the final pipeline was that the input they receive is the same and their output is of the same format - which enables one to run the same predictive models on the outputs but get different results in terms of accuracy.

The resulting KNIME workflow:

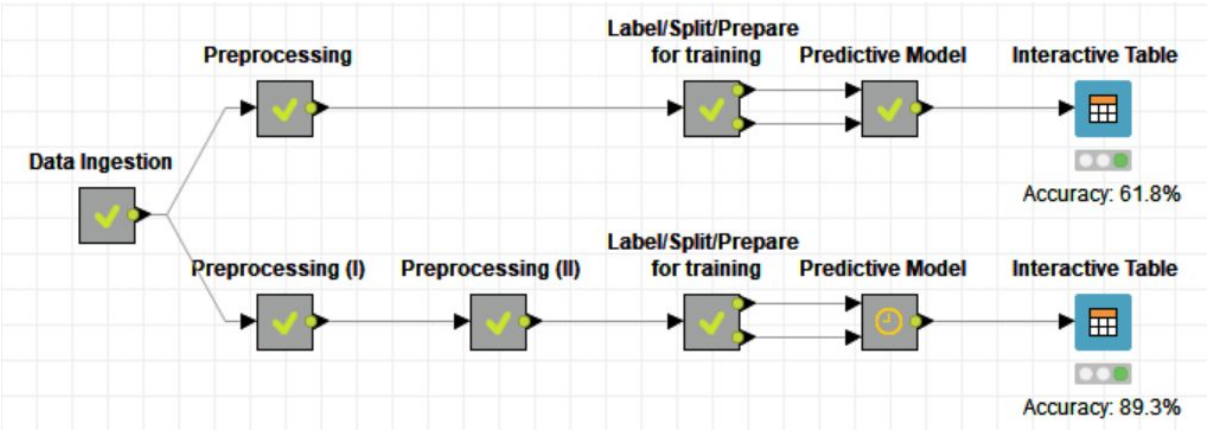

Figure 1: Baseline (top) and Final (bottom) pipeline along with their respective accuracies

Data ingestion

The following figure outlines the data ingestion workflow.

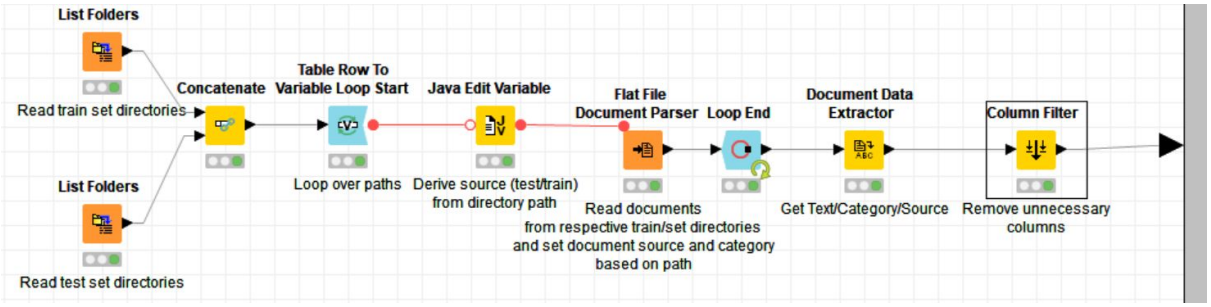

Figure 2: Data ingestion workflow

The input to the **List Folders** nodes on the left are the **train** and **test** directories from where the data resides C:\git\uor_csmbd16\10newsgroup. The list of folders is then concatenated and their subfolders are then looped through and the files in them are read into documents via the **Flat File Document Parser** node. This node uses the output from **Java Edit Variable** which provides it with the source (train/test) and category (1..10) of the respective file.

Baseline pipeline

From a bird's eye view, the Baseline pipeline has the following steps(Figure 1, top part):
Data Ingestion -> Preprocessing -> Label/Split/Prepare for training -> Predictive Model
The Data Ingestion is discussed above, what follows is a description of the rest of the components.

Preprocessing

The baseline preprocessing is meant to be crude but with a specific output: a Document Vector.

