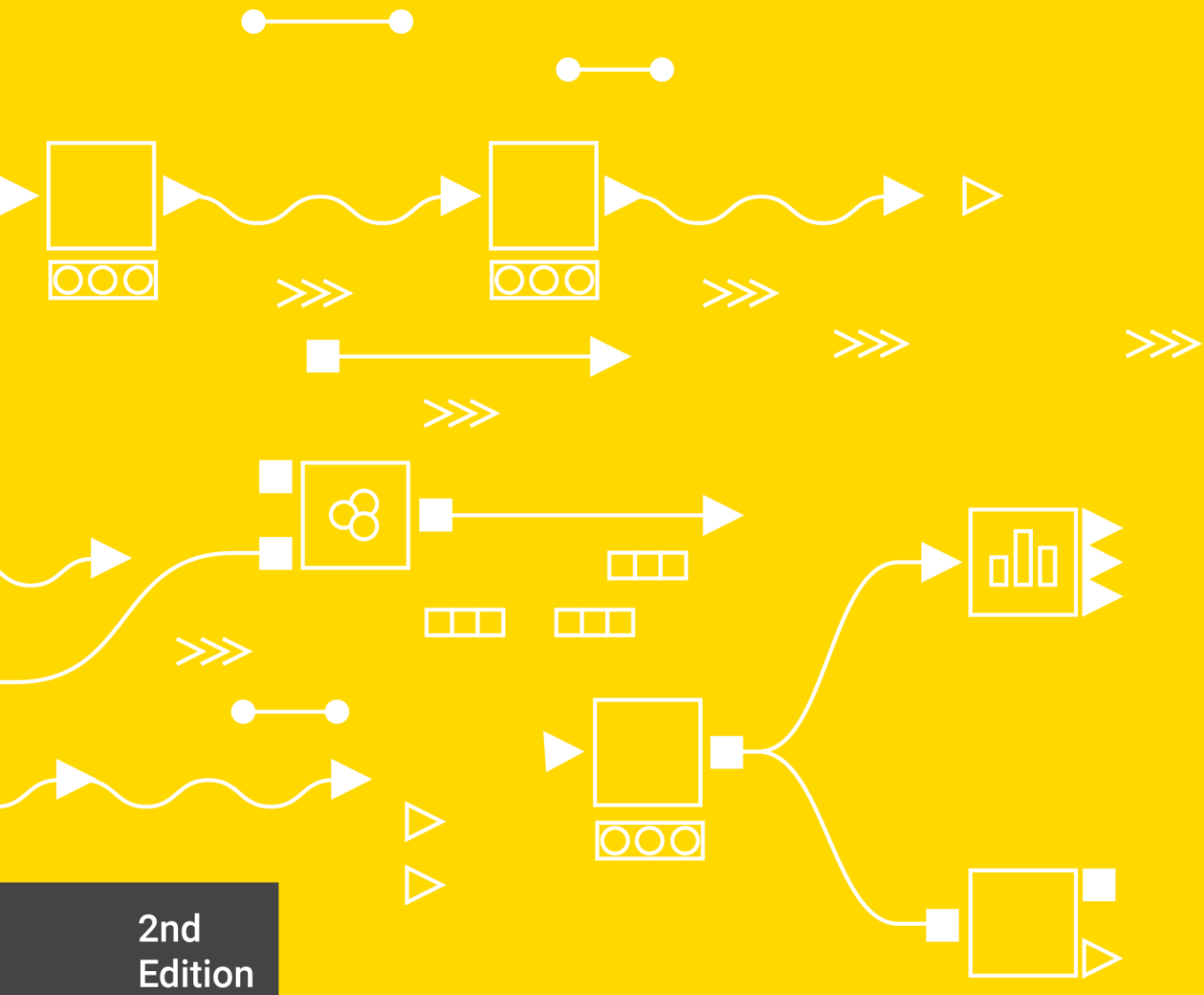
Open for Innovation ®
KNIME

Editor: Rosaria Silipo

A Collection of Case Studies

Practicing Data Science



2nd Edition

This book has been updated for **KNIME 3.7**.

For information regarding permissions and sales, write to:

KNIME Press
Technoparkstr. 1
8005 Zurich
Switzerland

knimepress@knime.com

Table of Contents

Introduction

We could start a philosophical discussion about what data science is. But that is not that kind of book. This book is a collection of experiences in data science projects.

There are many declinations of data science projects: with or without labeled data; stopping at data wrangling or involving machine learning algorithms; predicting classes or predicting numbers; with unevenly distributed classes, with binary classes, or even with no examples at all of one of the classes; with structured data and with unstructured data; using past samples or just remaining in the present; with real time or close to real-time time execution requirements or for acceptably slower performances; showing the results in shiny reports or hiding the nitty and gritty behind a neutral IT architecture; and last but not least with large budgets or no budget at all.

In the course of my professional life, I have seen many of such projects and their data science nuances. So much experience - and the inevitably related mistakes - should not be lost. Therefore, the idea of this book: a collection of data science case studies from past projects.

While the general development of a data science project is relatively standard, following, for example, the CRISP-DM cycle, each project often needs the cycle to be customized: that special ingredient added to adapt to the particular data, goals, constraints, domain knowledge, or even budget of the project.

This book is organized by application fields. We start with the oldest area in data science in chapter 1: the analysis of CRM data. We move on to retail stores with recommendation engines. And then discuss projects in the financial industry, about social media, time series analysis in IoT, and close finally with a number of cybersecurity projects.

All examples described in this book refer to a workflow (or two) which are available on the KNIME EXAMPLES server. These reference workflows are dutifully reported at the beginning of each section. Please notice that all example workflows have been simplified for these use cases. All optimizations, model comparisons, model selections, and other experiments, are not shown here, to better focus on the conclusive details of each project only.

We will update this book as frequently as possible with the descriptions and workflows from the newest, most recent data science projects, as they become available.

We hope this collection of data science experiences will help grow the practical data science skills in the next generation of data scientists.

Rosaria Silipo

Chapter 1. Customer Intelligence

1.1. Churn Prediction

By Rosaria Silipo
Access workflow on hub.knime.com
Or from: EXAMPLES/50_Applications/18_Churn_Prediction

Every company has CRM data. Even though a dataset from a CRM system is usually not that large, interesting applications can be developed using customer data from within the company. Churn prediction is one such application.

What You Need

To predict the likelihood of your current customers churning, you need data from previous customers with their churn history. Typically, customer information in your CRM system concerns demographics, behavioral data, and revenue information. At the time of renewing contracts, some customers did, and some did not, i.e. they churned. These example customers, both the ones who churned and the ones who did not, can be used to train a model to predict which of the current customers are at risk of churning.

The dataset we used for this example is available as a free download from the Ian Pardoe course and consists of two files. The first file includes contract data for 3333 telco customers and the second file includes operational data for the same customers. The contract data contains, among various attributes, a churn field: churn = 0 indicates a renewed contract; churn = 1 indicates a closed contract.

The Workflow to Train the Model

This is a binary classification problem. We want to predict which customer will churn (churn = 1) and which customer will not (churn = 0). That is:

attr 1, attr 2, …, attr n => churn (0/1)

After rejoining the two parts of the data, contractual and operational, converting the churn attribute to a string for the upcoming machine learning algorithm, and coloring data rows in red (churn=1) or blue (churn=0) for purely esthetical purposes, we trained a machine learning model to predict churn as 0 or 1 depending on all other customer attributes.

Which Model?

Which model shall we train? KNIME Analytics Platform offers a large variety of machine learning models to choose from. For this example, we trained a decision tree because of its appealing tree visualization. However, we could have used any other available machine learning algorithm for nominal class-like predictions, for example, Random Forests, Gradient Boosted Trees, or even deep learning. Given the small size of the dataset, though, we preferred not to overkill the application with an overly complex model.

Whatever machine learning algorithm you end up with, you always need to train it and evaluate (test) it. For this reason, the Partitioning node is required to split the data into one dataset for training and another one for testing. We chose a proportion with 80% training data vs. 20% test data.

The Decision Tree Learner node was fed with the training set (80% of the data). To train the decision tree (Decision Tree Learner node), we specified:

- the column with the class values to be learned (in this case: Churn)
- an information (quality) measure

- a pruning strategy (if any)
- the depth of the tree through the minimum number of records per node (higher number –> shallower tree)
- the split strategies for nominal and numerical values.

At the end of the training phase, the "View" option in the node context menu showed the decision path throughout the tree to reach the leaves with churning and non-churning customers. After training, the decision tree model was saved to a file in PMML format.

At this point, we needed to evaluate the model before running it on real data. For the evaluation, we used the test set (the remaining 20% of data) to feed a Decision Tree Predictor node. This node applies the model to all data rows one by one and predicts the likelihood of that customer to churn given his/her contractual and operational data (P(Churn=0/1)). Depending on the value of such probability, a predicted class will be assigned to the data row (Prediction (Churn) =0/1).

The number of times that the predicted class coincides with the original churn class is the basis for any measure of model quality as it is calculated by the Scorer node. The Scorer node offers a number of quality measures, like accuracy or Cohen's Kappa. Notice that the customers with churn=0 are many more than the customers with Churn=1. Thus, in this case, the Cohen's Kappa produces a more realistic measure of the model performance.

Notice also that the Scorer node – or any other scoring node – allows you to evaluate and compare different models. A subsequent Sorter node would allow you to select and retain only the best performing model.

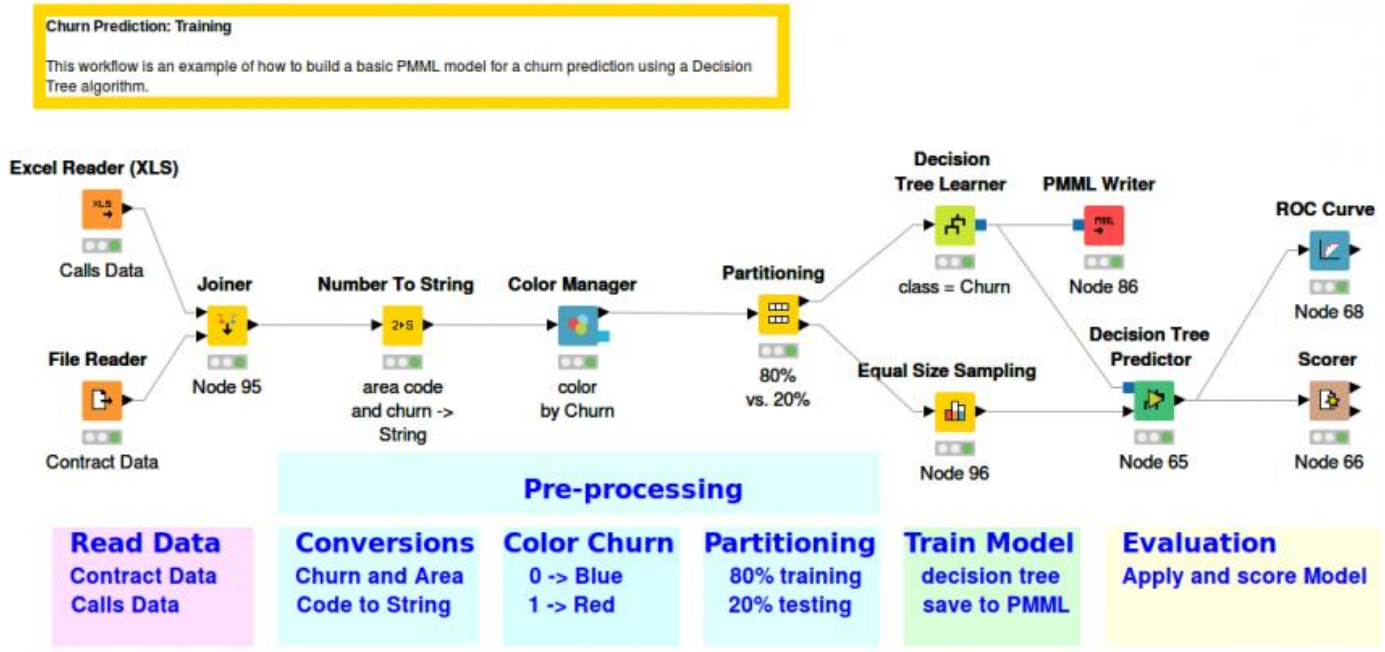


Figure 1. Training a Decision Tree model to predict the likelihood of customers to churn. Any other machine learning algorithm able to deal with binary classification can also be used instead of a decision tree.

Deployment

Once the model performances are accepted, we moved the model into production for deployment on real data. Here we need only to read the stream of real-life data coming in through a file or database i.e. the data source and apply the generated model.

We then applied a Decision Tree Predictor node to run the model on the real-life input data (Figure 2). Notice that as the model is structured in PMML format, we could have also used a PMML Predictor or a JPMML Classifier node. The output data will contain a few additional columns with the prediction class and the probability distributions for both classes, churn=0 and churn=1, (providing this is specified in the configuration settings of the predictor node).

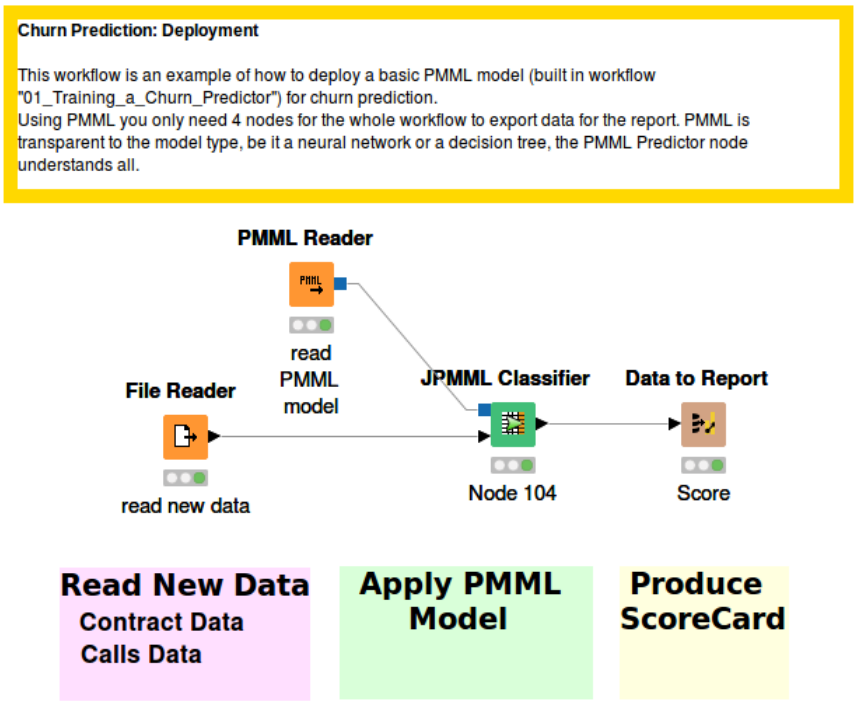


Figure 2. Applying the trained model to predict the likelihood of the current customer churning.

The last part of the production workflow is the result display. In our case, the input is data from one particular customer and the output is the likelihood that this customer is going to churn. We showed this via a speedometer chart in a BIRT report (Figure 3).

This likelihood score can be reassuring (below 40%), somewhat perplexing (above 40% and below 80%), or straightforward alarming (above 80%). Colors in the speedometer have been chosen accordingly to the alarm level. Different customer care actions have been devised for different likelihood scores.

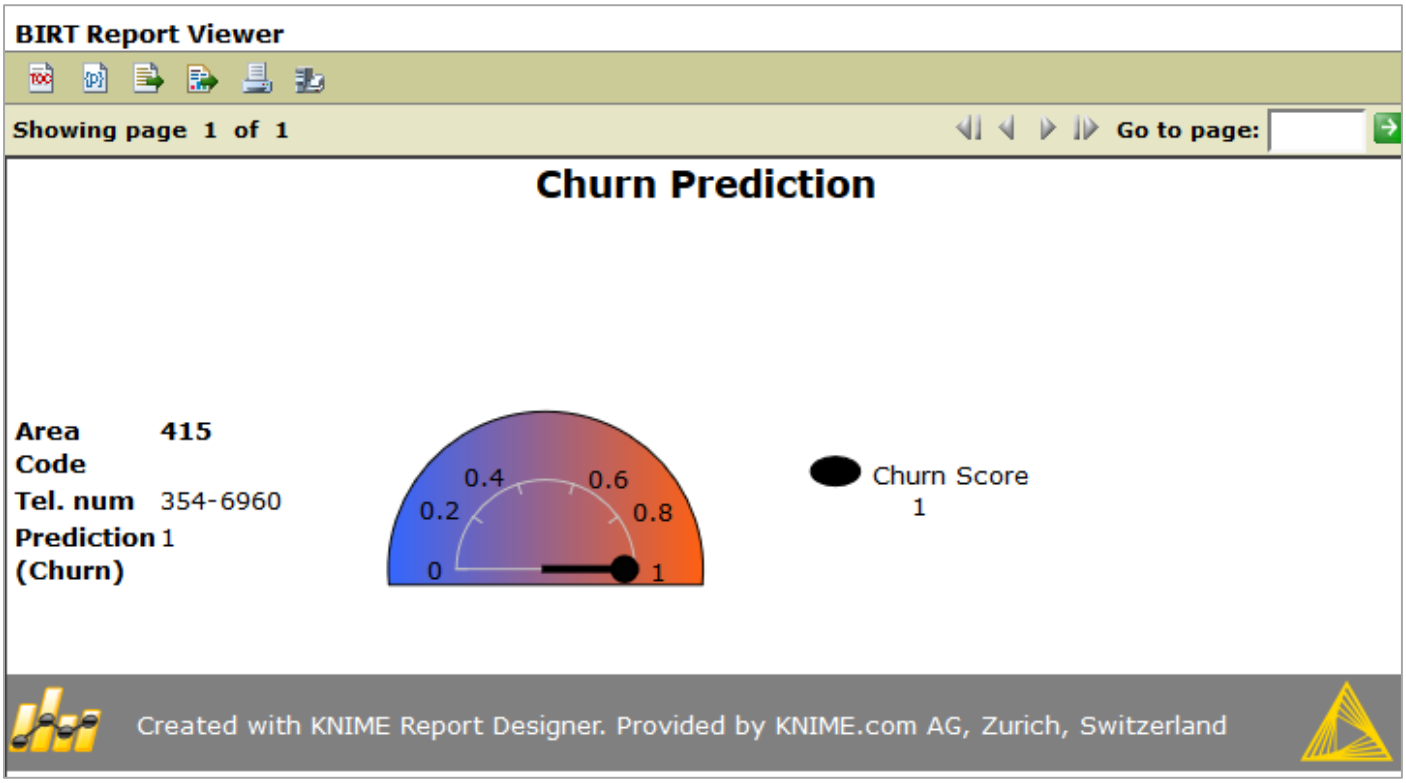


Figure 3. Displaying the likelihood to churn of the current customer via BIRT report.

1.2. Customer Segmentation

By Rosaria Silipo and Vincenzo Tursi
Access workflow on hub.knime.com
Or from: EXAMPLES/50_Applications/24_Customer_Segmentation_UseCase

Customer segmentation has undoubtedly been one of the most implemented applications in data analytics since the birth of customer intelligence and CRM data. The concept is simple. Group your customers together based on some attribute values, such as revenue creation, loyalty, demographics, buying behavior, etc …, or any combination of these.

Customer Segmentation Strategies

The group (or segment) definition can follow many strategies, depending on the degree of expertise and domain knowledge of the data scientist.

1. **Grouping by rules**. Somebody in the company already knows how the system works and how the customers tick. It is already known how to group them together with respect to a given task, like for example a campaign. A Rule Engine node would suffice to implement this set of experience-based rules. This approach is highly interpretable, but not very portable to new analysis fields. In the event of a new goal, new knowledge, or new data the whole rule system needs to be redesigned.

2. **Grouping as binning**. Sometimes the goal is clear and not negotiable. One of the many features describing our customers is selected as the representative one, be it revenues, loyalty, demographics, or anything else. In this case, the operation of segmenting the customers in groups is reduced to a pure binning operation. Here customer segments are built along one or more attributes by means of bins. This task can be implemented easily, using one of the many binner nodes available in KNIME Analytics Platform.

3. **Grouping with no knowledge**. It is often safe to assume that the data scientist does not know enough of the business at hand to build his/her own customer segmentation rules. In this case, if no business analyst is around to help, the data scientist should resolve to a plain blind clustering procedure. The subsequent work of cluster interpretation can be done by the business analyst, who is (or should be) the domain expert.

With this goal in mind, of making this workflow suitable for a number of different use cases, we chose the third option.

There are many clustering procedures, which you can find in KNIME Analytics Platform in the category Analytics/Mining/Clustering of the Node Repository panel: e.g. k-Means, nearest neighbors, DBSCAN, hierarchical clustering, SOTA, etc … We went for the most commonly used: the k-Means algorithm.

What You Need

To cluster your current customers into different groups, you need data describing past and present customers. You can see your customers from many points of view: demographics, money flow, shopping behavior, loyalty, number of contracts, purchased products, and probably even more that are more strictly related to your business.

Often raw data do not describe all those customer perspectives. Usually, a pre-processing phase is needed to aggregate and transform the data to move from a number of contracts, for example, to the money flow or to the loyalty score, from a weblog session to the clickstream history, etc … We will not describe these aggregation procedures here, since they can be quite complex and often specific to the particular business. We will assume that our data adequately describe some aspects of our customers.

For more details on preparing customer data, you can check the following posts from the “Data Chef ETL Battles” series published in the KNIME blog:

- "Customer Transactions. Money vs. Loyalty", 2017
- "Energy Consumption Time Series. Behavioral Measures over Time and Seasonality Index from Auto-Correlation", 2017
- "A social Forum. Sentiment vs Influence", 2018

The dataset we used for this example is available as a free download from the Ian Pardoe course and consists of two files. The first file includes contract data for 3333 telco customers and the second file includes operational data for the same customers.