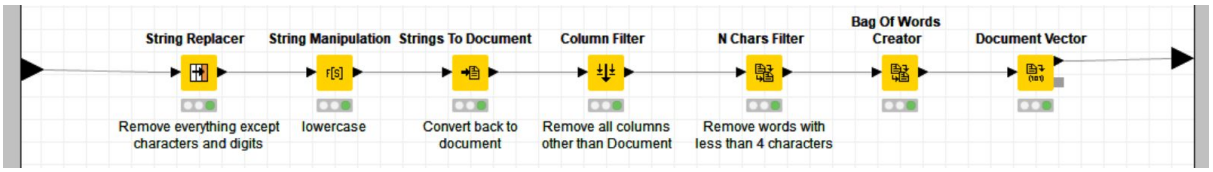

Figure 3: Baseline Preprocessing workflow

Preprocessing steps:

- **String Replacer** replaces all non-digit, non-alphabetic and non-space characters with a space using the [^\d\w\s]+ regular expression. The overall effect of this is removing punctuation. The reason why this was achieved via string replacement rather than the **Punctuation Erasure** node was that the latter ended up not stripping all special characters and there were issues creating a Document Vector.
- **String Manipulation** lowercases the text of the document.
- **Strings to Document** is then used to convert the document text back into a document type which, in turn, is passed on to the rest of the pipeline.
- **Column Filter** leaves only the relevant data that the rest of the pipeline needs which is one **Document** per row.
- **N Chars Filter** removes words that have less than four characters.
- **Bag of Words Creator**
- **Document Vector**

The results of the **Preprocessing** metanode are then fed into the **Label/Split/Prepare for training** metanode.

Label/Split/Prepare for training

This metanode is used by both the **Baseline** and **Final** pipelines. There are two identical copies of this metanode, each dealing with the respective pipeline - having two of them allows for more ease of development so that testing the **Final** pipeline would not reset the results of the **Baseline** pipeline.

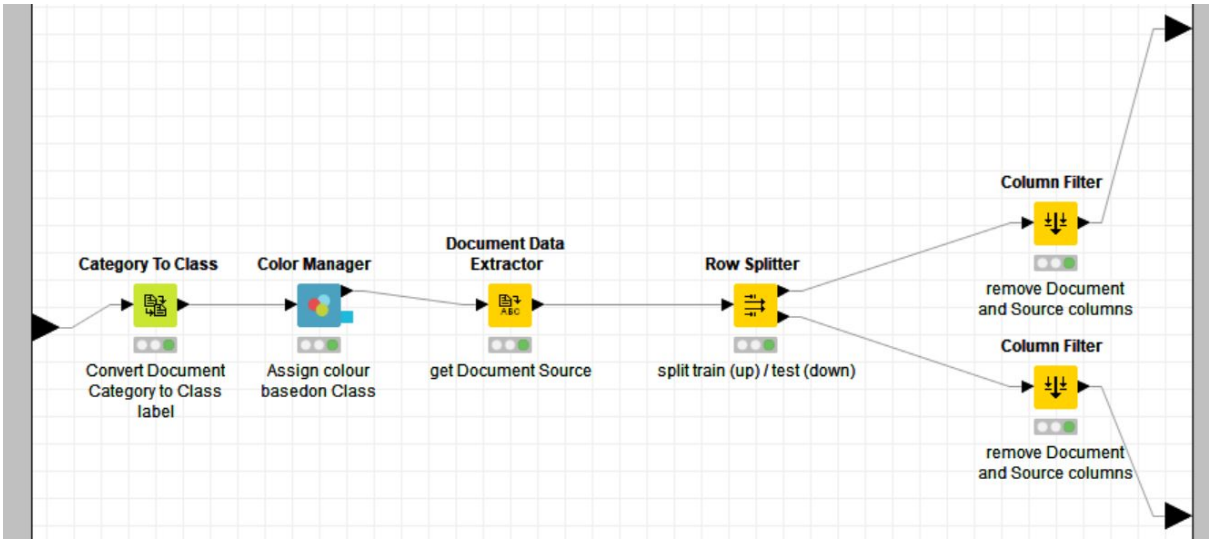

Figure 4: Label/Split/Prepare for training workflow - reused by both Baseline and Final pipelines

The metanode executes the following steps:

- **Category to Class** is used to convert the document category (1..10) to a class label which will later be used by the predictive models.
- **Color Manager** assigns colour values to the ten class labels.
- **Document Data Extractor** extracts the document **Source** column which has the values of either “train” or “test” as set by the **Data Ingestion** metanode described previously.
- **Row Splitter** splits the data into a training and test set (based on the value of the Source column) which are then fed separately to the predictive models.
- **Column Filter** removes all irrelevant columns such as “Document” and “Source” from the document vector - leaving only numeric columns and a Class label.