Basic Workflow for Customer Segmentation with Clustering Procedure

The basic workflow for customer segmentation consists of only three steps: data reading, data pre-processing, and k-Means clustering. The clustering procedure - in this case the k-Means algorithm - and its associated normalization / de-normalization transformations represent the segmentation engine; that is the intelligent part of this workflow.

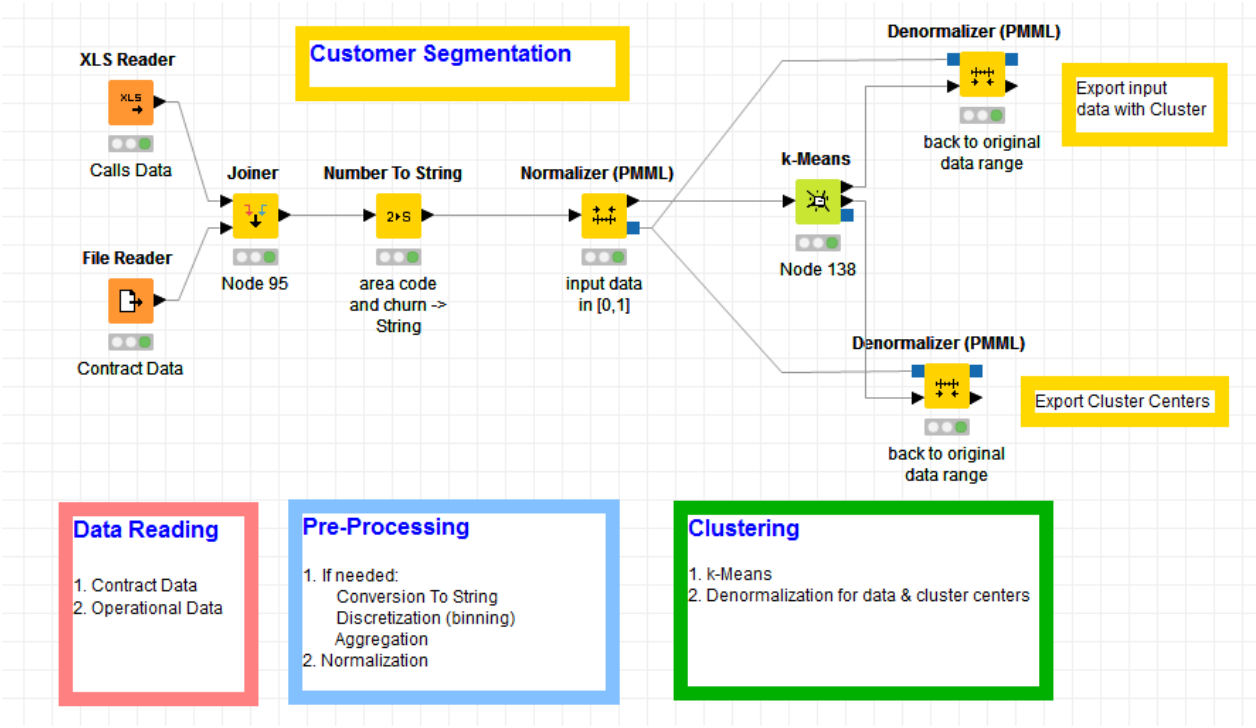


Figure 4. Basic Workflow for Customer Segmentation using k-Means as clustering technique.

Clustering could be replaced by a Rule Engine node (grouping by rules) or a Binner node (grouping as binning), if knowledge becomes available and we decide to change the segmentation strategy.

Model Deployment

The node to assign each data row to its cluster is the Cluster Assigner node. Here a distance is calculated between the input data row and all of the cluster centers. The cluster with the minimum distance is assigned to the input customer. Of course, the same pre-processing as performed before running the k-Means workflow is also required in the deployment workflow.

Refining Customer Segments with Business Knowledge by Domain Experts

A desirable improvement of the previous segmentation workflow could consist of involving business analysts in the process. Modern business analysts have precious knowledge of the data acquisition process and of the business case. Allowing them to interact with the results of the segmentation is often beneficial.

The idea thus is to guide modern business analysts through all phases of the analysis not from within the workflow, but from a web browser.

In the second part of this project, we deployed the k-Means segmentation workflow on the KNIME WebPortal. In this phase a web-based visualization wizard is created by strategically adding a number of Quickform and Javascript nodes to the workflow.

Indeed, on the KNIME WebPortal, workflow execution hops from a wrapped metanode containing Quickform or Javascript nodes to the next, producing at each step a web-based Guided User Interface (GUI) wizard. It is then possible to organize full guidance of the analysis on a web browser: for example, step 1 selects appropriate values, step 2 inspects the plot and readjusts selected values, and so on.

For each step, it is possible to design the GUI through multiple User Interface (UI) items on a web page layout, such as dropdown menus, radio buttons, interactive plots, and more. These UI components, generated by Quickform and Javascript nodes, form a web page, if placed inside a wrapped metanode. The layout of the web page produced by the wrapped metanode is controlled through a matrix layout available via a button in the tool bar at the top of KNIME Analytics Platform's workbench. This button is active only when the wrapped metanode is open in the workflow editor.

The first GUI step of the web-based wizard requires the number of segments and the data columns to be used for the segmentation. Segments (clusters) are then created in the background.

In the second step, a summary of all segments is displayed in a scatter plot and proposed to business analysts for inspection. The scatter plot is interactive, and the business analysts can decide whether or not to change the coordinates for a new inspection perspective, remove outliers, or drill down on a group of points. However, already with only 4 clusters, the scatter plot becomes hard to interpret.

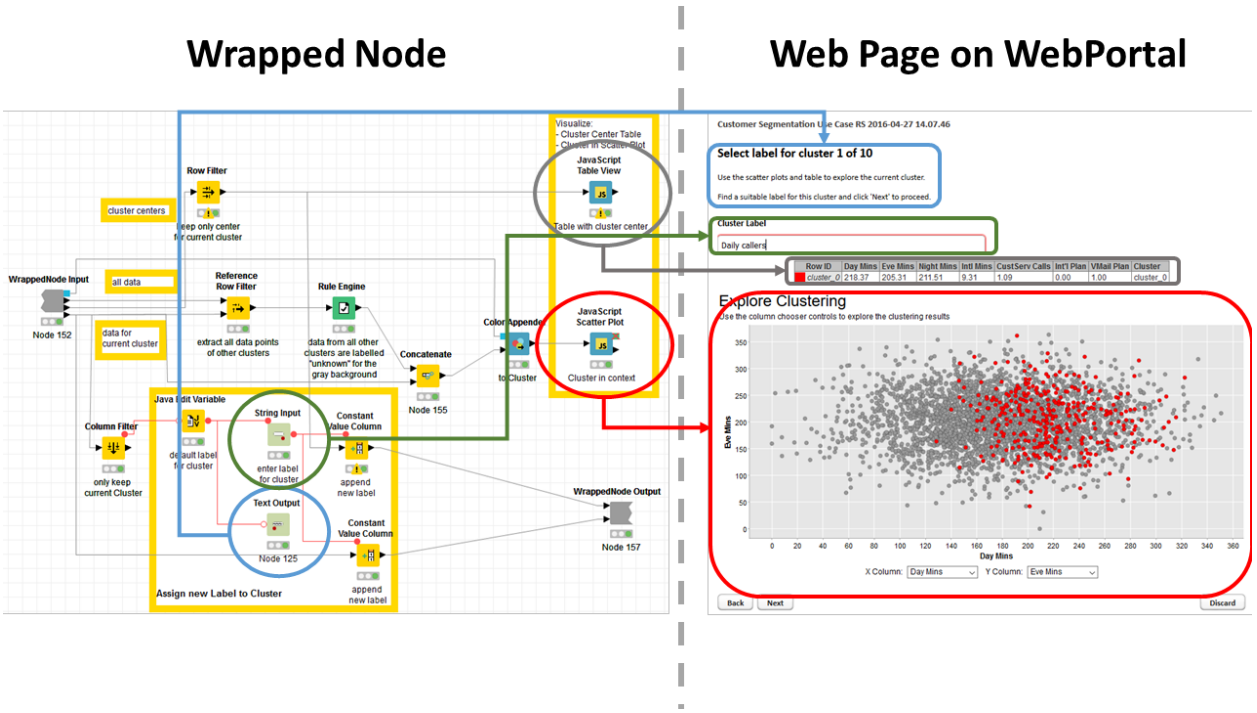


Figure 5. Quickform and Javascript nodes in a wrapped metanode on right produce the webpage on left when running on KNIME WebPortal.

So, in the next steps the segments are displayed one by one. The data are taken through a loop where at each iteration the data points of one cluster are displayed in color against all other points displayed in gray. At each iteration, the wizard web page reports the scatter plot with the cluster, the table with the cluster centers, and a textbox to enter free text. The textbox allows the business analysts to appropriately label or annotate – for example with calls to action regarding the customer segment under scrutiny.

The complete workflow is shown in the figure below (Figure 6)Figure 6. Final Workflow derived from the basic workflow for customer segmentation. The loop on the right goes through all clusters one by one. The wrapped metanode, Label Cluster, produces the web page shown in the previous figure. The whole sequence allows the business analysts to annotate and label the customer segments one by one., with the k-Means based segmentation part on the left and the visualization-interaction loop on the right. The content of the wrapped metanode, Label Cluster, and its corresponding web page is shown in the figure above (Figure 5).

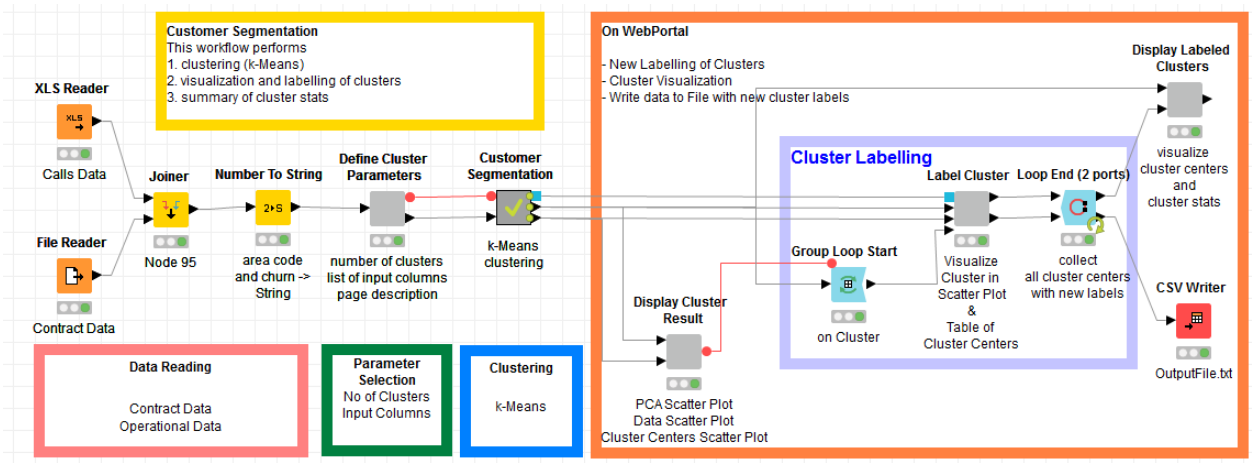


Figure 6. Final Workflow derived from the basic workflow for customer segmentation. The loop on the right goes through all clusters one by one. The wrapped metanode, Label Cluster, produces the web page shown in the previous figure. The whole sequence allows the business analysts to annotate and label the customer segments one by one.

This web-based analytics approach brings together the machine learning background of data analysts and the business domain knowledge of modern business analysts.

The whole project is described in detail in our most recent whitepaper, named “Customer Segmentation Conveniently from a Web Browser. Combining Data Science and Business Expertise” and downloadable from https://www.knime.org/files/white-papers/customer_segmentation.pdf.

The two workflows described in this post – for basic customer segmentation and for Web GUI guided customer segmentation - can be downloaded from the KNIME EXAMPLES server under *50_Applications/24_Customer_Segmentation_UseCase.*

Chapter 2. Retail and Supply Chain

2.1. Market Basket Analysis with the Apriori Algorithm

By Rosaria Silipo

Access workflow on hub.knime.com

Or from: EXAMPLES/50_Applications/16_MarketBasketAnalysis

A market basket analysis or recommendation engine [1] is what is behind the recommendations we get when we go shopping online or receive targeted advertising. The underlying engine collects information about people's habits and knows that, for example, if people buy pasta and wine, they are usually also interested in pasta sauces. In this section we will build an engine for market basket analysis using one of the many available association rule algorithms.

What You Need

The dataset required needs examples of past shopping baskets with shopping items (products) in it. If products are identified via product IDs, then a shopping basket is a series of product IDs, something like:

<Prod_ID1, Prod_ID2, Prod_ID3, ….>

The dataset used here was artificially generated with the workflow available in the KNIME EXAMPLES Server under *01_Data_Access/03_Data_Generation/09_Generating_a_Shopping_Market_Data_Set*. This workflow generates a set of Gaussian random basket IDs and fills them with a set of Gaussian random product IDs. After performing a few adjustments on the a priori probabilities, our artificial shopping basket dataset is ready to use [2].

This dataset consists of two KNIME tables: one containing the transaction data - i.e. the sequences of product IDs in imaginary baskets - and one containing product info – i.e. product ID, name, and price. The sequence of product IDs (the basket) is output as a string value, concatenating many substrings (the product IDs).

Workflow to Build Association Rules with the Apriori Algorithm

A typical analysis goal when applying market basket analysis, is to produce a set of association rules in the following form: *IF {pasta, wine, garlic} THEN pasta-sauce*

The first part of the rule is known as the "antecedent", and the second part, "consequent". A few measures, such as support, confidence, and lift, define how reliable each rule is. The most famous algorithm generating these rules is the Apriori algorithm [3].

The central part in building a recommendation engine is the Association Rule Learner node, which implements the Apriori algorithm in either the traditional [3] or the Borgelt [4] version. The Borgelt implementation offers a few performance improvements over the traditional algorithm. The produced association rule set, however, remains the same. Both Association Rule Learner nodes work on a collection of product IDs.

A collection is a particular data cell type, assembling together data cells. There are many ways of producing a collection data cell from other data cells [5]. The Cell Splitter node for example generates collection type columns, when the configuration setting "as set (remove duplicates)" is enabled. We used a Cell Splitter node to split the basket strings into product IDs substrings, setting the space as the delimiter character. The product IDs substrings were then assembled together and output in a collection column to feed the Association Rule Learner node.

After running on a dataset with past shopping basket examples, the Association Rule Learner node produces a number of rules. Each rule includes a collection of product IDs as antecedent, one product ID as consequent, and a few quality measures, such as *support*, *confidence*, and *lift*.

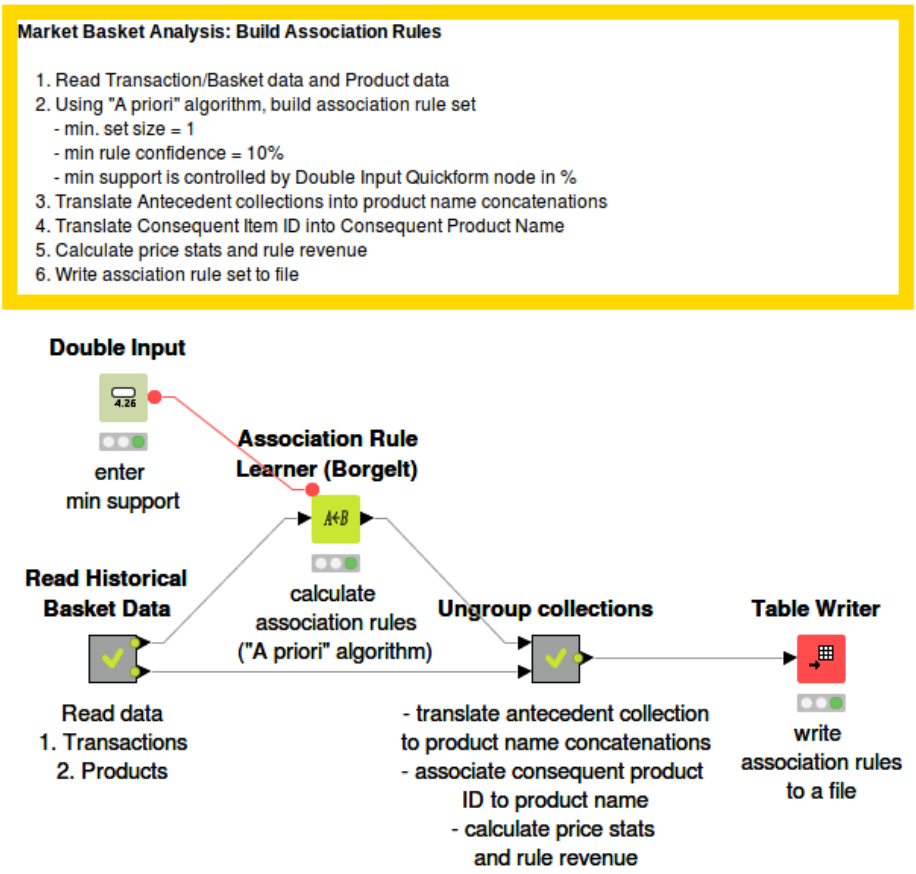


Figure 7. Workflow to train association rules using the Borgelt's variation of the Apriori algorithm.

In Borgelt's implementation of the Apriori algorithm, three support measures are available for each rule. If *A* is the antecedent and *C* is the consequent, then:

Body Set Support = support(*A*) = # items/transactions containing *A*

Head Set Support = support(*C*) = # items/transactions containing *C*

Item Set Support = support ($A \cup C$) = # items/transactions containing both antecedent and consequent

Item Set Support tells us how often antecedent *A* and consequent *C* are found together in an item set in the whole dataset. However, the same antecedent can produce a number of different consequents. So, another measure of the rule quality is how often antecedent *A* produces consequent *C* among all possible consequents. This is the rule confidence.

Rule Confidence = support($A \cup C$) /support(*A*)

One more quality measure – the rule lift – tells us how precise this rule is, compared to just the Apriori probability of consequent *C*.

Rule Lift = Rule Confidence(A → C) / RuleConfidence(∅ → C)

∅ is the whole dataset and support(∅) is the number of items/transactions in the dataset.

You can make your association rule engine larger or smaller, restrictive or tolerant, by changing a few threshold values in the Association Rule Learner configuration settings, like the "minimum set size", the "minimum rule confidence", and the "minimum support" referring to the minimum Item Set Support value.

We also associated a potential revenue to each rule as:

Revenue = price of consequent product x rule item set support

Based on this set of association rules, we can say that if a customer buys wine, pasta, and garlic, (antecedent) usually – or as usually as support says – he/she also buys pasta-sauce (consequent); we can trust this statement with the confidence percentage that comes with the rule.

After some preprocessing to add the product names and prices to the plain product IDs, the association rule engine with its antecedents and consequents is saved in the form of a KNIME table file.

Deployment

Let's move away now from the dataset with past examples of shopping baskets and into real life. Customer X enters the shop and buys pasta and wine. Are there any other products we can recommend?

The second workflow prepared for this use case takes a real-life customer basket and looks for the closest antecedent among all antecedents in the association rule set. The central node of this workflow then is the Subset Matcher node.

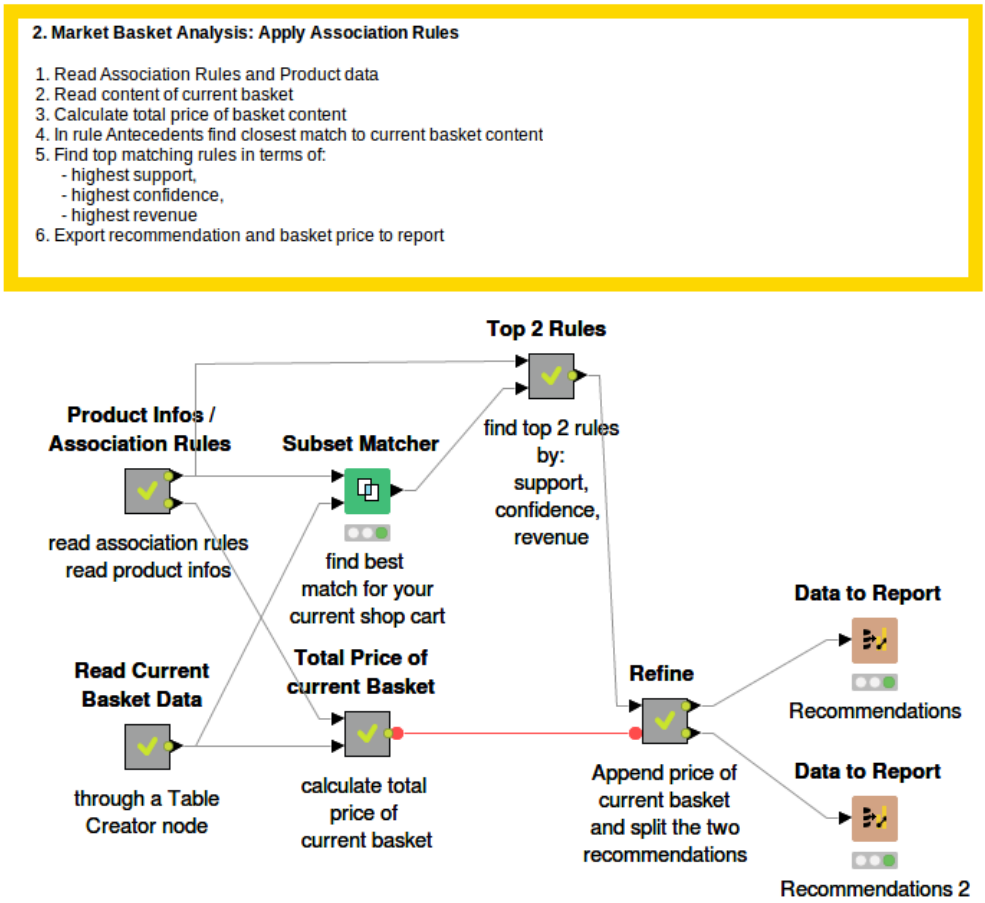


Figure 8. Extracting top recommended items for current basket. Here the subset matcher explores all rule antecedents to find the appropriate match with the current basket items.

The Subset Matcher node takes two collection columns as input: the antecedents in the rule set (top input port) and the content of the current shopping basket (lower input port). It then matches the current basket item set with all possible subsets in the rule antecedent item sets. The output table contains pairs of matching cells: the current shopping basket and the totally or partially matching antecedents from the rule set.

By joining the matching antecedents with the rest of the corresponding association rule – that is with consequent, support, confidence, and lift – we obtain the products that could be recommended to customer X, each one with its rule's confidence, support, revenue, and lift. Only the top 2 consequents, in terms of highest item set support (renamed as rule support), highest confidence, and highest revenue, are retained.

Finally, a short report displays the total price for the current basket and two recommendations, from the two top consequents.

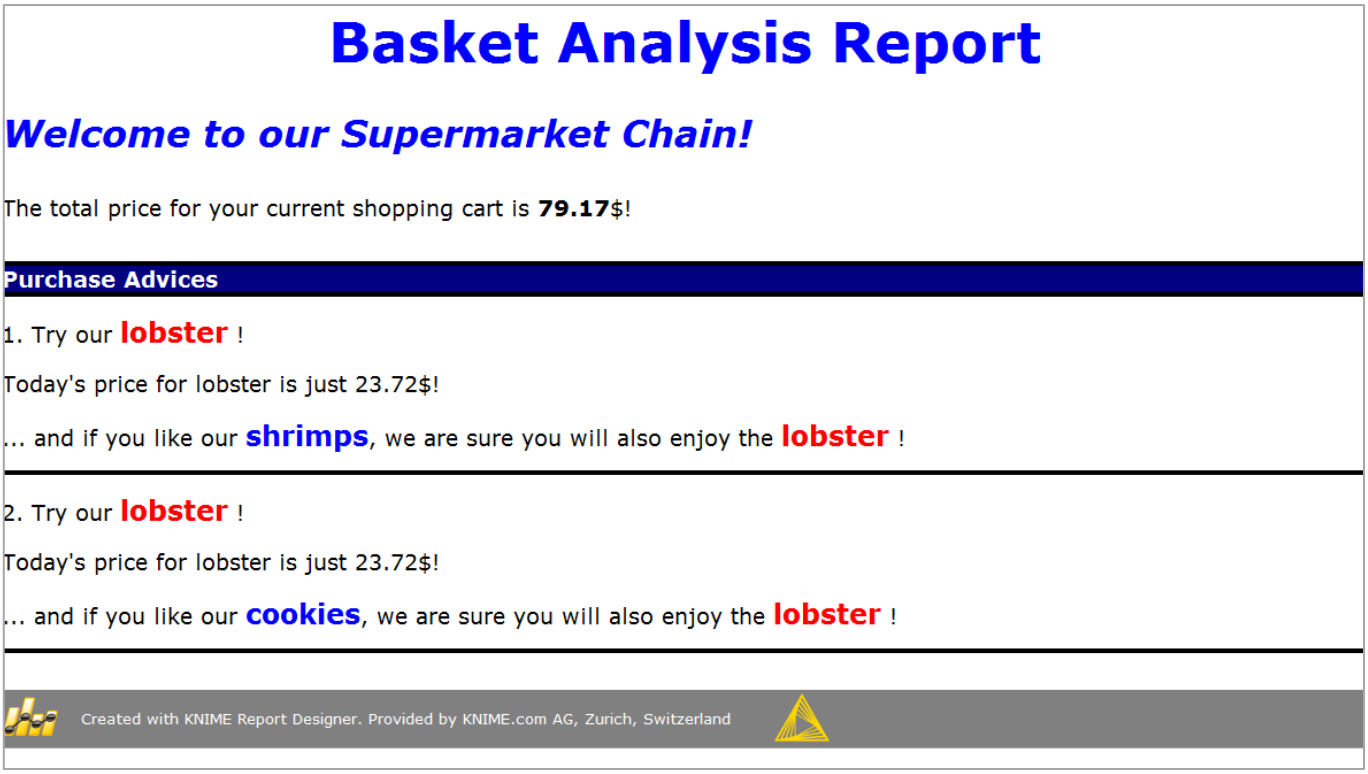


Figure 9. Final Recommendations on a Report.

In our case, the recommended product is always lobster, associated once with cookies and once with shrimps. While shrimps and lobster are typically common-sense advice, cookies and lobster seem to belong to a more hidden niche of food experts!

References

1. "Association Rule Learning", Wikipedia http://en.wikipedia.org/wiki/Association_rule_learning
2. I. Adä, M. Berthold, "The New Iris Data : Modular Data Generators", SIGKDD, 2010
3. R. Agrawal and R. Srikant, Proc. 20th Int. Conf. on Very Large Databases (VLDB 1994, Santiago de Chile), 487-499, Morgan Kaufmann, San Mateo, CA, USA 1994
4. "Find Frequent Item Sets and Association Rules with the Apriori Algorithm" C. Borgelt's home page http://www.borgelt.net/doc/apriori/apriori.html
5. "Collection Cookbook", Tobias Koetter, KNIME Blog, http://www.knime.org/blog/collection-cookbook

2.2. Movie Recommendations with Spark Collaborative Filtering

By Rosaria Silipo
Access workflow on hub.knime.com
Or from:
EXAMPLES/10_Big_Data/02_Spark_Executor/10_Recommendation_Engine_w_Spark_Collaborative_Filtering

Collaborative Filtering (CF) [1] based on the Alternating Least Squares (ALS) technique [2] is another algorithm used to generate recommendations. Collaborative Filtering (CF) is an algorithm that makes automatic predictions (*filtering*) about the interests of a user by collecting preferences from many other users (*collaborating*). The underlying assumption of the collaborative filtering approach is that if a person *A* has the same opinion as person *B* on an issue, A is more likely to have B's opinion on a different issue than that of a randomly chosen person. This algorithm gained a lot of traction in the data science community after it was used by the team that won the Netflix prize.

The algorithm has also been implemented in Spark MLlib [3] with the aim to address fast execution also on very large datasets. KNIME Analytics Platform with its Big Data Extensions offers the CF algorithm in the Spark Collaborative Filtering node. We will use it, in this section, to recommend movies to a new user. This use case is a KNIME implementation of the Collaborative Filtering solution provided in the Infofarm blog post referred in [4].

What You Need

A general dataset with movie ratings by users

For this use case, we used the large MovieLens dataset. This dataset contains many different files all related to movies and movie ratings. We'll use the files ratings.csv and movies.csv.

The dataset in file *ratings.csv* contains 20M movie ratings by circa 130K users and it is organized as:

movieID, userID, rating, timestamp

Each row contains the rating to each movie – identified by movieID - by one of the users – identified by userID.

The dataset in file *movies.csv* contains circa 27K movies, organized as:

movieID, title, genre

Movie preferences by current user

The idea of the ALS algorithm is to find other users in the training set with preferences similar to the current, selected user. Recommendations for the current user are then created based on the preferences of such similar profiles. This means that we need a profile for the current user to match the profiles of other existing users in the training set.

Let's suppose that you are the current user, with assigned userID=9999. It is likely that the Movielens dataset has no data about your movie preferences. Thus, in order to issue some movie recommendations, we would first need to build your movie preference profile. So, we will start the workflow by asking you to rate 20 movies, randomly extracted from the movie list in movies.csv file. Rating ranges between 0 and 5 (0 – horrible movie; 5 – fantastic movie). You can use rating -1, in case you have not seen the proposed movie. Movies with rating -1 will be removed from the list. Movies with not -1 ratings will become training set material.

The web page below is the result of a Text Output node and a Table Editor (JavaScript) node inside a wrapped metanode executed on KNIME WebPortal. Your rating can be manually inserted in the last column to the right.

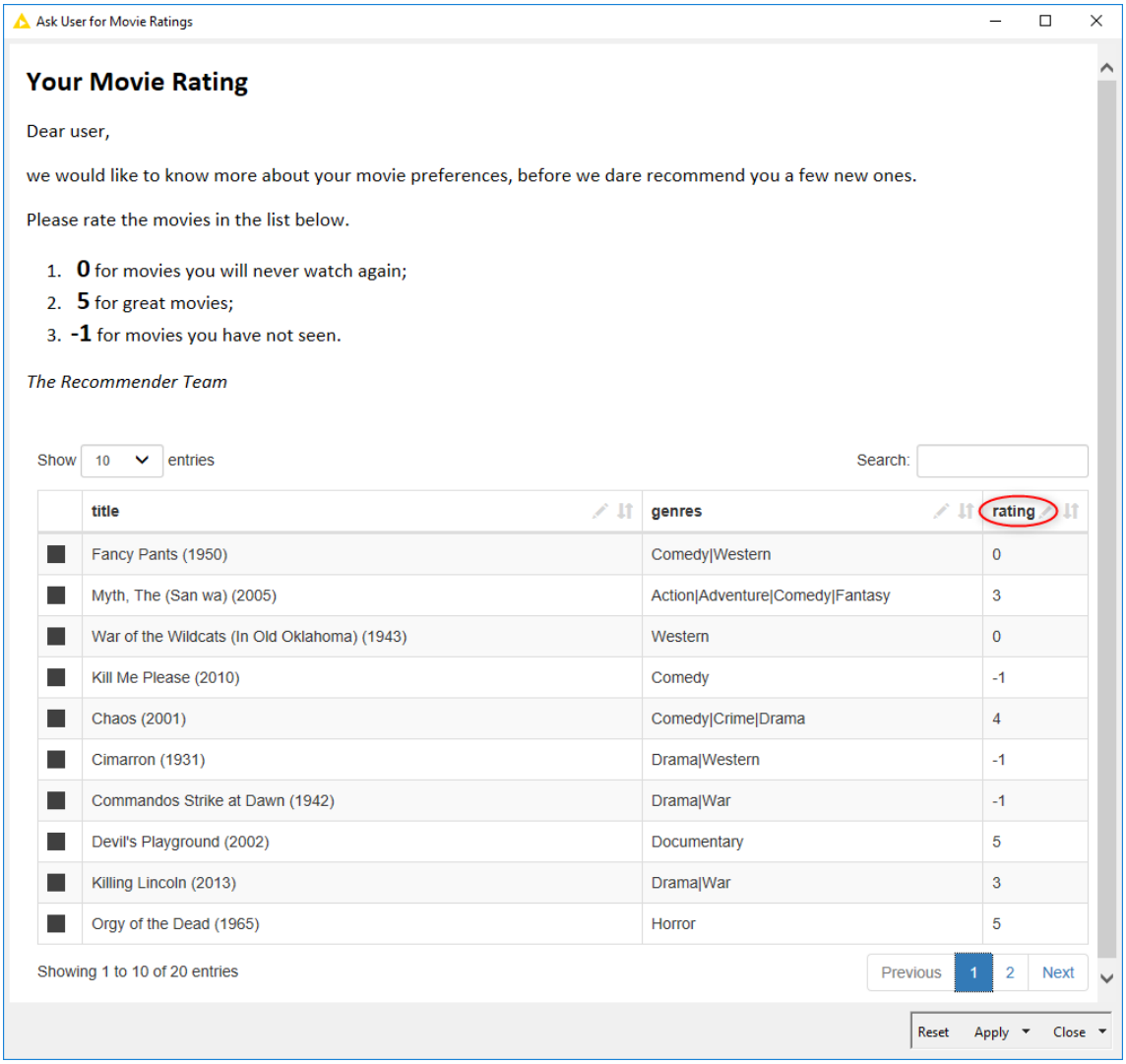


Figure 10. Interviewing the current user (userID =9999) about his/her movie ratings. We need this information to create the current user profile and to match it with profiles of other users available in the training set. Preferences from similar users can provide recommendations for our current user.

A Spark Context

The CF - ALS algorithm has been implemented in KNIME Analytics Platform via the Spark Collaborative Filtering Learner node. This node belongs to the KNIME Big Data extensions, which need to be installed on your KNIME Analytics Platform to run this use case (see instructions on how to install a KNIME extension).

The Spark Collaborative Filtering Learner node executes within a Spark context; which means that you also need a big data platform and a Spark context to run this use case. This is usually a show stopper, due to the difficulty and potential cost of installing a big data platform, especially if the project is just a proof of concept. Indeed, installing a big data platform is a complex operation and might not be worth it for just a prototype workflow. Installing it on the cloud might also carry additional unforeseeable costs.

Note. Version 3.6 of KNIME Analytics Platform and KNIME Big Data extension includes a new precious node: the "Create Local Big Data Environment" node. . This node creates a simple but complete local big data environment with Apache Spark, Apache Hive and Apache HDFS, and does not require any further software installation.

Figure 11. The new node Create Local Big Data Environment available in KNIME Analytics Platform 3.6 This node creates a local simple instance of Spark, Hive, and HDFS. While it may not provide the desired scalability and performance, it is useful for prototyping and offline development.

The Create Local Big Data Environment node has no input port, since it needs no input data, and produces three output objects:

- A red database port to connect to Hive
- A light blue HDFS connection port to connect to the HDFS underlying system
- A gray Spark port to connect to the Spark context

By default, the local Spark, Hive and HDFS instances will be disposed of when the Spark context is destroyed or when KNIME Analytics Platform closes. In this case, even if the workflow has been saved with the "executed" status, intermediate results of the Spark nodes will be lost.

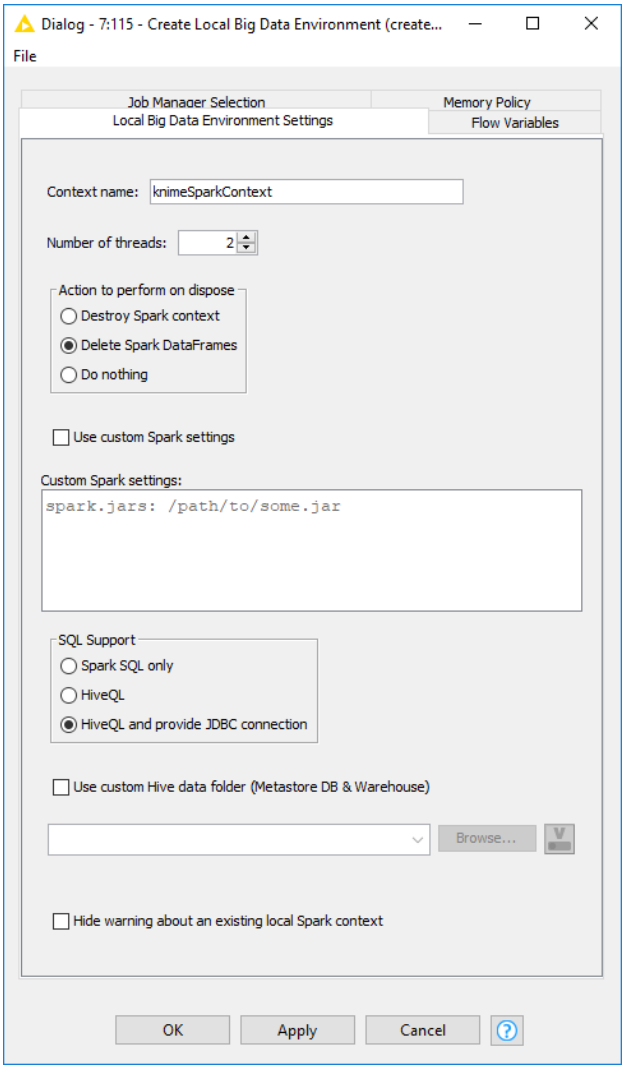


Figure 12. Configuration window of the node, Create Local Big Data Environment. Settings involve: actions to perform "on dispose"; custom Spark settings; SQL support; Hive custom folder; warning options.

The configuration window of the Create Local Big Data Environment node includes a frame with options related to the "on dispose" action.

- Destroy Spark Context destroys the Spark context and all allocated resources;–this is the most destructive, but cleanest, option.
- Delete SparkData Frames deletes the intermediate results of the Spark nodes in the workflow, but keeps the Spark context open to be reused.
- Do Nothing keeps both the Spark DataFrames and context alive. If you save the already executed workflow and reopen it later, you can still access the intermediate results of the Spark nodes within. This is the most conservative option, but also keeps space and memory busy on the execution machine.

Option number 2 is set as default, as a compromise between resource consumption and reuse.

Workflow to Build the Recommendation Engine with Collaborative Filtering

In this workflow we use the Spark MLlib implementation of the collaborative filtering algorithm, in which users and products are described by a small set of latent factors. These latent factors can then be used to predict the missing entries in the dataset. Spark MLlib uses the Alternating Least Squares (ALS) algorithm for the matrix factorization, to learn the latent factors.

Note. It is necessary that movie preferences of the current user are part of the training set. This is why we ask the current user to rate 20 random movies, in order to get a sample of his/her preferences.

The Collaborative Filtering technique is implemented and trained in the Spark Collaborative Filtering Learner node, which runs on a Spark cluster. At its input port, the node receives a number of records with product, user, and corresponding rating. At the output port, it produces the recommendation model and the predicted ratings for all input data rows, including user and object.

Note. The matrix factorization model output by the node contains references to the Spark DataFrames/RDDs used in execution and thus is not self-contained. The referenced Spark DataFrames/RDDs are deleted, like for any other Spark nodes, when the node is reset or the workflow is closed. Therefore, the model cannot be reused in another context in another workflow.

Like the KNIME native Numeric Scorer node, the Spark Numeric Scorer node calculates a number of numeric error metrics between the original values – in this case the ratings – and the predicted values. Ratings range between 0 and 5, as number of stars assigned by a user to a movie. Predicted ratings try to predict the original ratings between 0 and 5.

The error metrics on the test set show a mean absolute error of 0.6 and a root mean squared error of 0.8. Basically, predicted ratings deviate from the original ratings +/- 0.6, which is close enough for our recommendation purpose.

The original movie rating dataset was split into a training set and a test set. The training set was used to build the recommendations with a Spark Collaborative Filtering Learner node and the test set to evaluate their quality with a generic Spark Predictor node followed by a Spark Numeric Scorer node.

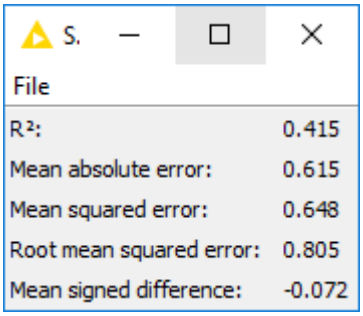


Figure 13. Numerical error metrics calculated on the original movie ratings and the predicted movie ratings with a Spark Numeric Scorer node.

Deployment

We previously asked the current user to rate 20 randomly chosen movies. These ratings were added to the training set. Using a generic Spark Predictor node, we now estimate the ratings of our current user (ID=9999) on all remaining unrated movies. Movies are then sorted by predicted ratings and the top 10 are recommended to the current user on a web page via KNIME WebPortal.

Since I volunteered to be the current user for this experiment, based on my ratings of 20 randomly selected movies, I got back a list of 10 recommended movies shown below. I haven't seen most of them; however, some of them I do know and appreciate. I will now add "watch recommended movies" on my list of things to do for the next month.

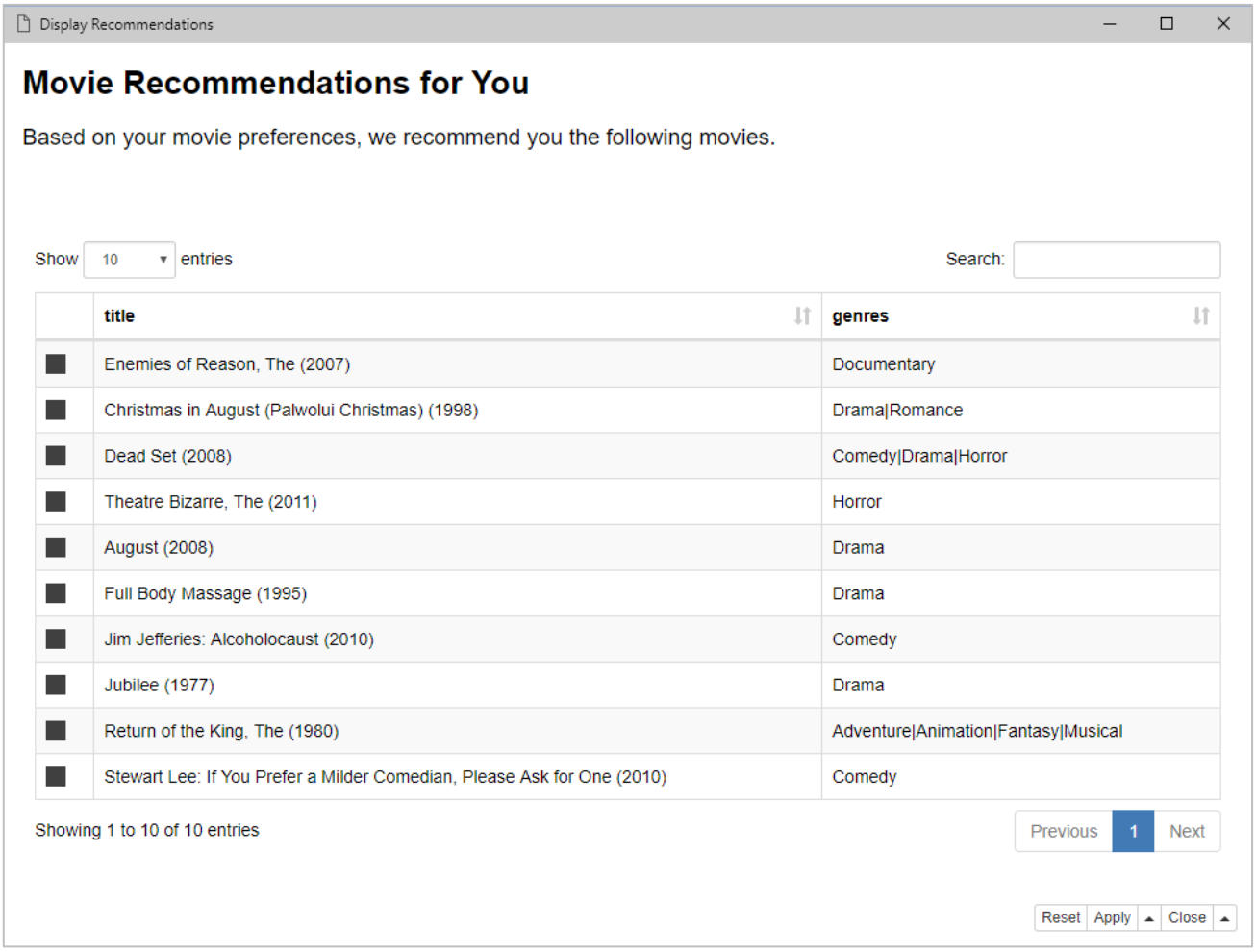


Figure 14. Final list of top 10 recommended movies, based on my earlier ratings to the 20 randomly selected movies.