The resulting matrices only contain the document vectors themselves and the document’s classes - this is all the information that is required by the predictive algorithms that follow this metanode.

Predictive Model

This node receives its input from **Label/Split/Prepare for training** and can run an arbitrary learner/predictor in order to output a table with Accuracy statistics. In the Baseline pipeline case, the predictor is a vanilla Decision Tree model. For the Final pipeline, a Support Vector Machine (SVM) predictor is used.

Final Pipeline

The Final Pipeline (Figure 1, bottom part) differs from the Baseline Pipeline in two ways - it uses different Preprocessing steps and a different Predictive Model (SVM). The rest of the components - **Data Ingestion** and **Label/Split/Prepare for training** are reused.
Since the Preprocessing step of the Final Pipeline is more involved than the one in the Baseline Pipeline, it has been split into two parts.

Preprocessing (I)

The input for the firstpreprocessing step is the output from the **Data Ingestion** metanode - and the output is a set of preprocessed documents.

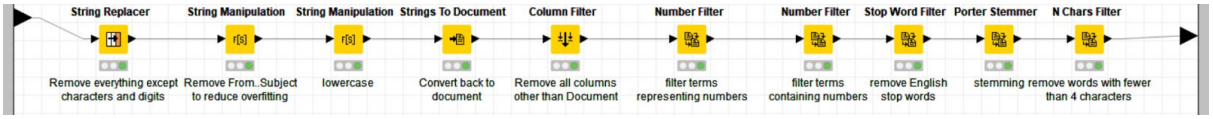

Figure 5: 1st preprocessing step of Final pipeline

Overall, this is what the flow does:

1) remove punctuation using a regular expression, same as Baseline preprocessing
2) remove From..Subject from the start of each document. This step is performed in order to **reduce overfitting** the model to the training set. For instance, if we have a particular person (email address) that has posted a lot in a particular newsgroup. If their email address is not removed from the document, it is likely that it will be given a higher weighting towards this particular category - which may cause their posts in different newsgroups to be misclassified by our predictive model.
3) Remove terms containing and representing numbers
4) Remove English stop words
5) Perform Porter stemming
6) Remove words with fewer than 4 characters

In the end, the document is stripped to its essentials and mostly contains terms with high predictive potential. Irrelevant terms are further weeded out in the next preprocessing step.

Preprocessing (II)

The output of Preprocessing (II) is a set of documents that is effectively stemmed lowercased words as numbers have been removed. The output of Preprocessing (II) is a Document Vector.
It is mostly due to the calculations and filters applied in this step that the Final workflow ends up with a higher prediction accuracy.