Note. Please notice that this is one of the rare cases where training and deployment are included in the same workflow.

Indeed, the Collaborative Filtering model produced by the Spark Collaborative Filtering Learner node is not self-contained but depends on the Spark Data Frame/RDDs used during training execution and therefore cannot be reused later in a separate deployment workflow.

The Collaborative Filtering algorithm is not computationally heavy and does not take long to execute. So, including the training phase in the deployment workflow does not noticeably hinder recommendation performance. However, if recommendation performance is indeed a problem, the workflow could be partially executed on KNIME Analytics Platform or KNIME Server until the collaborative filtering model is trained and then the rest of the workflow can be executed on demand for each existing user in the training set.

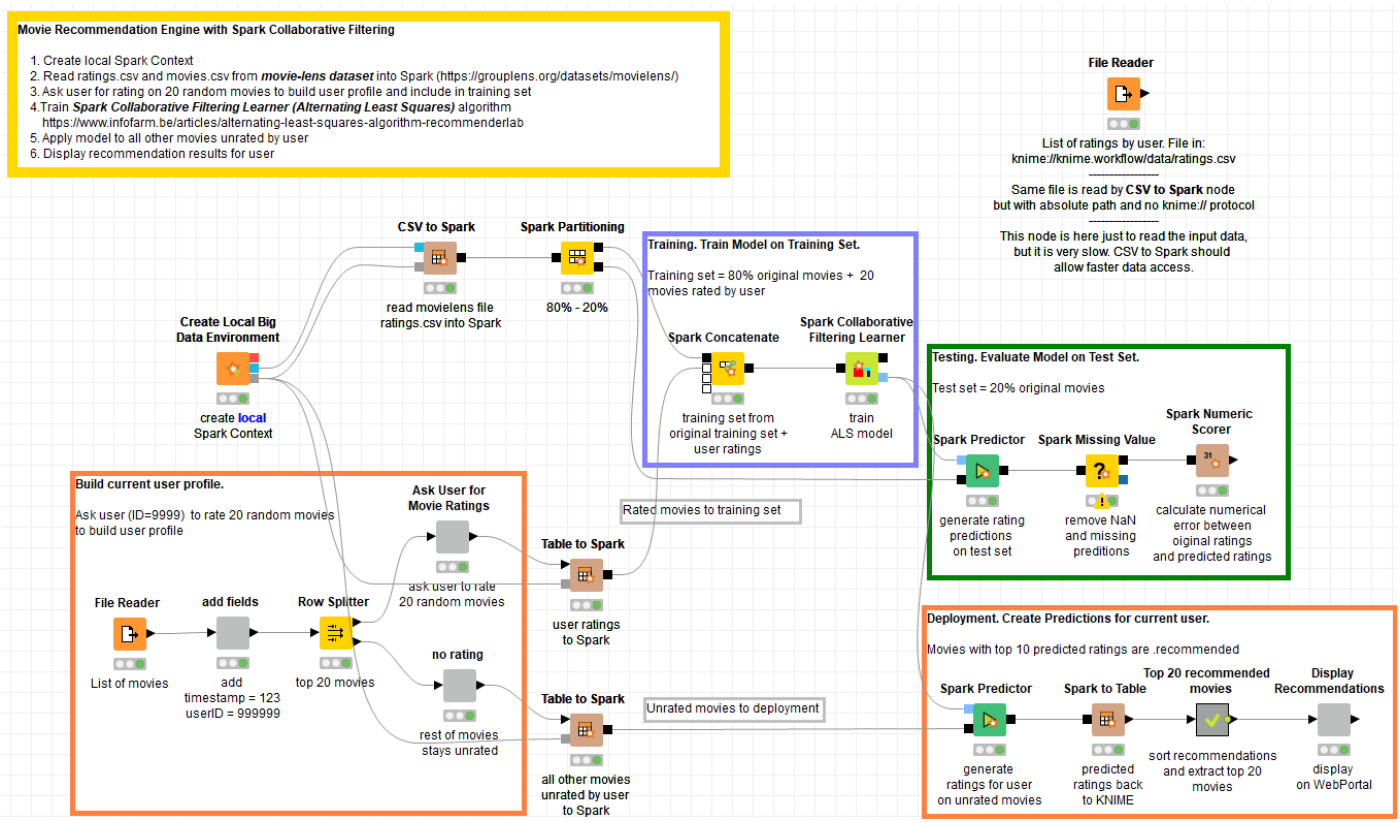


Figure 15. Workflow *EXAMPLES/10_Big_Data/02_Spark_Executor/10_Recommendation_Engine_w_Spark_Collaborative_Filtering* asks the new user to rate 20 randomly selected movies via web browser, with this data trains a Collaborative Filtering model, evaluates the model performance via some numeric error metric, and finally proposes a list of top 10 recommended movies based on the previously asked ratings.

References

1. "Collaborative Filtering", Wikipedia https://en.wikipedia.org/wiki/Collaborative_filtering
2. Y. Koren, R. Bell, C. Volinsky, "Matrix Factorization Techniques for Recommender Systems", in Computer Journal, Volume 42 Issue 8, August 2009, Pages 30-37 https://dl.acm.org/citation.cfm?id=1608614
3. "Collaborative Filtering. RDD based API" The Spark MLlib implementation http://spark.apache.org/docs/latest/mllib-collaborative-filtering.html
4. "The Alternating Least Square Algorithm in RecommenderLab" Infofarm blog post by Bregt Verreet Sep 16 2016 https://www.infofarm.be/articles/alternating-least-squares-algorithm-recommenderlab

Chapter 3. Banking and Insurance

3.1. Credit Risk Assessment

By Vincenzo Tursi and Rosaria Silipo
Access workflow on hub.knime.com
Or from: EXAMPLES/50_Applications/38_Credit_Risk_Assessment

When asking for a loan or signing up for an insurance, you trigger a complex process, where your data are collected mainly to estimate your likelihood to default.

The likelihood to default is a metric that lenders use to assess risk on customers, group them into similar risk bands, and determine the interest rate to charge. A lender will use probability of default to calculate expected return or loss at the individual and portfolio level. The process of estimating the probability of default of a customer is called Risk Scoring.

The process of Risk Scoring involves data collection about the customer, group definition, consultation of look up tables, some gut feeling from the lender's experience, and finally a number for the probability of default. Since the final result is a probability, we can automatize the process and probably improve the accuracy of the final result by turning to machine learning algorithms.

What You Need

In order to estimate the default probability of customers, first of all we need some data describing solvent and insolvent people when it comes to their loans.

For this use case we will use the "Give me some Credit" public dataset. This dataset describes around 150,000 individuals in terms of age, monthly income, past credit history, and past delinquency history. The dataset is not specifically labeled for risk score or default probability. However, the dataset contains a binary column named "SeriousDlqin2yrs" describing whether the person has been seriously delinquent in making payments for at least 90 days in the last 2 years. This is represented as delinquency (1) or not (0). This column can be used as the target column to imply insolvency by the customer.

Workflow to Predict Delinquency

The workflow used to predict default, i.e. serious delinquency in payment in the past 2 years, is reported below.

All input data are read and cleaned appropriately, missing values are imputed, and target column "SeriousDlqIn2Yrs" converted from 0/1 Integer to String. The machine learning model used is a Random Forest with 100 trees. The trained model is evaluated on a test set by a Scorer node. The whole process here is quite classic: data preprocessing, partitioning, model training, and evaluation.

There are a few steps in this workflow, though, that deserve more detailed description: the quite complex data pre-processing phase; the Linear Correlation as part of the data exploration phase; and the SMOTE algorithm.

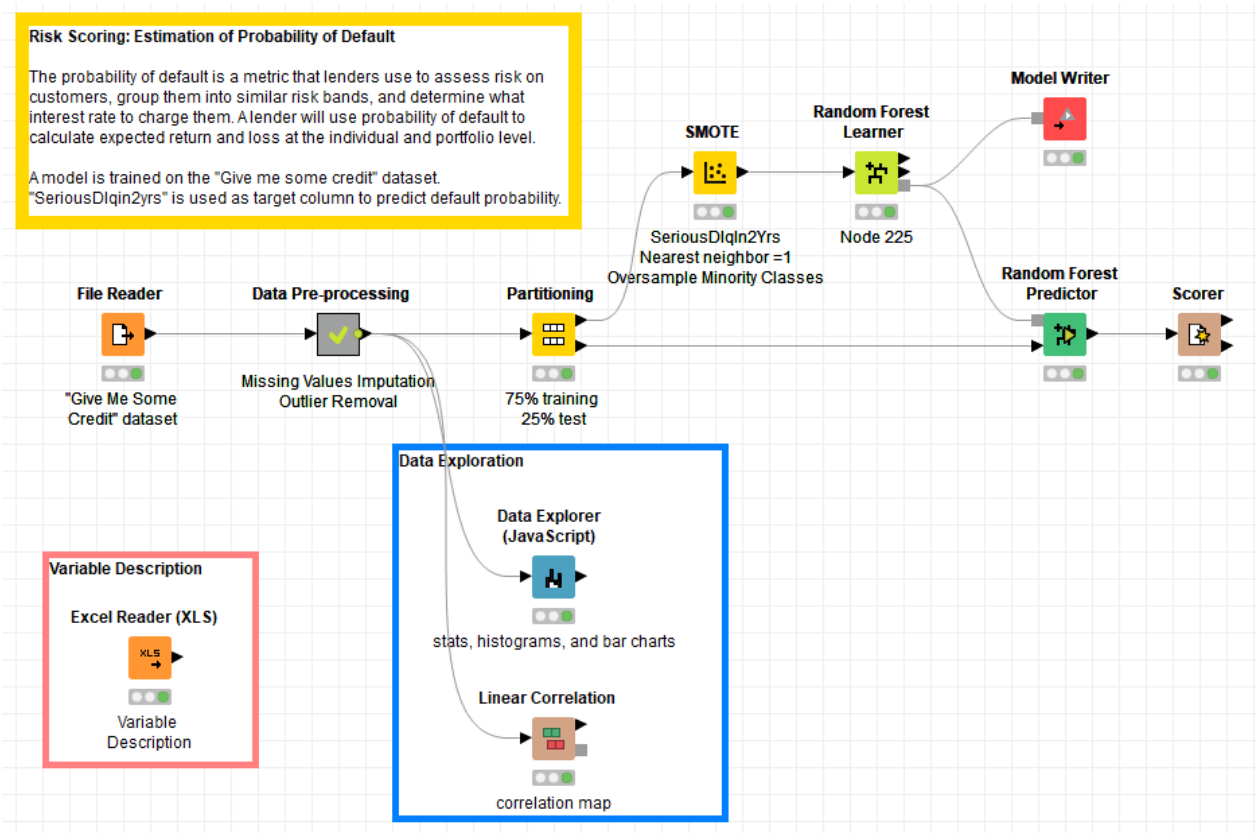


Figure 16. This workflow trains a random forest with 100 trees to predict the probability of delinquency of each applicant. The class imbalance requires under-sampling or over-sampling on the training set for better model performance. Here the over-sampling SMOTE procedure is applied. This workflow is available on the KNIME EXAMPLES server in *50_Applications/38_Credit_Risk_Assessment/Credit_Scoring_Training*.

Data Preprocessing

The Data Preprocessing metanode prepares the data for the upcoming machine learning algorithm. This mainly includes outlier removal and missing value imputation for all input columns.

Generally, missing values are imputed using a knowledge- or experience based fixed value or alternatively the median value of the column.

Outliers are removed from the columns “MonthlyIncome” (first and last quartiles) and “NumberOfOpenCreditLines AndLoans” (last two quintiles). This operation reduces the dataset from its original 150K rows to 85K rows. The outlier removal operation was a bit aggressive, but we wanted to make sure that all applicants with very low and very high income or with too many loans already on their shoulders were removed. We are interested in the loan application of the average person, where the decision is often not easy.

The selected target column to predict is “SeriousDlqin2yrs”, which is then converted from Integer to String. It is important to notice the unbalanced class distribution here in the dataset, since many more loan applications are accepted rather than rejected. This fact is even clearer when observing the bar chart of occurrences of the two classes, generated by the Data Explorer node in the “Visual Exploration” part of the workflow. Data rows in class 0 (no delinquency) are much more numerous than data rows in class 1 (serious delinquency).

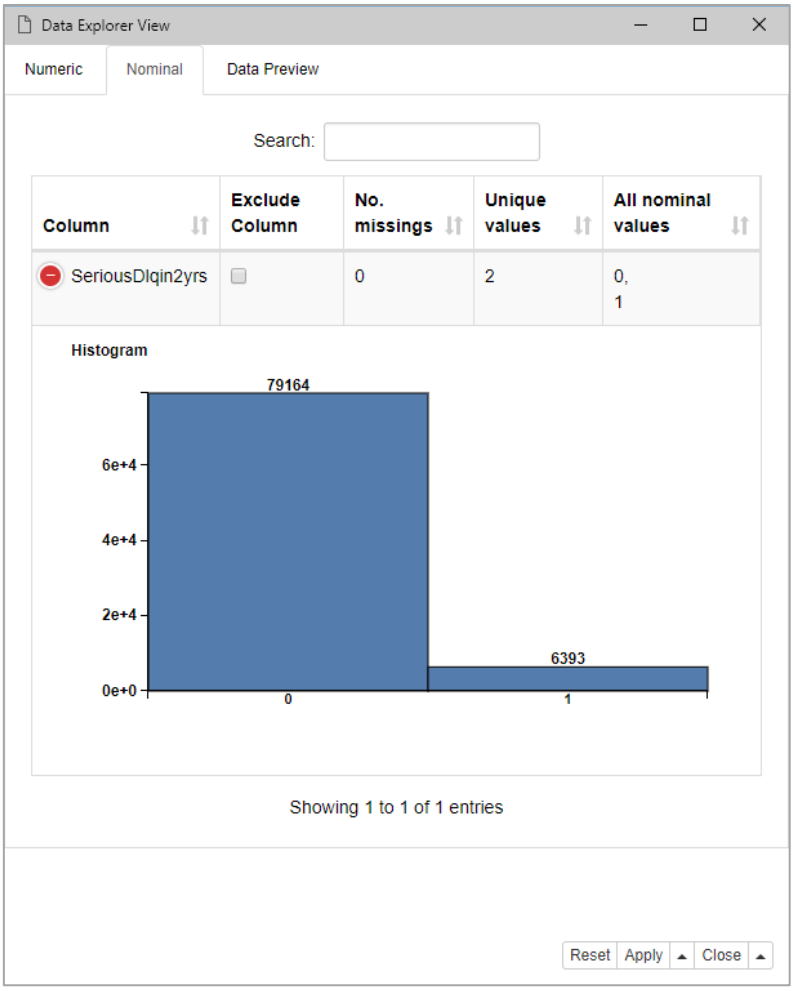


Figure 17. The bar chart of class occurrence in dataset clearly shows the class imbalance.

Linear Correlation Map

As part of the data exploration phase, we also created a linear correlation map. There you can see that "NumberOfRealEstateLoansAndLines" is highly correlated with "NumberOfOpenCreditLinesAndLoans", which is kind of saying that a lot of loan applications are house loan applications.

Also "NumberOfTimes90DaysLate" is highly correlated with "RevolvingUtilizationOfUnsecureLines", which might be more cryptic to understand.

Probably more known and less known facts will emerge from the inspection of this linear correlation map.

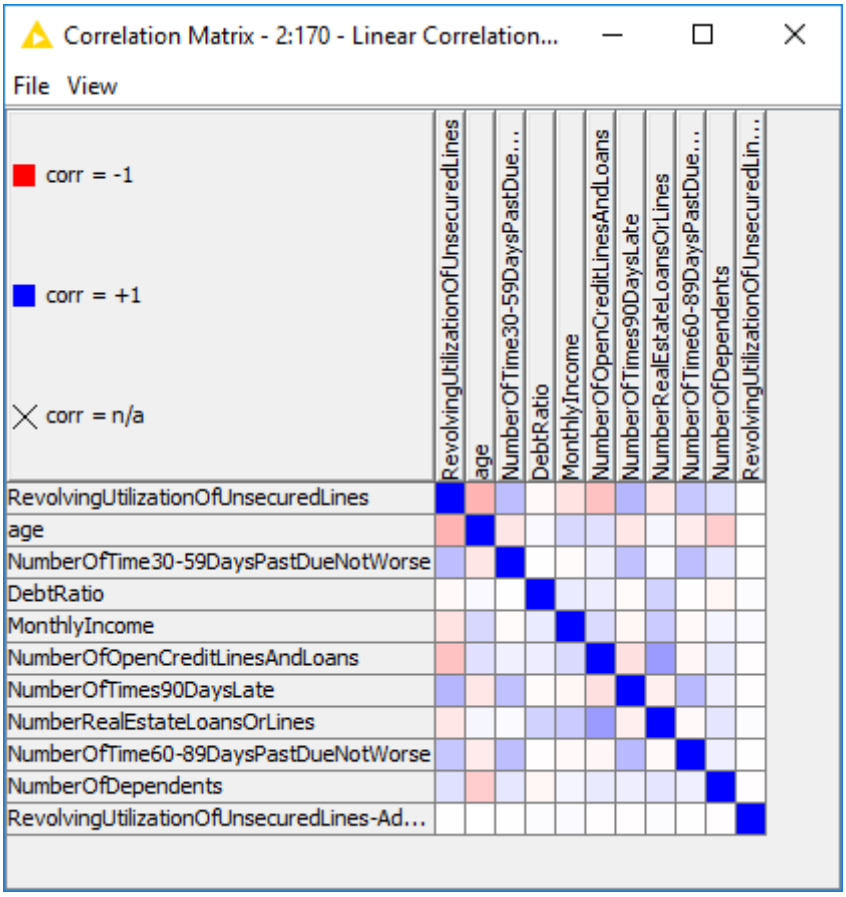


Figure 18. The linear correlation map shows correlation among some of the input features.

SMOTE Algorithm

After data preparation, the dataset was split into a training set (75%) and a test set (25%) and a Random Forest model was trained on the training set data. As usual, any other machine learning algorithm could have been used, e.g. neural networks, deep learning, gradient boosted trees, a simple decision tree, logistic regression, and so on. We went for a random forest model because it is easy to train, easy to interpret, and usually produces acceptable performances.

Evaluation on the test set produced a 92.83% accuracy and a 0.21 Cohen's Kappa. Remember the unbalanced class distribution? Such high accuracy might reflect only correct classification of the most frequent class. If we use a more balanced test set, by subsampling the most frequent class with an Equal Size Sampling node, accuracy gets reduced 57.04% and Cohen's Kappa to 0.14. This means that the model learned to better classify the most frequent class, neglecting the less frequent class patterns.

How can we force a machine learning algorithm to pay more attention to patterns in the least frequently occurring classes? Often increasing the frequency of the least numerous class in the training set forces the model to learn patterns in all classes equally. There are two ways to change the a priori distribution of the classes in the training set:

- Under-sampling the most frequent class
- Over-sampling the least occurring class

Under-sampling of the most frequent class, till we get more or less equally distributed classes, is implemented in the Equal Size Sampling node.

Over-sampling the least occurring class is implemented in the SMOTE node via the SMOTE algorithm. SMOTE (Synthetic Minority Over-sampling Technique) is an over-sampling approach in which *synthetic* new samples for the minority class are created with the same statistical properties [1]. This is a computationally expensive technique. For large datasets, it would be best to apply the under-sampling technique. However, our current dataset is not huge.

Thus, we decided to apply the SMOTE algorithm to the training set before proceeding with training the machine learning algorithm.

Model Evaluation

The model performances are reported below, evaluated on the original test set and on the under-sampled test set, when trained on the original training set, on the SMOTEd training set, or on the under-sampled training set.

As performance metric goes, accuracy is the most used metric even though it favors performance on the most frequent class; Cohen's Kappa on the other hand is supposed to be less partial, but can sometimes be difficult to interpret [2].

Training Set/Test Set	Original Test Set		Under-sampled Test Set		SMOTEd Test Set	
	Accuracy	Cohen's Kappa	Accuracy	Cohen's Kappa	Accuracy	Cohen's Kappa
Original Training Set	92.8%	0.21	57.0%	0.14	57.9%	0.16
Under-sampled Tr. Set	82.0%	0.31	79.1%	0.58	81.4%	0.63
SMOTE Training Set	92.7%	0.31	61.7%	0.23	94.4%	0.89

The table speaks for itself. However, to give a quick summary, the ***model trained only on the original training set*** performs well in terms of accuracy on a test set where the class imbalance is preserved. As soon as the minority class becomes artificially equally present, accuracy plummets. Obviously, the model has not learned a proper representation of the minority class.

The ***model trained on the under-sampled training*** set produces a lower accuracy and a higher Cohen's Kappa on the original unaltered test set, which might speak for a better performance on minority class. To be sure, let's check the performance metrics on more balanced test sets. On these test sets, accuracy is similar, but Cohen's Kappa improves dramatically. It is possible that this time, the model has not managed to learn all nuances of the most frequent class, due to the under-sampling.

Let's try now with the ***model trained on a SMOTEd training set***. In this case, the most frequent class should be represented adequately, while the minority class examples should still be enough to force the model to learn them. Here, accuracy on the original test set goes back to above 90%, which could mean a better internal representation of the most frequent class. Cohen's Kappa however is higher. We can hope that also the minority class was learned sufficiently well. Indeed, accuracy and Cohen's Kappa improved on the SMOTEd test set and to a lesser extent on the under-sampled test set.

What we have learned from this tiny experiment is that the SMOTE algorithm produces a larger and more general training set; however, it is computationally expensive and in some cases too slow to be used. In comparison, the under-sampling algorithm produces smaller and less general training sets, but at a smaller computational price.

We also learned that accuracy might not tell the whole story on an unbalanced test set and that machine learning algorithms might lean towards the most frequent class.

Based on the results above, we decided to apply the SMOTE algorithm to the training set and train a random forest with 100 trees on it.

Deployment

The deployment workflow reuses the trained random forest and applies it to the data of new applicants to estimate the risk of serious delinquency in the next two years as a binary prediction (1/0).

The workflow is a classic deployment workflow: read trained model, read deployment data, preprocess data, apply model to data, output predictions.

There are two things to notice here:

The ***preprocessing phase*** should be revised, since it cannot be exactly the same as in the training workflow. Outlier detection was removed since it does not make sense in a set of one or maybe just a few applications. Missing value imputation refers always to fixed values, sometimes extracted from the training set statistical properties, like the median values of a few columns.

This workflow is just a tiny piece within a Service Oriented Architecture. An external application service passes the data of the new applicant(s) to this prediction workflow via a JSON interface. The prediction result is then returned back to the calling service again via a JSON interface.

- In order to work as a REST service, the workflow is deployed on KNIME Server.
- In order to accept data in JSON format from the REST request, a Container Input (JSON) node is used.
- In order to return the results again in JSON format, a Container Output (JSON) node is used.

Notice the role of the Table to JSON and JSON to Table node to move data back and forth from KNIME to JSON format.

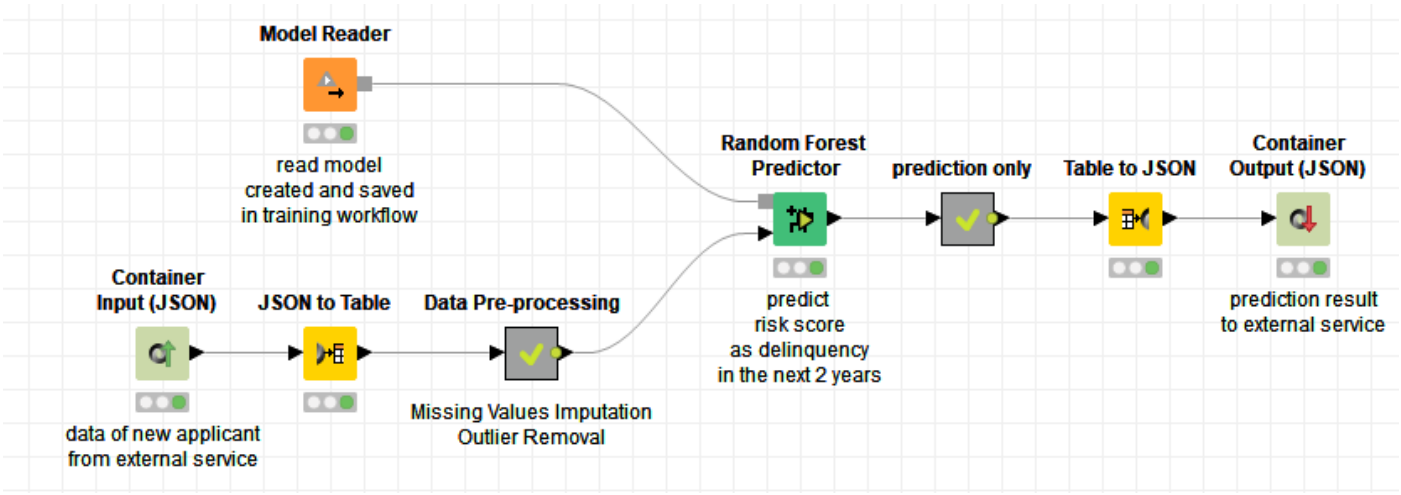


Figure 19. The prediction workflow is deployed as a REST service, accepting input data in JSON format and returning the results again in JSON format. This deployment workflow is available on the KNIME EXAMPLES server under *50_Applications/38_Credit_Risk_Assessment/Credit_Scoring_Deployment.*

References

[1] N. V. Chawla, K. W. Bowyer, L. O. , W. P. Kegelmeyer, "SMOTE: Synthetic Minority Over-sampling Technique", JAIR, Journal of Artificial intelligence Research, Vol. 16, 2002

[2] Arie Ben-David, "Comparison of classification accuracy using Cohen's Weighted Kappa", Expert Systems with Applications, Volume 34, Issue 2, February 2008, Pages 825-832.

Chapter 4. Web & Social Media

Social Media analytics is an important branch of data analytics. The goal is to extract insights from unstructured data, like text and connections, rather than numbers. Extract insights can take many different meanings: classify documents, understand conversation topics, detect text moods, track connection networks, or discover influencers.

The trick here is to transform unstructured data into numbers; that is to extract numerical descriptive measures from the unstructured mass of conversations.

4.1. Document Classification: Spam vs. Ham

By Kilian Thiel and Rosaria Silipo
Access workflow on hub.knime.com
Or from: EXAMPLES/08_Other_Analytics_Types/01_Text_Processing/02_Document_Classification

This use case is a text classification problem. We have all been there. Tons of comments under videos and blog posts, tons of email messages - and it's all garbage, or, in more modern terms, spam. One common use case for document classification is a spam filter. A spam filter needs to be able to distinguish spam messages from messages with actual content (ham). Thus, the goal here is to build a classifier of text data to distinguish spam vs. ham.

What You Need

For this use case, we will work on YouTube comments. In order to approach this problem as a classification task, we need a labeled dataset. We used the YouTube Spam Collection Data Set from the UCI ML Repository. This dataset [1] includes:

- a text column with the comments;
- the date the comment was posted;
- the author of the comment;
- the comment class: Spam (1) vs. Ham (0)

The dataset contains 1956 comments on the YouTube videos of five artists. Class distribution is roughly equal for spam and ham comments.

Workflow

It is, indeed, a classic classification problem. The only new step here is the conversion of text into numbers; i.e. text vectorization. There are many techniques for text vectorization: one-hot encoding, frequency encoding, number encoding, and recently added to the list word2vec encoding.

One-hot encoding represents text as a vector of 0/1 words. The dimension of the vector is the size of the dictionary and 0/1 indicates absence or presence of a given word in the text.

Instead of a binary encoding, a frequency, like a count, a relative frequency, or any other frequency measure, can be used. In this case we have a **frequency encoding**.

Words can also be associated with a number. A text then becomes a sequence of numbers. This is **number encoding**.

Recently, two deep learning encoder networks have been proposed. One using context as input to predict the target word and one that uses one word to predict its context. In both cases, the context words and the target word are one-hot encoded and the network has one hidden layer with lower dimensionality than the encoded word. The output values of the hidden layer are then used as **wod2vec** encoding for the word [2].

Before reaching the word encoding and text vectorization phase, the text should be cleaned, normalized, and words should be lemmatized or stemmed [3]. After reading the dataset, a series of dedicated nodes are used to perform all such operations on our YouTube comments. In particular we use the Snowball Stemmer node to reduce each word to its stem. The Snowball Stemmer node can deal with stemming in a number of different languages.

Note. All nodes for text processing are included in the KNIME TextProcessing extension and work on a special data type: Document. Hence the Strings to Document node at the very beginning of the pipeline.

After that, the document texts are parsed into words. The subset of most descriptive words is extracted and used as the base dictionary for text vectorization operations. The detection of the subset of most descriptive words for each text is based on keyword extraction techniques [3].

Next, the documents are transformed into document vectors with the Document Vector node. The document vectors are numerical representations of documents. That is, documents are now just sequences of numbers and can be classified with any machine learning algorithm. The final workflow training a Support Vector Machine on 70% of vectorized texts is shown in Figure 20. The Scorer shows a 95% accuracy of the model on the test set.

This training workflow is available in the upper part of the workflow *02_Document_Calssification* from the KNIME Community Workflow Hub and on the KNIME EXAMPLES server under *08_Other_Analytics_Types/01_Text_Processing*.

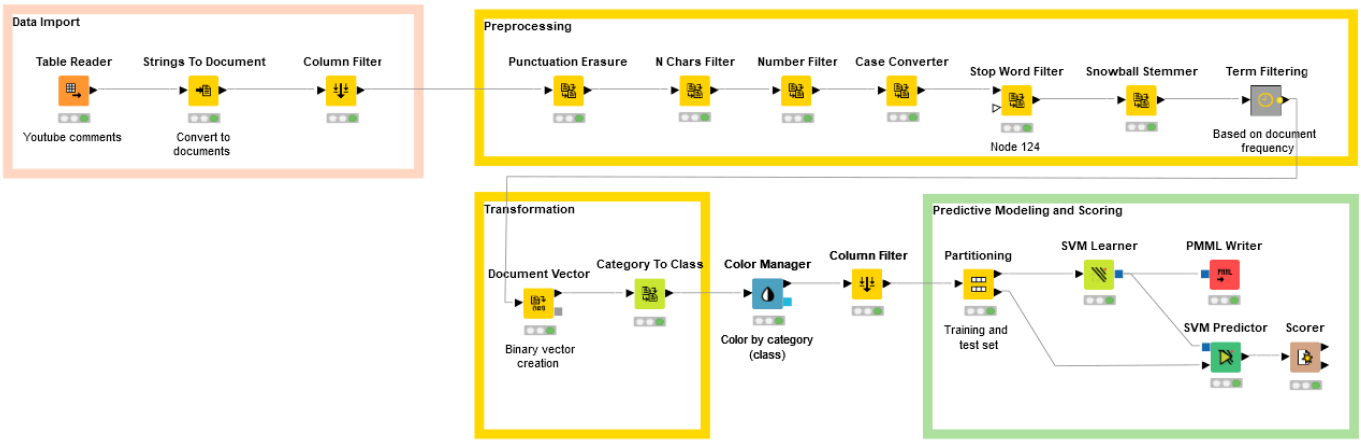


Figure 20. Training and evaluating a Machine Learning model (here SVM) to distinguish spam comments vs. "ham" comments. Notice the data preparation steps, which for text processing include a number of specific text cleaning and standardization procedures. This workflow is available on the KNIME EXAMPLES server in *08_Other_Analytics_Types/01_Text_Processing/02_Document_Classification.*

Deployment

The deployment workflow collects comments on a selected video directly from YouTube to ascertain whether they are ham or spam.

The comments are read here from a .table file with a Table Reader node. This is just a commodity, since reading a .table file allows the workflow to be fully executed without the need of YouTube credentials.

Note. Comments on a YouTube video could be extracted directly from the YouTube platform calling the YouTube web service with a GET Request node. An example of such data retrieval from YouTube can be found on the EXAMPLES server under *01_Data_Access/05_REST_Web_Services/03_Access_YouTube_REST_API* and is explained in the blog post "YouTube Metadata meet WebLog Files. What will it be tonight – a movie or a book?".

After acquiring the YouTube comments, the deployment workflow implements the same text preprocessing steps as the model training workflow: conversion from Strings to Documents, general clean-up and text standardization, keyword extraction, through to text vectorization with the Document Vector node.

Did you notice the Table Reader and Concatenate node after the Document Vector node? This is done to ensure the same structure and size of the document vector as for the document vectors used in training set. The Table Reader reads the basic empty structure of the document vector based on the full keyword dictionary as used in the training phase, while the Concatenate node followed by the Missing Value node ensure zero padding.

After this part of data preparation, the model, previously saved during the training phase, is read and applied to the deployment data. The last column of the output data table of the SVM Predictor node contains the ham vs. spam prediction of our model.

For example, the following comment:

"The first billion viewed this because they thought it was really cool, the other billion and a half came to see how stupid the first billion were..."

was classified as "ham".

This deployment workflow is available in the lower part of the workflow *02_Document_Calssification* on the KNIME Community Workflow Hub or from KNIME EXAMPLES server under *08_Other_Analytics_Types/01_Text_Processing*.

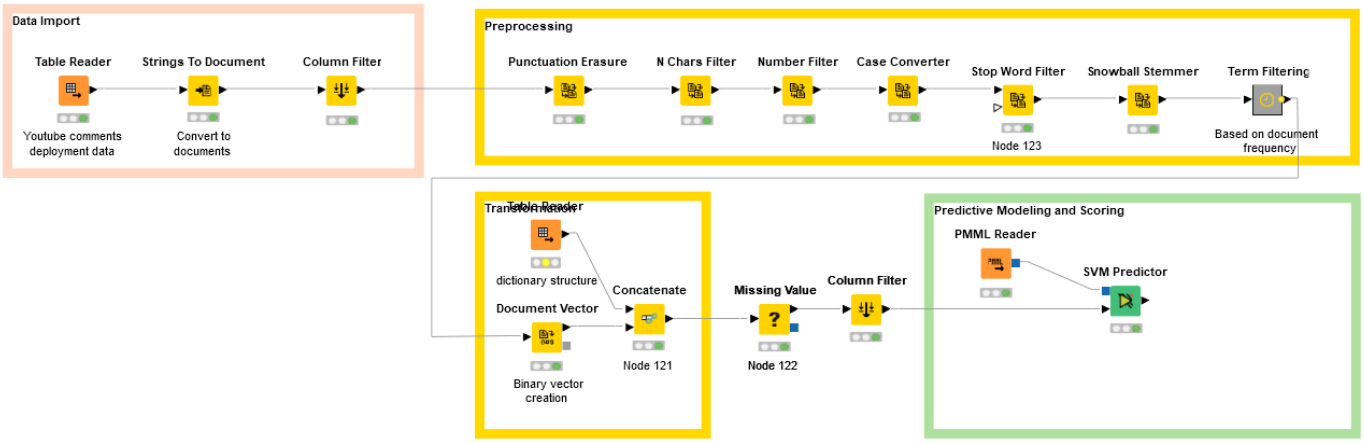


Figure 21. The deployment workflow to recognize current live comment as spam or ham. The model is a SVM and has been previously trained on the YouTube Spam Collection Corpus. Notice the Table Reader – Concatenate – Missing Value nodes to implement zero padding of current keyword representation on training dictionary.

References

[1] Alberto, T.C., Lochter J.V., Almeida, T.A. "TubeSpam: Comment Spam Filtering on YouTube", Proceedings of the 14th IEEE International Conference on Machine Learning and Applications (ICMLA'15), 1-6, Miami, FL, USA, December, 2015. (preprint)

[2] R. Silipo, "Word Embedding: Word2Vec Explained", KNIME blog, 2018

[3] V. Tursi, R. Silipo, "From Words To Wisdom", KNIME Press, 2017

4.2. Topic Detection: What Is It All About?

By Rosaria Silipo

Access workflow on hub.knime.com

Or from: Workflow in:

EXAMPLES/08_Other_Analytics_Types/01_Text_Processing/25_Topic_Detection_LDA

Text Summarization

Another classic use case in text analytics is text summarization; that is the art of extracting the most meaningful words from a text document to represent it.

For example, if we have to catalogue the tragedy of "Romeo and Juliet" by Shakespeare with let's say 5 keywords only, "love", "death", "young", "gentlemen", "quarrel" might be sufficiently descriptive. Of course, the poetry is lost but the main topics are preserved via this keyword-based representation of the text.

To obtain this kind of text summarization, we can use any of the keyword extraction techniques available in the KNIME TextProcessing extension: Keygraph Keyword Extractor, Chisquare Keyword Extractor, keyword extraction based on TF*IDF frequency measure. For a detailed description of the algorithms behind these techniques, refer to chapter 4 in [1]. Another common way to perform summarization of a text could be to use the LDA (Latent Dirichlet Allocation) algorithm.

The node implementing the LDA algorithm in KNIME Analytics Platform is the "Topic Extractor (Parallel LDA)" node. For more details on the LDA algorithm and the corresponding node, check chapter 6 in [1].

If we apply the "Topic Extractor (Parallel LDA)" node to the "Romeo and Juliet" tragedy searching for 3 topics each one represented by 10 keywords, we find one dominant topic mainly described by "love", "death", "ladi", "night", and "thou" and two minor topics described respectively by "gentlemen", "pretty", "ladi", "quarrel" and by "die", "youth", "villain", "slaughter". The topic importance is quantified by its keyword's weights. If we report those topics' keywords on a word cloud and we apply their weight to the word size, we easily see that topic 0 in red is the dominant one (Figure 23).

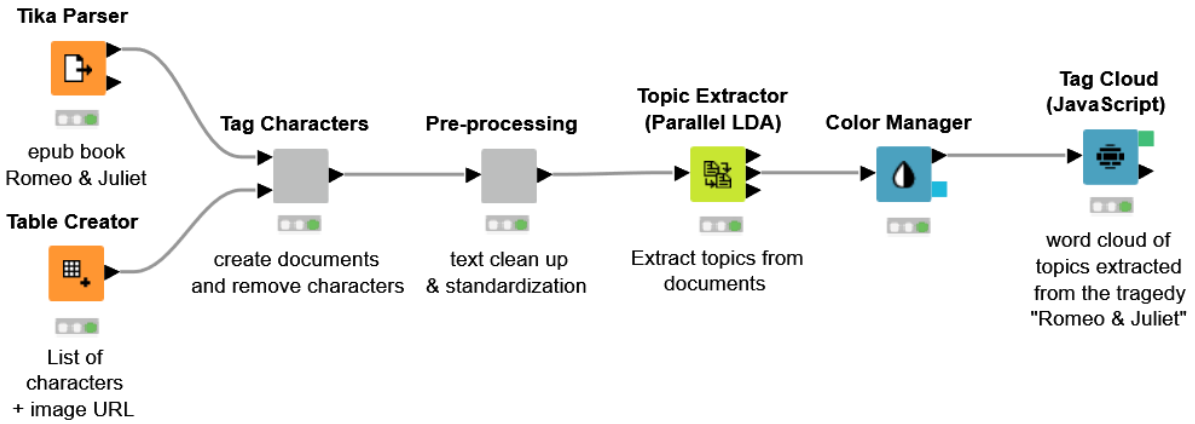


Figure 22. Summarizing the "Romeo and Juliet" tragedy with 10 keywords x 3 topics from a "Topic Extractor (Parallel LDA)" node.

Note. The Tika Parser node, used here to read the epub document of "Romeo & Juliet" tragedy, is a very versatile node which can read a number of different formats for text documents: from .epub to .pdf, from .pptx to .docx, and many more.

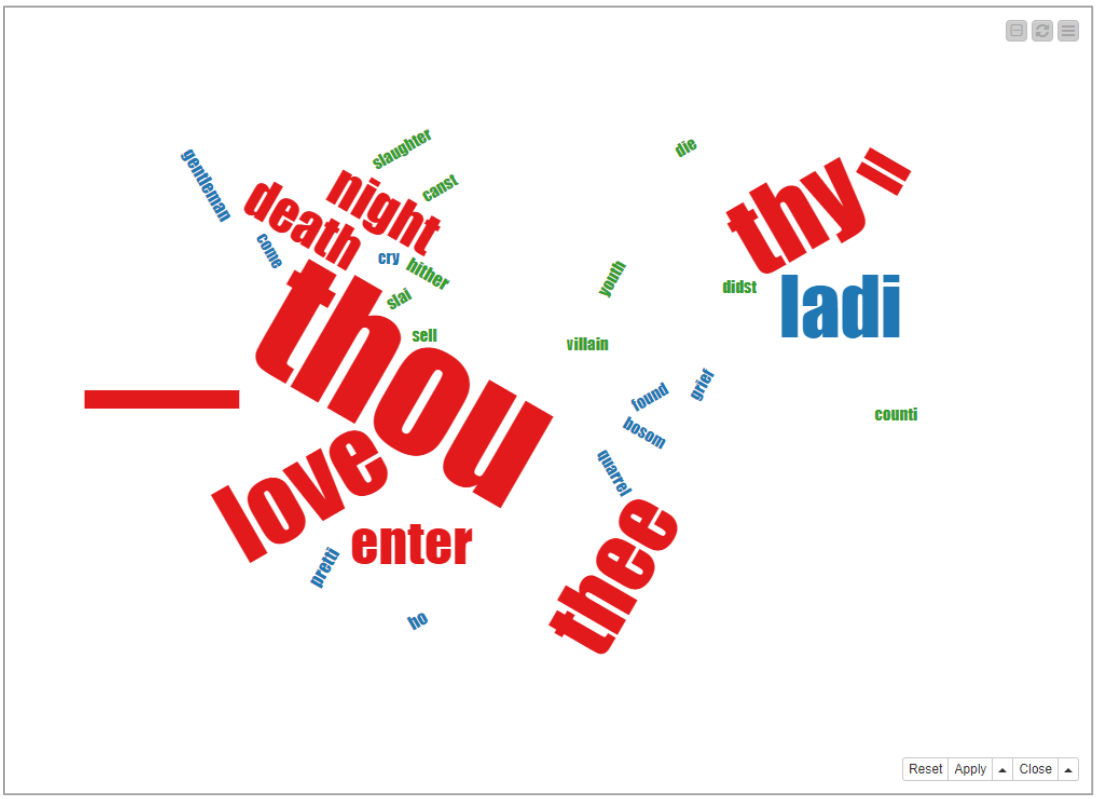


Figure 23. The resulting 3 topics - in green, blue, and red - summarizing the „Romeo and Juliet" tragedy. Keyword weight defines keyword size in the word cloud. Note the dominant red topic of "love" and "death".

Topic Detection

The LDA algorithm actually performs more than just text summarization; it also discovers recurring topics in a document collection. In this case we talk of topic detection.