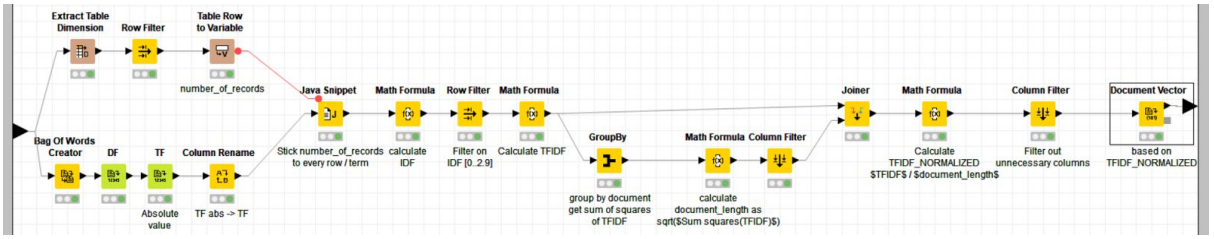

Figure 6: 2nd preprocessing step of Final pipeline

Let's consider this workflow on a component basis, left to right.
The leftmost part is responsible for several things. The bottom part calculates a Bag of Words based on the documents and it then goes on to calculate Document Frequency (DF) and absolute Term Frequency (TF) of those terms. The top part gets the number of documents (6,972) and appends it as a **number_of_documents** column to every row. Then, having DF, TF and (total) number_of_documents for every term, we are able to calculate the Inverse Document Frequency (IDF) in the middle section of the workflow using a **Math Formula** (after **Java Snippet**).
The formula is the standard one: IDF = log($number_of_records$ / DF)
Once we have DF, TF and IDF, we are able to filter out irrelevant terms (such that have low predictive value) based on the IDF value for the term. For this particular dataset, the best IDF range is [0..2.9]. This was established by looking at an IDF histogram and running the model multiple times and comparing results. The abovementioned histogram, along with a number of nodes that produced word clouds for the different document classes, have been stripped out of the final workflow in an effort to improve clarity and simplify the workflow. However, a lot of exploratory analysis has been done in order to come up with the optimal values for the filters and other preprocessing steps.
Having filtered out irrelevant terms based on the IDF values, the flow then calculates the TFIDF for every term and normalizes it accordingly such that the sum of squares of TFIDF values for all terms in a document is equal to 1. The respective values of TFIDF_NORMALIZED are then used to build a **Document Vector** as the output of the Preprocessing (II) step which goes on to **Label/Split/Prepare for training** metanode.

Replicating the results

In order to replicate the results one needs to checkout the master branch from the https://github.com/deyandyankov/uor_csmbd16 repository into C:\git\uor_csmbd16 and import the csmbd16.knwf file into KNIME, provided that KNIME and KNIME Labs are already set up correctly. The project will run with little modification on UNIX-based operating systems - one only needs to adjust the paths to the datasets in the **List Folders** nodes in the **Data Ingestion** metanode. Moreover, one can still use the same workflows with a different number of document categories - given that the documents are placed correctly in a file/directory structure. This requires no modification on the KNIME workflow part. Finally, CPU and memory resources required for the SVM classifier - the project was ran on a workstation with 16GB RAM.

Evaluation

There is more than one way to classify documents. The approach taken here is one from traditional machine learning. The state of the art in document classification at the moment makes use of unsupervised learning and deep learning methods.
For example, a popular unsupervised learning approach would be to use Latent Dirichlet Allocation[1] to separate the documents into a *k* number of topics and then link those topics into their respective categories (classes) manually. An example of a deep learning approach would be to do similar preprocessing as used in this project, but then make use of a neural network for the classification of the documents given the resulting document vector. This approach is likely to improve the overall accuracy but it comes at the cost of making decisions such as what sort of neural network should be used and what values would be appropriate for its hyperparameters.
Moreover, different weighting techniques, other than TF-IDF, are also worth exploring. Different representations, like word embeddings (word2vec[2], GloVe[3], FastText[4]), are also likely to improve the obtained results.
Given more time, it would be a good learning opportunity to experiment with the techniques mentioned above and compare their results with the ones obtained with the current workflows.
On the positive side - the final software package that this report supplements is a good starting point and a good learning material for anyone who would like to try and build on it. A lot of time was spent on getting this as easy to replicate as possible.

Conclusion

The implemented KNIME workflow is able to separate the documents in the test set into their categories with a relatively high accuracy. The same workflow can be used off the shelf to classify other documents given that they are stored in a specific file/directory structure. As discussed in the Evaluation section, the overall classification accuracy is likely to be improved if particular methods are explored further.

References

1. David M. Blei, Andrew Y. Ng, Michael I. Latent Dirichlet Allocation. Journal of Machine Learning Research 3 (2003) 993-1022 [http://jmlr.org/papers/volume3/blei03a/blei03a.pdf]
2. Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, Jeffrey Dean. Distributed Representations of Words and Phrases and their Compositionality. Google Inc, 2013. [https://arxiv.org/abs/1310.4546]
3. Jeffrey Pennington, Richard Socher, Christopher D. Manning. GloVe: Global Vectors for Word Representation. Stanford University, 2014 [https://nlp.stanford.edu/pubs/glove.pdf]

4. Armand Joulin, Edouard Grave, Piotr Bojanowski, Tomas Mikolov. Bag of Tricks for Efficient Text Classification. Facebook AI Research, 2016. [https://arxiv.org/pdf/1607.01759.pdf]