The LDA algorithm extracts a set of keywords from each text document. Documents are then clustered together to find out the recurring keywords in groups of documents. These sets of recurring keywords can then be considered a topic covering a number of documents in the collection.

The node implementing the LDA algorithm in KNIME Analytics Platform is the "Topic Extractor (Parallel LDA)" node. For more details on the LDA algorithm and the corresponding node, check chapter 6 in [1].

Where is topic detection used?

Let's suppose we have a series of reviews for a product, a restaurant, or a touristic location. By extracting a few topics common to review groups, we can discover the features of the product/restaurant/touristic location that have impressed the reviewers the most.

Let's suppose we have a confused set of pictures with descriptions. We could reorganize them based on the topics associated with their description.

Let's suppose we have a number of newspapers each reporting on a set of news; detecting the common topics helps identify the journal orientation and the trend of the day.

You get the point. I am sure you can come up with a number of similar use cases involving topic detection.

What You Need

To implement a topic detection application via the LDA algorithm, you first need a collection of text documents, not necessarily labeled. LDA is a clustering technique: no labels are necessary. For this example, we have used a collection of 190 news articles from various newspapers in a file *news.table*. The goal is to label each news with the corresponding topic.

The Workflow

The central piece of a workflow implementing topic detection is the LDA node. After preparing the data, we feed the text documents into the LDA node. We then set the LDA node to report n topics (for example 7) and to describe each of these topics with m words (for example 10). Again, a word cloud of the topic keywords, with size proportional to the keyword weights, has been built and can be seen in Figure 24.

Note. Here, simply document based representation is sufficient. Word/Term extraction as well as text vectorization here are not necessary. The Topic Extractor (LDA) node performs all such operations internally.

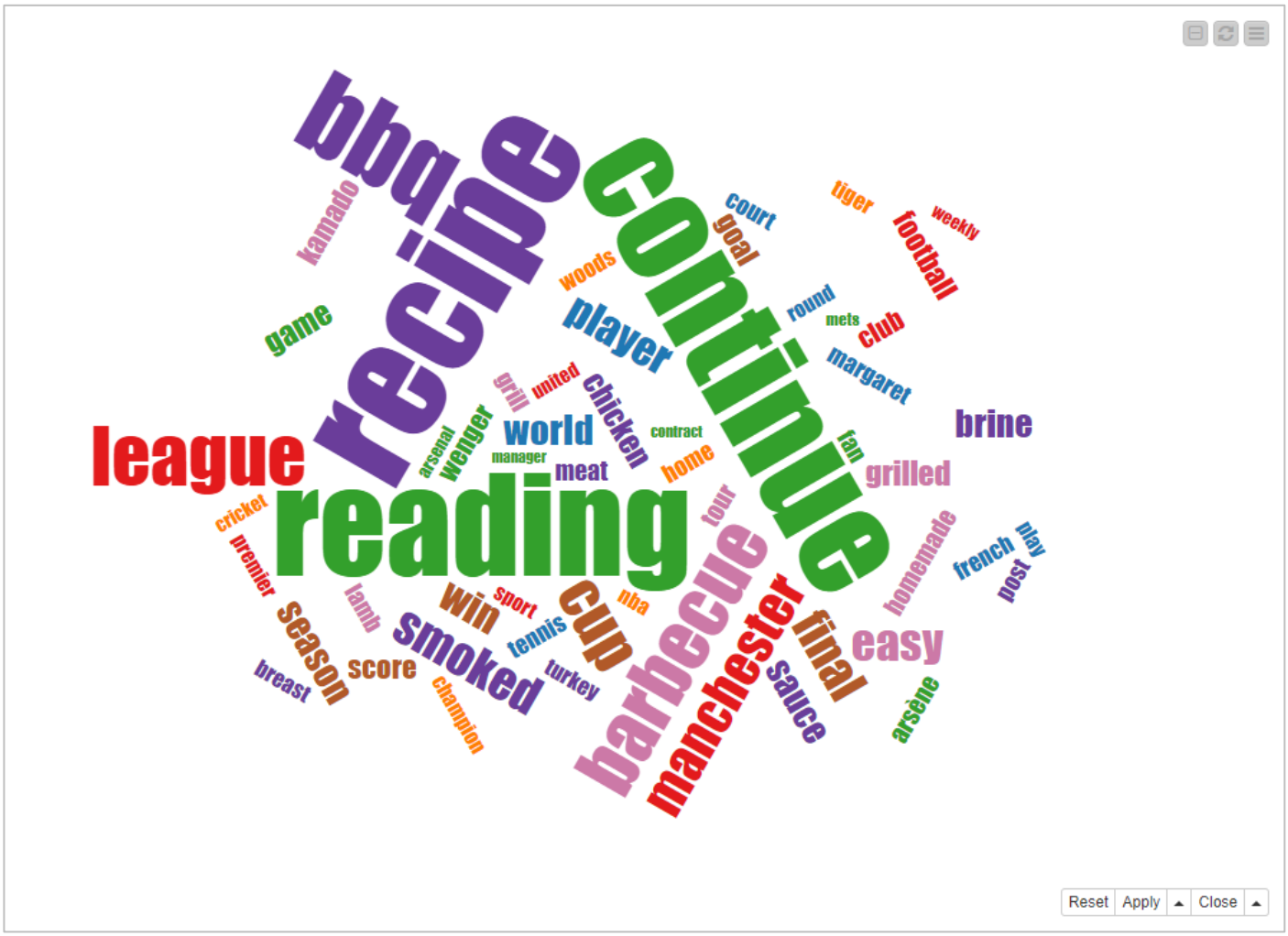


Figure 24. Word cloud of 10 keywords x 7 topics in the news dataset. The journal news all seem to relate to sport and bbq.

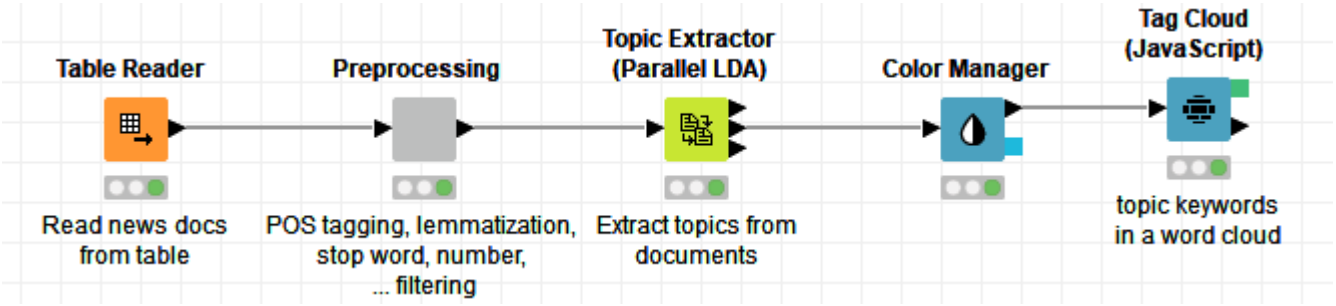


Figure 25. Workflow extracting 7 topics – each one described by 10 keywords – on the news dataset with the "Topic Extractor (Parallel LDA)" node.

From the word cloud clearly emerges a topic (in purple) about BBQ recipes, a second topic (in green) about reading sport news, followed by other topics concerning sports and grilling. It is obviously a news collection about outdoor cooking and sport watching.

In the first output table of the LDA node, each news article has been assigned to one of the discovered topics. The news articles are then grouped into 7 topic groups, as was the goal of this small project.

Deployment

There is no deployment for this use case, as there is no LDA Apply node. Indeed, the LDA node works only on one set of text documents. To calculate the distance between a word/keyword-based representation of a document and that of the detected topics, it would be really hard, if not impossible, mainly because of the reduced topic keyword dictionary. It is very unlikely that a new document will contain many – if any - of the few keywords used to describe the topics. That is why deployment in this case is the same as detecting the topics on a text collection.

In this use case, we have shown two workflows: one for single text summarization and one for a more general topic detection on a text collection. We have also explained the differences in goals and techniques among the two use cases.

References

[1] V. Tursi, R. Silipo, "From Words To Wisdom", KNIME Press, 2017

4.3. Sentiment Analysis: What's With the Tone?

By Rosaria Silipo

Access workflows on hub.knime.com – Sentiment Classification and Sentiment Analysis Lexicon Based Approach

Or from:

EXAMPLES/08_Other_Analytics_Types/01_Text_Processing/03_Sentiment_Classification

EXAMPLES/08_Other_Analytics_Types/01_Text_Processing/26_Sentiment_Analysis_Lexicon_Based_Approach

Introduction

Besides understanding what people are talking about, it is sometimes important to understand the tone of the conversation. A typical use case of this kind involves the analysis of feedback, any kind of wordy feedback. Depending on the tone of the feedback - upset, very upset, neutral, happy, and very happy – the feedback takes a different path in a support center. Similarly, the number of negative vs. positive comments can decide the future of a YouTube video or a Netflix movie.

Sentiment Analysis: The Techniques

There are two basic techniques for sentiment analysis: NLP (Natural Language Processing) based and ML (Machine Learning) based.

The NLP based approach relies on the words in the text and the sentiment they carry. This technique uses NLP concepts and a dictionary to extract the tone of the conversation.

The ML based approach needs a sentiment labeled collection of documents; this is a collection in which each document has been manually evaluated and labeled in terms of sentiment. Next, a ML supervised algorithm is trained to recognize the sentiment in each text. If a labeled dataset is not available, we could also change the ML supervised algorithm with a clustering procedure. However, we will not cover this approach here though.

What You Need

To evaluate sentiment in sentences or texts, we need of course some sentences or text examples. We used here the dataset of movie reviews provided by IMDB [1].

If we pursue the NLP based approach, we also need a dictionary of words with the sentiment they carry; that is at least a list of negative words and a list of positive words. We got those lists from the MPQA Corpus.

If we pursue the machine learning based approach, we need a sentiment label for each of our text examples. The IMDB dataset provides a positive vs. negative label, manually evaluated for each sentence.

The Workflow

NLP based Sentiment Analysis

The workflow for the NLP based sentiment analysis needs to:

- Clean and standardize the text in the document collection
- Tag all words as positive or negative according to the dictionary lists provided by the MPQA Corpus. (To do that we use the Dictionary Tagger node twice. All other words are removed.)
- Extract all remaining words from each document with a Create BoW node
- Calculate the sentiment score for each document as:

  Sentiment score = (# positive words - # negative words) / (# words in document)
- Define a threshold value as the average sentiment score

- Subsequently classify documents as:

  positive if sentiment score > threshold

  negative otherwise

- If you want to be more cautious you can define the positive and negative thresholds as:

  thresholds = avg(sentiment score) ± stddev (sentiment score)

  All documents with sentiment score in between the two thresholds can be classified as neutral.

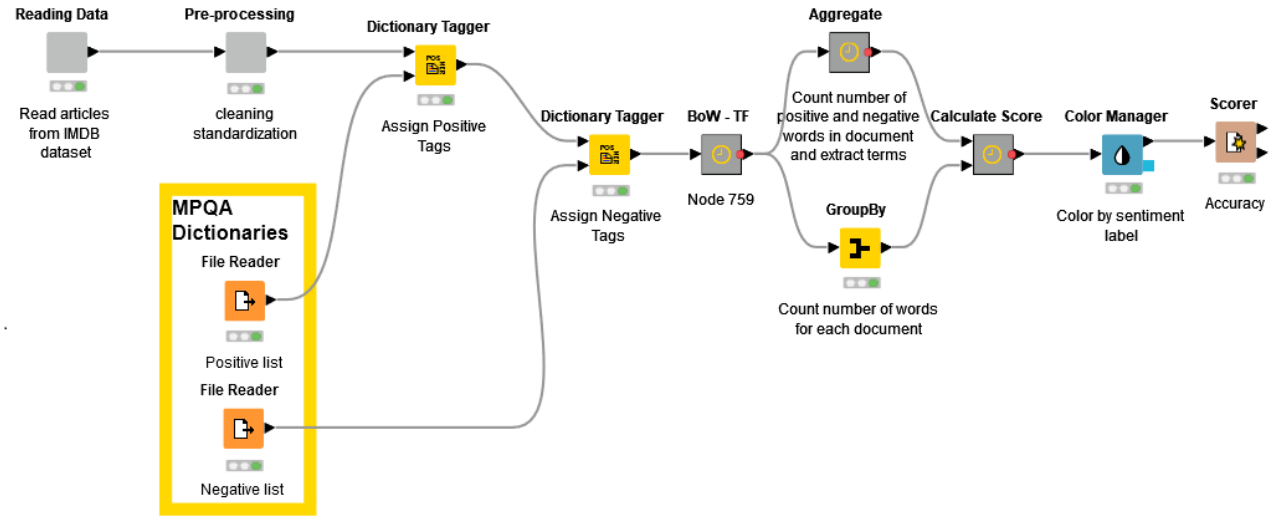


Figure 26. NLP based approach to sentiment analysis. Here you need a set of NLP based rules, which in the simplistic case consists of just two lists of words: one list for positive words and one list for negative words. Here the two lists of words are taken from the MPQA Corpus.

If we assign colors to the IDBM reviews according to the predicted sentiment – green for positive and red for negative sentiment – we get the result table in Figure 27.

Note. This is a very crude calculation of the sentiment score. More complex rules could of course be applied, like for example inverting the word sentiment polarity after a negation and taking into account the time evolution of the sentence.

Table with Colors - 0:276 - Color Manager (Color by sentiment)

File Hilite Navigation View

Table "default" - Rows: 2000 | Spec - Columns: 7 | Properties | Flow Variables

Row ID	Document	Negativ...	Positive...	all Words	Sentim...	Sentim...	Docum...
Row0_Row407	"httpwwwimdbcomtitlett0000010userc...	5	34	489	0.059	POS	POS
Row1_Row20	"httpwwwimdbcomtitlett0001032userc...	3	8	145	0.034	NEG	POS
Row2_Row260	"httpwwwimdbcomtitlett0005557userc...	20	53	624	0.053	POS	POS
Row3_Row165	"httpwwwimdbcomtitlett0013662userc...	16	54	431	0.088	POS	POS
Row4_Row519	"httpwwwimdbcomtitlett0014538userc...	2	3	53	0.019	NEG	POS
Row5_Row1540	"httpwwwimdbcomtitlett0017196userc...	8	14	209	0.029	NEG	NEG
Row6_Row944	"httpwwwimdbcomtitlett0018016userc...	1	10	106	0.085	POS	POS
Row7_Row377	"httpwwwimdbcomtitlett0018016userc...	10	9	141	-0.007	NEG	POS
Row8_Row710	"httpwwwimdbcomtitlett0019071userc...	2	9	181	0.039	NEG	POS
Row9_Row267	"httpwwwimdbcomtitlett0019071userc...	23	67	861	0.051	POS	POS

Figure 27. Movie reviews with predicted sentiment from NLP based approach: red for negative and green for positive sentiment.

ML based Sentiment Analysis

The workflow implementing the Machine Learning (ML) based approach to sentiment analysis needs to:

- Again, clean and standardize the texts in the documents

- Extract all words from the documents with a Create BoW node
- Produce a text vectorization of the document with a Document Vector node
- Train a ML algorithm to recognize positive vs. negative texts
- Evaluate the created model on test documents

Note. The training and test phase are implemented exactly as in any other machine learning based analysis.

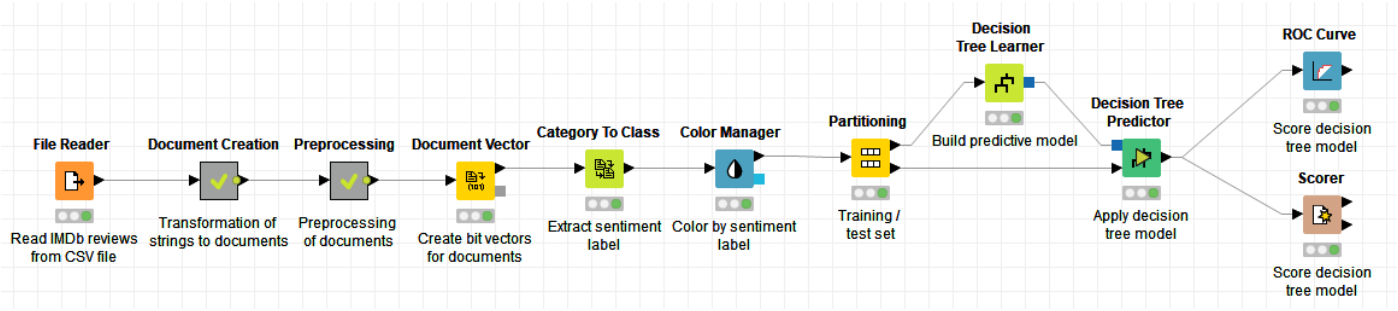


Figure 28. Machine Learning based approach for sentiment analysis. Here we train a decision tree, but of course any other supervised ML model for classification could be used.

Usually the machine learning based approach performs better than the dictionary-based approach, especially when using the simple sentiment score adopted in our NLP approach. However, sometimes there is no choice, since a sentiment labeled dataset is not available.

Deployment

Deployment workflow for an NLP based sentiment analysis is practically the same as the training workflow in Figure 26. The only difference consists in the threshold calculation. Indeed, the threshold is calculated only on the training set and then just adopted in the deployment workflow.

The deployment workflow for a machine learning based sentiment analysis looks like any other machine learning based deployment workflow. Data are imported and preprocessed as needed; the model is acquired; data are fed into model; predictions from model on input data are generated and presented to the end user.

References

[1] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. (2011). Learning Word Vectors for Sentiment Analysis. *The 49th Annual Meeting of the Association for Computational Linguistics (ACL 2011).*

4.4. Find the Influencers

By Marten Pfannenschmidt and Paolo Tamagnini

Access workflow on hub.knime.com

Or from:

EXAMPLES/50_Applications/19_TwitterAnalysis/03_Visualizing_Twitter_Network_with_a_Chord_Diagram

Introduction

There are two main analytics streams when it comes to social media: the topic and tone of the conversations and the network of connections. You can learn a lot about a user from their connection network!

Let's take Twitter for example. The number of followers is often assumed to be an index of popularity. Furthermore, the number of retweets quantifies the popularity of a topic. The number of crossed retweets between two connections indicates the livelihood and strength of the connection. And there are many more such metrics.

@KNIME on Twitter counts more than 4500 followers (as of October 2018): the social niche of the KNIME real-life community. How many of them are expert KNIME users, how many are data scientists, how many are attentive followers of posted content?

In this use case we want to analyze the top 20 active followers of @KNIME on Twitter and arrange them in a chord diagram (Figure 29).

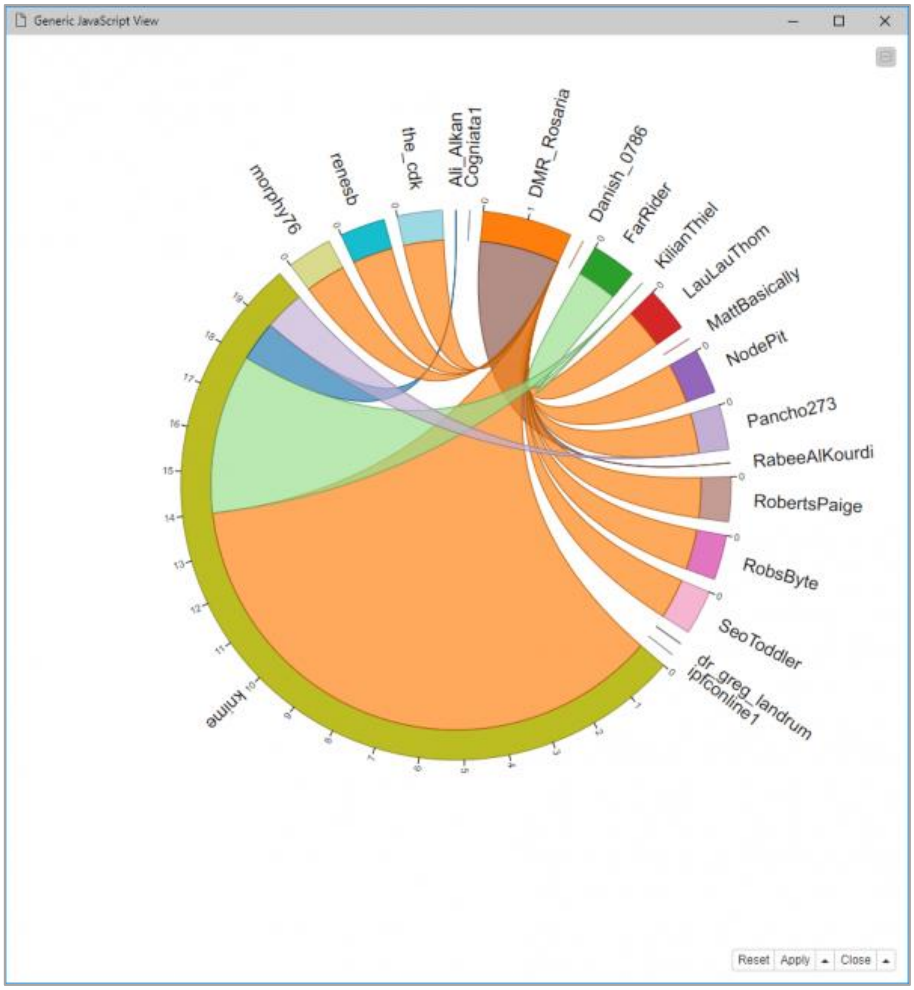


Figure 29. Chord diagram visualizing interactions from the top 20 Twitter users around #knime. Nodes are represented as arcs along the outer circle, and connected to each other via chords. The total number of retweeted tweets defines the size of the circle portion (the node) assigned to the user. A chord (the connection area) shows how often a user's tweets have been retweeted by a specific user, and is in the retweeter's color.

A chord diagram is another graphical representation of a graph. The nodes are represented as arcs along the outer circle and are connected to each other via chords.

The chord diagram displayed above refers to tweets including #knime during the week following July 12, 2018. This was the week immediately after the KNIME 3.6 release. The number of retweeted tweets defines the size of the circle portion (the node). Each node/user has been assigned a random color. For example @KNIME is olive, @DMR_Rosaria is orange, and @KilianThiel is green.

Being the week after the release of KNIME Analytics Platform 3.6, it is not surprising that @KNIME occupies such a large space on the outer circle.

The number of retweets by another user defines the connection area (chord), which is then displayed in the color of the retweeter. @DMR_Rosaria is an avid retweeter. She has managed to retweet the tweets by @KNIME and KNIME followers disproportionally more than everybody else and has therefore managed to make the color orange the dominant color of this chart. Moving on from orange, we can see that the second retweeter of KNIME tweets for that week has been @KilianThiel.

The Workflow

We'd now like to show how we built the chord diagram.

Data Access

We access the data by using the Twitter nodes included in the KNIME Twitter API extensions. If you want to learn more about how to access Twitter data with KNIME, have a look at the Twitter Data Collection workflow or one of the other Twitter workflows on the KNIME Workflow Hub.

We gathered the sample of data around the hashtag *#knime* during the week following the release of KNIME Analytics Platform 3.6 on July 12 2018. Each record consists of the user name, the tweet itself, the posting date, the number of reactions and retweets and, if applicable, who retweeted it.

Let's build the network of retweeters. A network contains edges and nodes. The users represent the nodes and their relations, i.e. how often user A retweets user B is represented by the edges. Let's build the edges first:

1. We filter out all tweets with no retweets or that consist of auto-retweets only.
2. We count the number of retweets a user has retweeted tweets of another user.

To clean the data and compute the edges of the network all you need are two Row Filter nodes and a GroupBy node.

The Matrix of Nodes and Interactions

Now we want to build a weighted adjacency matrix of the network with usernames as column headers and row IDs, and the number of retweets by one username on the tweets of the other in the data cell. We achieve that by addressing the following steps.

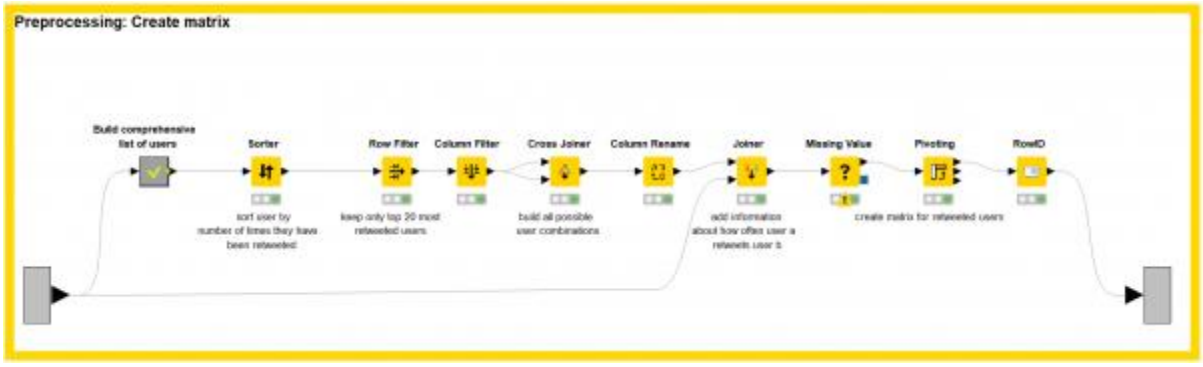


Figure 30. This metanode builds the matrix of interactions between Twitter usernames around #knime.

1. We build a comprehensive list of all users (usernames), both tweeting and retweeting, and count the number of times their tweets have been retweeted overall. These numbers will fill the nodes of the network.
2. We narrow our analysis down to investigate only the 20 topmost retweeted users. That is why we sort them in descending order with respect to the number of retweets on their tweets and keep only the top 20.
3. Using a Cross Joiner node, we build the pairs of users. From an original set of twenty users we end up with 400 different user pairs.
4. To these user pairs we add the previously computed edges by using a Joiner node.
5. The Pivoting node then creates the matrix structure from the (username1, username2, count of retweets) data table.

You can read it like this: "The user named in Row ID's row was retweeted n times by the user named in the column header's column."

Drawing the Chord Plot

- The matrix we created is the data input for a Generic JavaScript node.
- The Generic JavaScript node draws the chord diagram.
- To draw the chord diagram, we need the D3 library which can be added to the code in the Generic JS node.
- The JS code required to draw this chart is relatively simple and is shown below:

```
// creating the chord layout given the entire matrix of connections.
var g = svg.append("g")
.attr("transform", "translate(" + width / 2 + "," + height / 2 + ")")
.datum(chord(matrix));

// creating groups, one for each twitter user.
// each group will have a donut chart segment, ticks and labels.
var group = g.append("g")
.attr("class", "groups")
.selectAll("g")
.data(function(chords) { return chords.groups; })
.enter().append("g")
.on("mouseover", mouseover)
.on("mouseout", mouseout)
.on("click", click);

// creating the donut chart segments in the groups.
group.append("path")
.style("fill", function(d) { return color(d.index); })
.style("stroke", function(d) { return d3.rgb(color(d.index)).darker(); })
.attr("d", arc)
.attr("id", function(d) {
return "group" + d.index;
})

// creating the chords (also called ribbons) connections,
// one for each twitter users pair with at least 1 retweet.
g.append("g")
.attr("class", "ribbons")
.selectAll("path")
.data(function(chords) { return chords; })
.enter().append("path")
.attr("d", ribbon)
.style("fill", function(d) { return color(d.target.index); })
.style("stroke", function(d) { return d3.rgb(color(d.target.index)).darker(); });
```

More Formal Network Analysis Technique

If you would prefer a more traditional method to visualize your results, there are also common KNIME nodes to analyzing your social media network in the "classic" and more formal way. Using these nodes also means that you don't have to use any JavaScript programming. What we want to do is analyze the same network of the 20 most active followers of @KNIME on Twitter, but this time with the KNIME Network Viewer node.

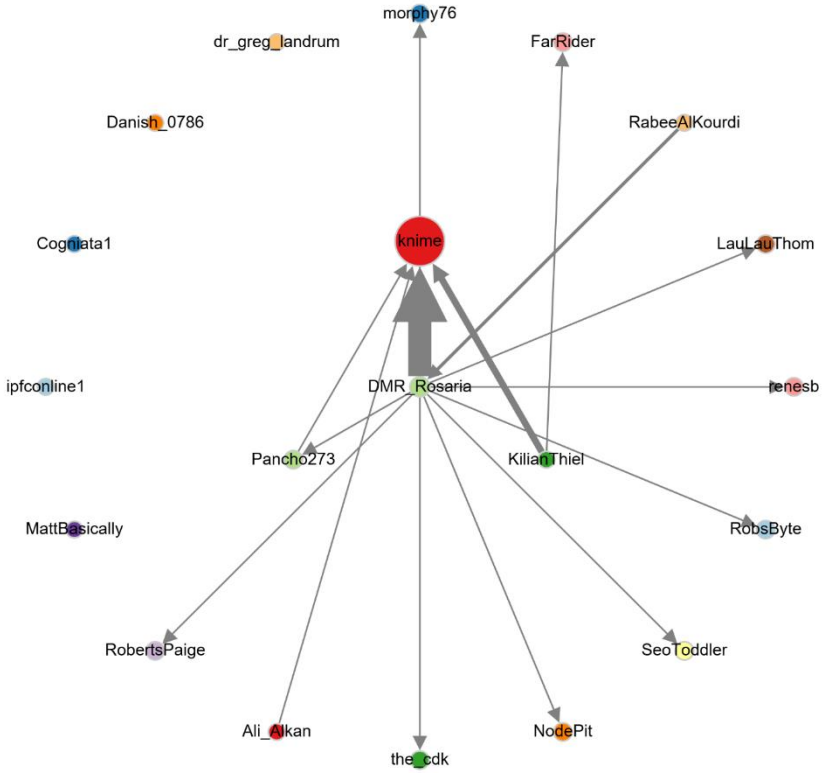


Figure 31. Top 20 Twitter users around #knime visualized as network map from KNIME Network Viewer node. Nodes of the underlying graph are represented as circles and are connected via arrows. The size of a circle (node) is defined by the total amount of times a user has been retweeted by one of the other users. The size of an arrow (edge) represents how often one user retweeted another user's tweets.

This network map displays the graph with the following key elements: nodes are represented by a specific shape, size, color, and position. We arbitrarily chose circles for the shape. Each node is colored and labeled with respect to the user it represents. The circle's size is dependent on the overall amount of times the specific user's tweets have been retweeted by other users. The position of nodes in this case is defined by their degree. The more input and output connections a node has, the higher its degree and the more centric it is displayed on the network map. Another key element are the edges, which connect the nodes. They are visualized as arrows, as we visualize a directed graph. The direction of the arrow shows which user retweets who's tweets, while its size depends on the number of retweets.

Conclusion

To summarize: we have shown formal graph as well as an alternative approach to the more traditional network visualization techniques by using a chord diagram. We did this by leveraging the flexibility of the Generic JavaScript View node. The JavaScript code required is an adaption of an existing D3 template.

4.5. Sentiment & Influencers

By Rosaria Silipo, Kilian Thiel, Tobias Kötter, Phil Winters

Access workflow on hub.knime.com

Or from:

EXAMPLES/08_Other_Analytics_Types/04_Social_Media/02_NetworkAnalytics_meets_TextProcessing

Introduction

A few years ago, we started a debate about whether the loudest customers really were as important as everybody – including they themselves – thought! Customer care usually reacts faster to the loudest complainer. Is this convenient? And how to identify those complainers worth investing time with?

Happy and disgruntled users are easily identifiable via **sentiment analysis** on their social interaction texts. The degree of influence of each user can be measured through an **influence score**. There are many influence scores available. A widely adopted one is the centrality index. The idea of this use case is to combine the sentiment measure with the influence score and in this way to identify those disgruntled customers/users with a high degree of influence. Support time and resources should be invested on such customers or users.

What You Need

The original use case referred to the launch of a new product and aimed at collecting opinions from the beta users. Since it is impossible to share the original dataset due to the company privacy policy, for this particular case, we replaced it with a publicly available similar dataset: the Slashdot News Forum.

Slashdot (sometimes abbreviated as "/.") is a social news website, which was founded in 1997 for science and technology. Users can post news and stories about diverse topics and receive online comments from other users.

The Slashdot dataset collects posts and comments for a number of sub-forums, such as Science Fiction, Linux, Astronomy, etc… Most of the users posted or commented using their username, while some participated anonymously. The biggest sub-forum revolves around politics and contains about 140,000 comments to 496 articles from a total of about 24,000 users. For the purposes of this use case we focus on the "Politics" sub-forum.

Users in the Slashdot dataset are not strictly customers. However, when talking about politics, we can identify the political topic as the product and measure the user reactions as we would for a product.

Each new post is assigned a unique thread ID. Title, subdomain, user, date, main topic, and body all refer to this thread ID. A new data row is created for each comment with comment title, user, date, and body, and appending the thread ID, post title, post user, post date, and post body from the seed post as well.

The seed post and its related comments are connected via the unique Thread ID. The seed post is the first item in the thread and the title of the seed post becomes the title of the thread. In Figure 32 you can see the seed post data on the left and the data for the corresponding comments on the right. Notice that multiple comments might refer to the same seed post.

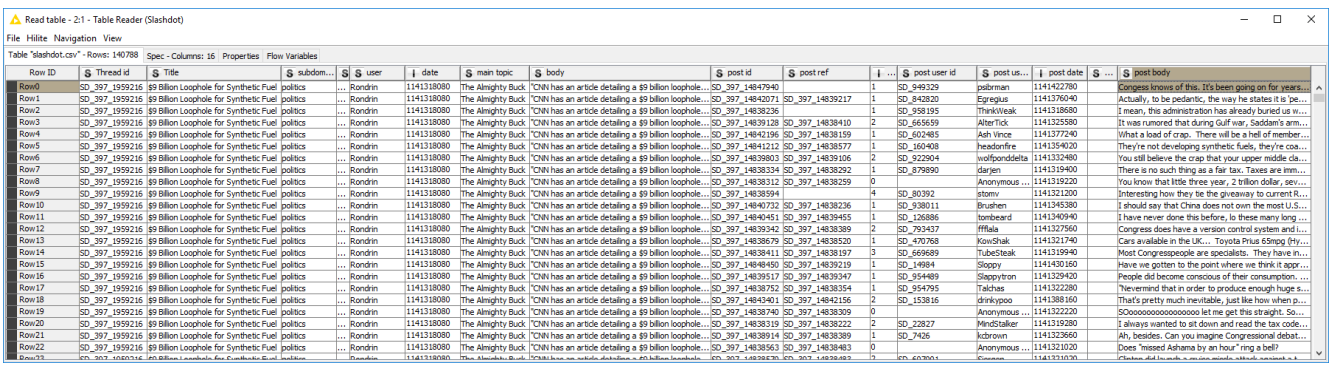


Figure 32. SlashDot Dataset. Data from seed post on the left; data from related comments on the right.

The Workflow

In the analysis, we took all of the non-anonymous users into consideration. Thus, the first step is to remove all data rows where “user” is either “Anonymous Coward” or “An anonymous reader”, where the username is too long, or there is no post ID. This happens in the “Preprocessing” metanode.

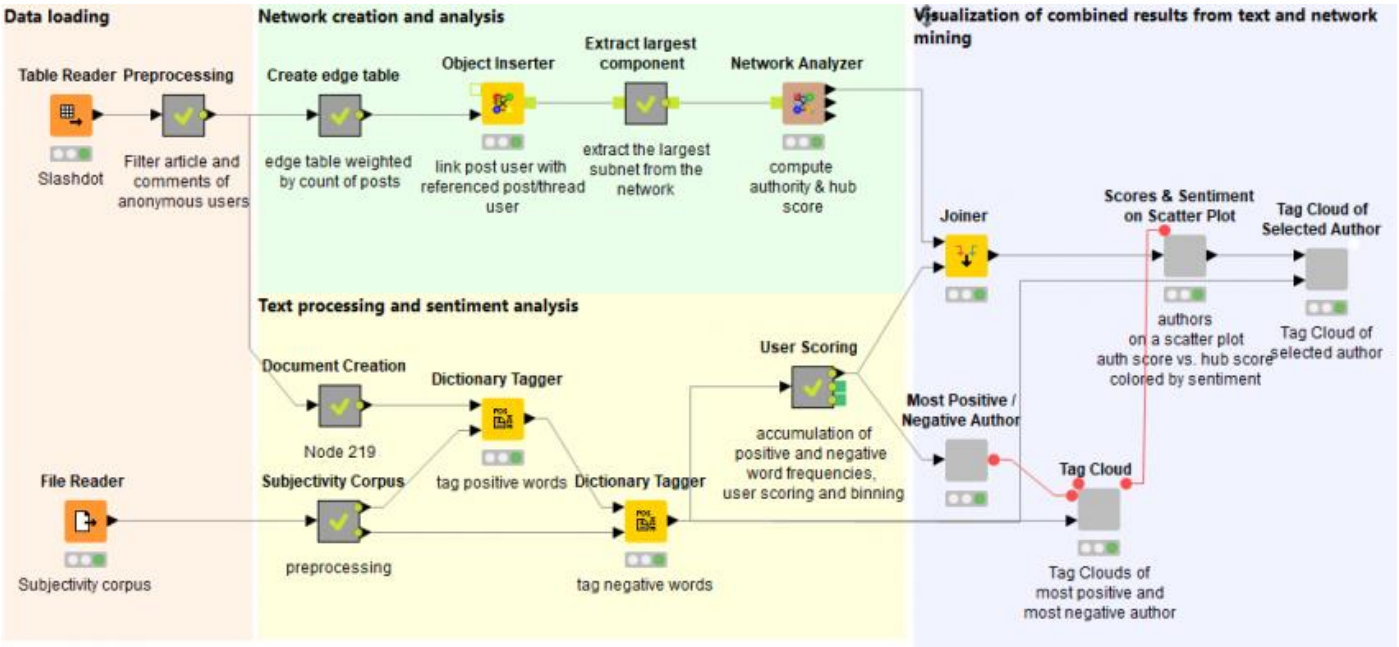


Figure 33. The upper part of the final workflow, referred to as "Network creation and analysis," calculates influence scores. The lower part, labeled "Text processing and sentiment analysis", measures the sentiment of each forum user. This workflow is available on KNIME EXAMPLES Server under: 08_Other_Analytics_Types/04_Social_Media/02_NetworkAnalytics_meets_TextProcessing.

Influence Scores

Here we want to find out who the most influential users are, by investigating the connections across users. Each post has a post ID - i.e. thread ID - a title, a user author, and a body text. Comment posts also have a field “post ref” identifying the post they are responding to. The goal is to build a network object to represent user interactions.

The first step is to prepare the edge table as the basis for the network. An edge table has a source column - the authors of the posts - and a target column – the authors of the reference posts. The edge table is built by the “Create edge table” metanode in the upper branch of the final workflow shown in Figure 33. A left outer join puts together all post authors (source) with all reference authors (target), if any. A GroupBy node then counts the number of occurrences for each connection from source to target. The string “<source user> -> <target user>” is set as the edge ID of the connection. All auto-connections, i.e. users answering themselves, are filtered out.

The edge table is now ready to be transformed into a network object. This is the job of the Object Inserter node. This node transforms the source and target users into nodes and connects them via an edge with the number of connection occurrences as value and the edge ID string as label.

After that, the metanode named “Extract largest component” first splits the network into all of its connected components, using a Network to Row node. Each sub-network is represented as a String and occupies a data row. Then it loops through all of the sub-networks and counts the number of edges and nodes, using a Network Analyzer node. Sub-networks are sorted based on their total number of edges and nodes and the largest sub-network is kept for further analysis.

Finally, a second Network Analyzer node calculates the Hub and Authority score.

The Network Analyzer node provides a great summary for social media activity. It calculates the number of different statistics on a network graph at both node and edge level. Such statistical measures try to establish the importance

of each node and edge by the number of its connections, their weight, their neighbors, the distance to their neighbors, and similar other parameters. Two of those importance measures are hub and authority scores.

The concept of hubs and authorities, as described in https://nlp.stanford.edu/IR-book/html/htmledition/hubs-and-authorities-1.html , is rooted in web pages. There are two primary kinds of web pages as results for *broad-topic searches*:

- Authoritative sources of information on the topic (*authorities*)
- Hand-compiled lists of links to authoritative web pages on the topic (*hubs*).

Hubs are not in themselves authoritative sources of topic-specific information, but rather direct you to more authoritative pages. The hub/authority score calculation relies on hub pages to discover the authority pages.

To calculate the hub and authority score, the Network Analyzer node implements the HITS algorithm in the JUNG (Java Universal Network/Graph) Framework.

Sentiment Analysis

Now we want to measure the sentiment, i.e. characterize each forum user in terms of positivity and negativity rather than authority.

The lower branch of the workflow in Figure 33 creates the list of Documents for each forum user, from posts or comments he/she has written. At the same time, it imports two lists of words: negative words and positive words from the English dictionary according to the MPQA Subjectivity Lexicon. Words in all documents are tagged as positive or negative by the two Dictionary Tagger nodes, depending on whether they match any of the words in these two lists. Untagged words are considered as neutral.

Each positive word is assigned a +1 value, each negative word a -1 value, each neutral word a 0 value. By summing up all word values across all documents written by each user, we calculate the user sentiment score. I wonder what the most negative author might say. Just out of curiosity, we shall draw a word cloud for the most positive and the most negative author (Figure 34).

Word clouds are calculated and displayed through Tag Cloud (Javascript) nodes in the last wrapped metanode on the right, named “Tag Cloud”. Here, two word clouds are displayed side by side: the word cloud for the most positive author on the left; the word cloud for the most negative author on the right. Positive words are colored in green, negative words in red, and neutral words in gray. Well, it is easy to see why the most negative author has been labeled … well… negatively!

Note that user sentiment score is calculated here using the absolute word frequency without taking into account the number of words used. For corpuses with longer documents, i.e. with a more considerable difference in number of words, the relative frequency might be more suitable.

Finally, forum users with a sentiment score above (average + standard deviation) are considered positive authors; forum users with sentiment score below (average – standard deviation) are considered negative; all other users in between are considered neutral. Positive users are color coded green, negative users red, and neutral users gray.

Figure 34. Word cloud respectively for the most positive user (on the left) and the most negative user (on the right). In the midst of all gray (neutral) words, you can see a predominance of green (positive) words on the left and of red (negative) words on the right. Notice the repeated word "Stupid" that has gained author pNutz the title of most negative post author.

Putting It All Together

To put it all together, a Joiner node joins the authority and hub score with the sentiment score by author.

A Scatter Plot (Javascript) node, inside the wrapped metanode "Scores and Sentiment on Scatter Plot", plots the forum users by hub score on the y-axis, authority score on the x-axis, and sentiment score as color.

Notice that the loudest complainers in red have actually very little authority scores and therefore cannot be considered influencers. Thus, this plot seems to go against the common belief that you should listen and pamper the most aggressive complainers. Notice also that the most authoritative users are actually neutral. This neutrality could well be one of the reasons why other users trust them.

The scatter plot view produced by the Scatter Plot (Javascript) node is interactive. By clicking the "Select mode" button at the top of the view, it is possible to select single points on the scatter plot with a single-click or group of points by drawing a rectangle around them. In Figure 35, the "Select mode" button is circled in red as well as the selected point, in this case a green point, i.e. a positive author named Guppy06.

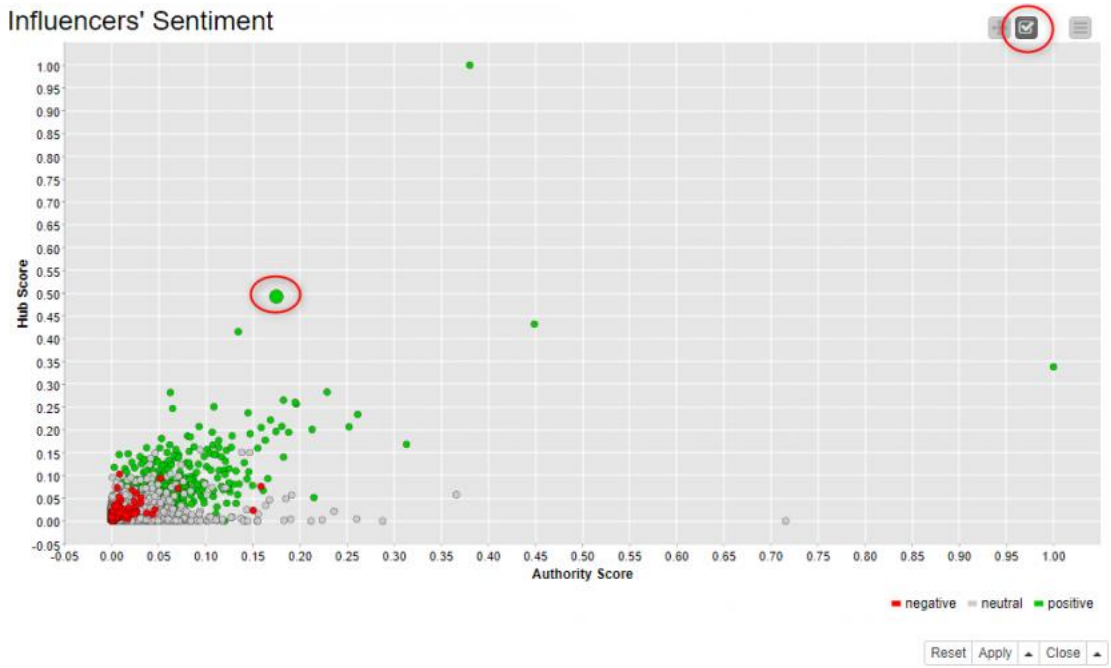


Figure 35. Authors as points on a scatter plot with authority score on the x-axis and hub score on the y-axis. Authors with a positive sentiment score, i.e. sentiment score > (average + std dev), are color coded green. Authors with a negative sentiment score, i.e. sentiment score < (average - std dev), are color coded red. Authors with sentiment score in between are labeled as neutral and depicted in gray. In the upper right corner, there are the buttons for zooming and selection. The circled button enables point/author selection. The bigger point in the plot is the point that has been selected by single-click.

After clicking the Close button and opting to keep the selection as the new default, the selected author is moved to the next visualization wrapped metanode, where all his/her posts are extracted and the word cloud is displayed.

Figure 36. Word cloud of the selected author in the previous scatter plot (Figure 35).

The selecting of single points in the scatter plot is made possible via the option “Enable Selection” in the “View Controls” tab in the configuration window of the “Scatter Plot (Javascript)” node.

This same sequence of plots can be visualized on a KNIME WebPortal. There, the Next and Back buttons allows you to move back and forth between the word cloud of the selected author and the scatter plot, where the author is selected.

The final workflow can be seen in Figure 33 and is located on the EXAMPLES server in:

08_Other_Analytics_Types/04_Social_Media/02_NetworkAnalytics_meets_TextProcessing

The flow variable connection (the red line) between the wrapped metanode named “Tag Cloud” and the wrapped metanode named “Score & Sentiment on Scatter Plot” is required for the correct page flow on the KNIME WebPortal.

So, How Did We Do?

Posts and connections in a forum can be analyzed by reducing them to numbers, sentiment measures or influence scores. In this blog post, they have been reduced to a sentiment score via text processing on the one hand and to an authority/hub score via network graph analytics on the other. Both representations produce valuable information. However, the combination of the two has proven to be of invaluable help when trying to isolate the most positive and authoritative users for reward and the most negative and authoritative critics for damage control.

Chapter 5. Web Analytics

5.1. ClickStream Analysis

By Maarit Widmann, Anna Martin, Rosaria Silipo

Access workflow on hub.knime.com

Or from: EXAMPLES/50_Applications/52_Clickstream_Analysis/Clickstream_Analysis

Introduction

How long does it take before we finally decide to buy a product? How long do we read the product description? How many times do we come back to the same page in search of a convincing reason to buy? How many other product pages do we explore to compare?

We are not all the same when it comes to buying. There are the impulsive buyers, the buyers who need deep reflection before buying, the buyers who need comparisons to be convinced, and so on. We all follow our own buying path. And even more so when it comes to online shopping.

Clicks, visiting times, purchases, and other related actions are recorded on all websites. If you are just a guest, your actions are recorded anonymously. If you are a known customer, your actions are recorded in connection with your user ID. Anonymously or not, all of us leave a trail as we click our way from page to page.

Clickstream analysis is the branch of data science that collects, summarizes, and analyzes the mass of data from web visitors by detecting patterns and relationships between actions and/or visitors. With this knowledge, the online shop can optimize their service, including temporary advertisements, targeted product suggestions, better web page layout and improved navigation options.

When applying data science to online shopping data, we build a statistical or machine learning model on the data coming from the mass of shoppers. Single shopper behavior is not as interesting as the behavior of the whole group of similar shoppers. The mass of users/customers sharing a similar behavior with the current user will provide the necessary information for improvements and other actions. A product recommendation comes from hundreds (if not thousands) of similar shoppers; a behavior class groups together a large number of similar users, where the single user features are lost.

In this section, we focus on visualizing relationships across user data, time, and web activities, such as the number and duration of visits to the web page. These data aggregations and visualizations can make the basis to train a prediction model or investigate follow-up actions.

What You Need

Typically, a website log file contains information about the date and timestamp of the visit, the URLs that were visited, IP address, user location and optionally user ID. For registered users, the data are enriched by personal information such as age, gender, location, family status, and interests.

In this example, we use clickstream data provided by HortonWorks, which contains data samples of website visits stored across three files:

1. Data about the web sessions extracted from the original web log file. This file shows the user ID, timestamp, visited web pages, and clicks.
2. User data. This contains birthdate and gender associated with the user IDs, where available.
3. The third file is a map of web pages and their associated metadata, such as home page, customer review, video review, celebrity recommendation, and product page.

The Workflow

Figure 37 shows the complete clickstream analysis workflow, which is downloadable for free from the KNIME EXAMPLES server under EXAMPLES/50_Applications/52_Clickstream_Analysis/Clickstream_Analysis.

The workflow comprises three main parts: data preprocessing, data preprocessing for visualization and visualization.

Starting from the left - **data preprocessing -** the first part of the workflow provides data access, session identification, data cleaning, calculation of customer age, and purchase information.

The second part **prepares the data for the visualization**. From the top, we calculate session metrics by user age and gender. Then, we define the sequence of clicks (the clickstream) by session and its statistics. Lastly, we calculate the frequency of bounce pages by category.

The third part of the workflow on the right handles **visualization** - producing graphs and plots, visualizing the correlation between user data and session metrics, the sequence of clicks, and the page statistics.

Below, we describe each of these three sections in detail.

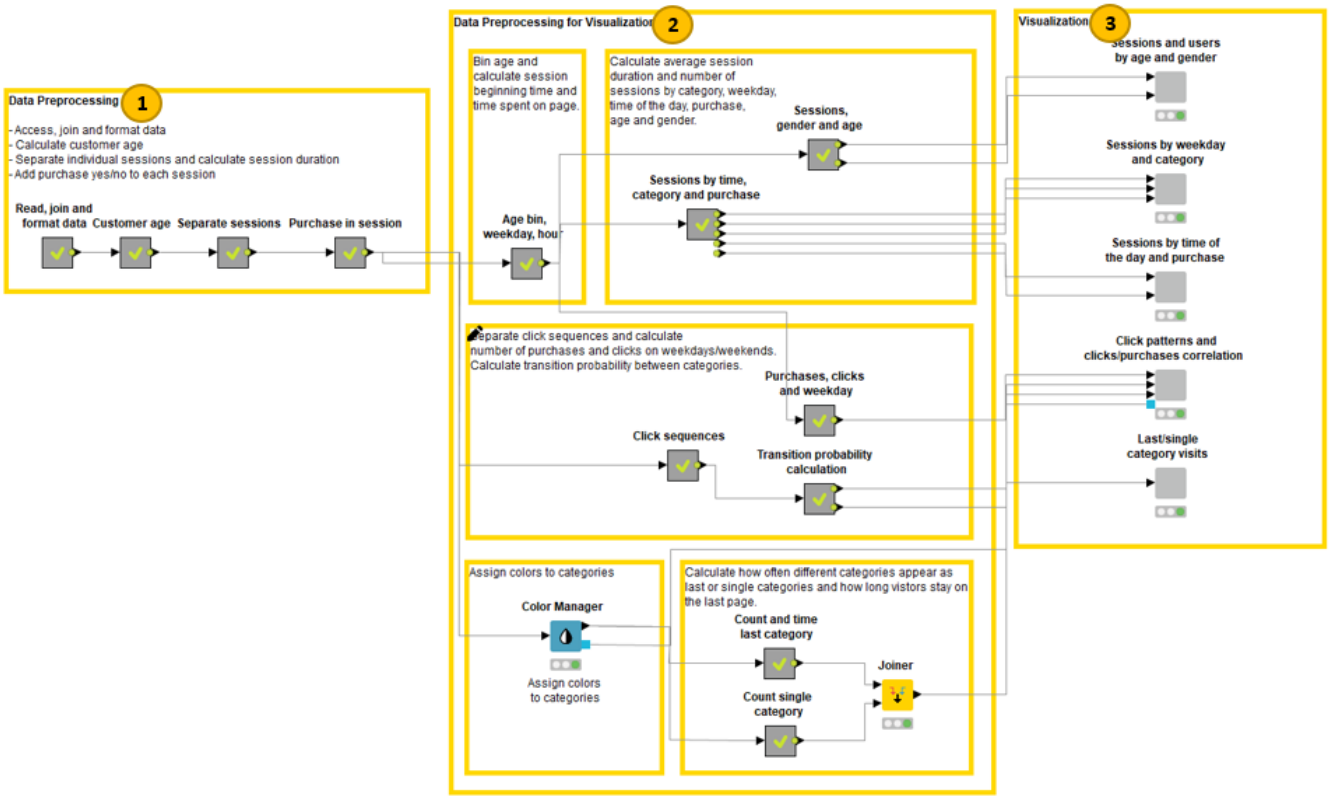


Figure 37. Workflow for Clickstream Analysis. From the left: data access, feature engineering, data preparation for visualization and visualizing clickstream data in interactive composite views

Preprocessing

Data Preprocessing

Here we read the data from the input files and perform some basic data cleaning and pre-processing.

All basic operations have been encapsulated into metanodes. A metanode is a gray node that contains a sub-workflow. Collapsing single nodes into a metanode keeps your workflow clean and tidy. Besides cleaning, organizing your workflow in logical blocks also has another purpose: metanodes can be saved as templates and reused across different workflows, similar to functions and macros in script based tools.

Let’s inspect the task handled by each metanode.

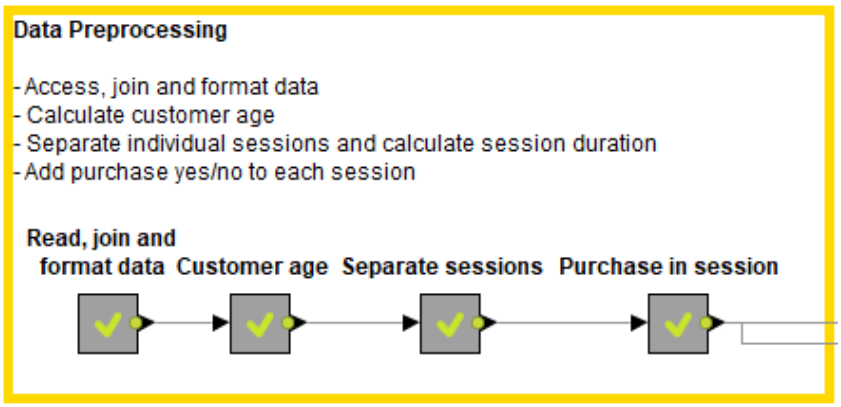


Figure 38. Accessing and Preprocessing Data for Clickstream Analysis

Read, join and format data:

- Accesses the data files with File Reader nodes
- Joins the session data with the user data and page categories

Customer age:

- Calculates the customer age based on the current timestamp and user birthdate

Separate sessions:

- Identifies single sessions based on the user ID and timestamps. A session is defined between website entry and logout or timeout. Each session contains one or more clicks.
- Calculates the time between subsequent clicks

Purchase in session:

- The website log file contains the information as to whether or not a product was purchased after a single click. A new column is then added showing whether the whole session included the product purchase at some point.

Data Preprocessing for Visualization

In this part of the workflow, we calculate a number of metrics based on page categories and features in user and session data.

Many of these calculations are completed inside the metanodes through classic nodes for data aggregation, namely GroupBy and Pivoting nodes. Let’s take a look at each metanode in more detail.

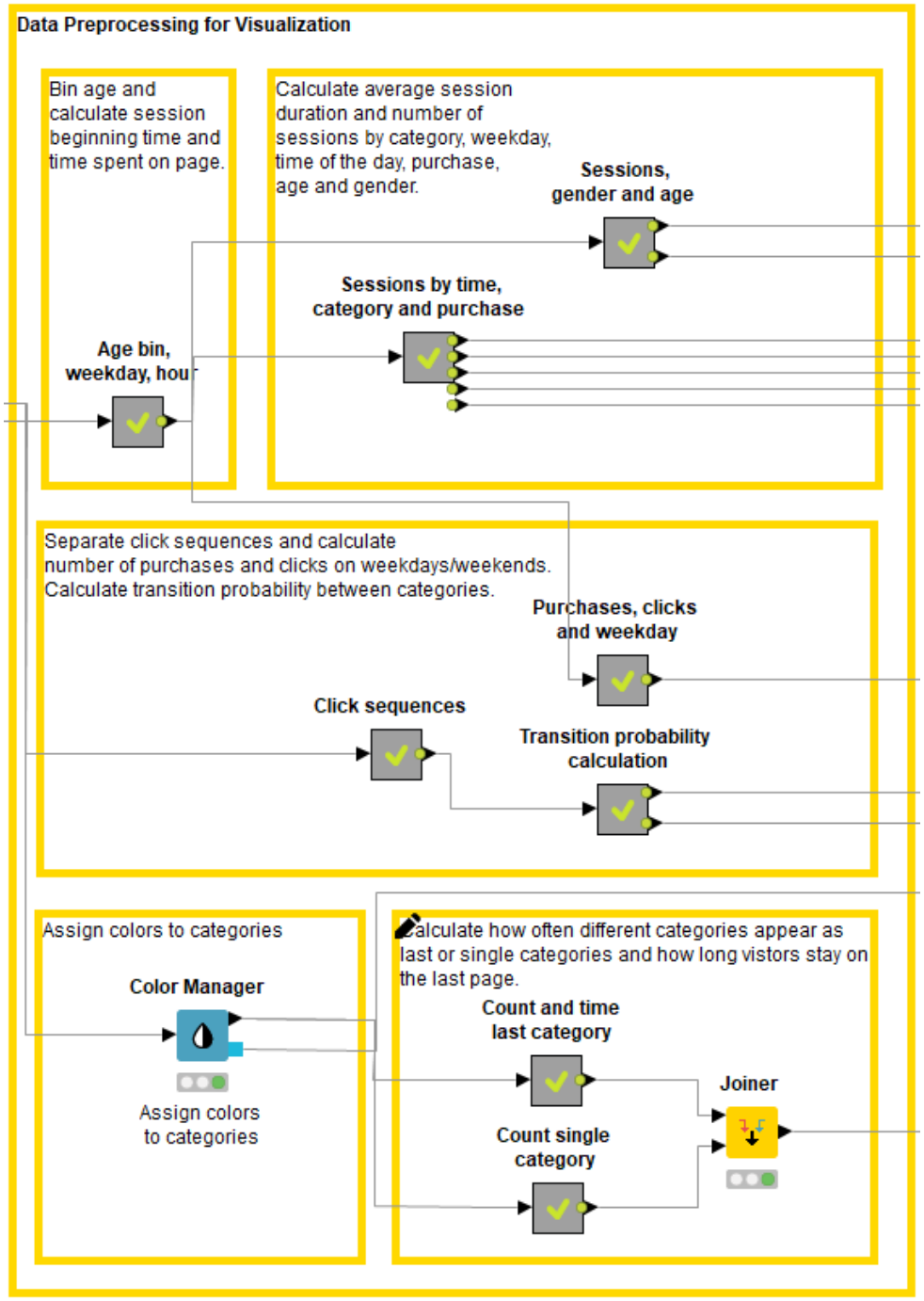


Figure 39. Preprocessing Data for Visualization

Age bin, weekday, hour:

- Bins customer age with the Numeric Binner node
- Calculates the session starting time, session duration, number of clicks and number of purchases according to session, category, user ID, age bin, gender and session purchase
- Extracts hour and weekday of single clicks with the Extract Date&Time Fields node

Sessions by time, category and purchase:

- Calculates the number of sessions and average session duration according to weekday, time of the day, web page category, and session purchase

Sessions, gender and age:

- Calculates the number of sessions and number of users according to gender and age

Click sequences:

- Concatenates categories occurring in each session
- Calculates the number of occurrences for each sequence of categories
- Creates columns for the category of the first, second, third, ... clicks with the Split Collection Column node

Purchases, clicks and weekday:

- Calculates the number of clicks and number of purchases per session and weekday

Transition probability calculation:

- Starts with a Column List Loop Start node and iterates over the columns created in the metanode *Click sequences*. In each iteration, pairs of columns containing categories of subsequent clicks are created.
- Concatenates the results from each iteration and calculates the transition probability for each pair of categories
- Tracks common click patterns by extracting click sequences occurring at least two times

Count and time last category:

- Calculates frequencies of categories as last pages in a session
- Calculates the visit time for each category given that it is the last category visited

Count single category:

- Calculates frequencies of categories as single visited pages

Visualization

In the data preprocessing steps, we created new features and aggregated the data by category, time, age bin and gender, along with some other features.

In this last part of the workflow, we build a few plots to visualize patterns of single and paired features as shown in the composite views (Figure 41, Figure 42, Figure 43, Figure 44) produced by the wrapped metanodes shown in Figure 40.

All visualizations are interactive since they were created with JavaScript based nodes. Views from JavaScript based nodes can be combined together in a single view by placing them in a single wrapped metanode. The composite view is available as the view output of the wrapped metanode and as a web page on the KNIME WebPortal.

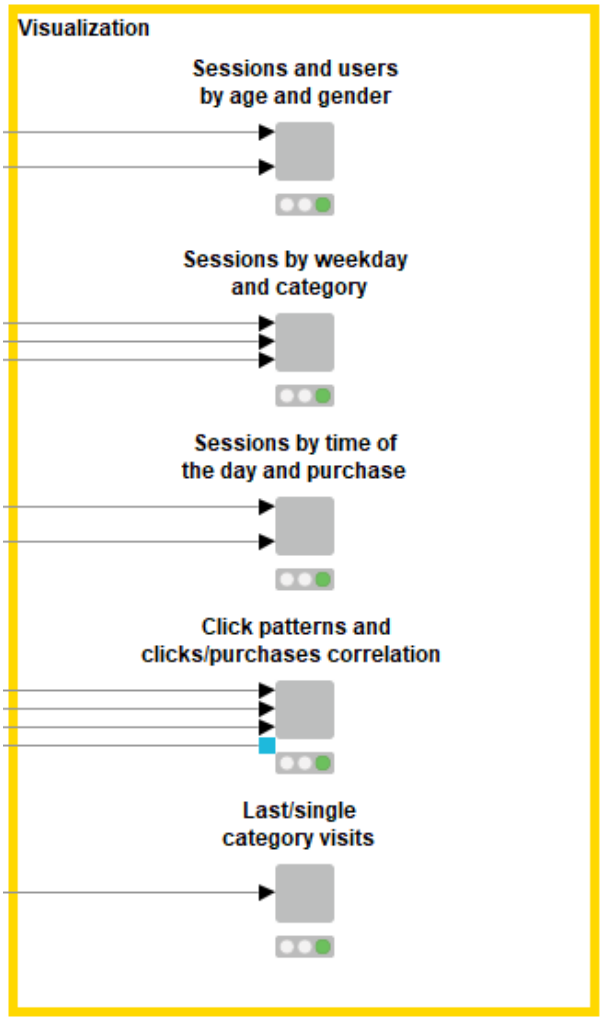


Figure 40. Visualizing Patterns and Relationships in Clickstream Data

The wrapped metanodes are listed below together with the JavaScript based nodes they contain and a description of their outputs.

Sessions and users by age and gender:

- Pie/Donut Chart (JavaScript) node to show the number of users by age bin
- Pie/Donut Chart (JavaScript) node to show the number of sessions by age bin
- Pie/Donut Chart (JavaScript) node to show the number of users by gender
- Pie/Donut Chart (JavaScript) node to show the number of sessions by gender

The composite view produced by this wrapped metanode is shown in Figure 41.

Sessions by weekday and category:

- Line Plot (JavaScript) node to show the average session duration according to category and weekday
- Stacked Area Chart (JavaScript) node to show the number of clicks according to category and weekday

The composite view produced by this wrapped metanode is shown in Figure 42.

Sessions by time of the day and purchase:

- Line Plot (JavaScript) node to show the normalized number of sessions with/without purchase according to time of the day
- Line Plot (JavaScript) node to show the normalized number of sessions with/without purchase according to weekday
- Bar Chart (JavaScript) node to show the number of sessions with/without purchase according to time of the day
- Bar Chart (JavaScript) node to show the number of sessions with/without purchase according to weekday

The composite view produced by this wrapped metanode is shown in Figure 43.

Click patterns and clicks/purchases correlation:

- Sunburst Chart (JavaScript) node to show the click patterns that occur at least twice
- Heatmap (JavaScript) node to show the transition probability from one category to another
- Scatter Plot (JavaScript) node to show the correlation between the number of clicks and number of purchases

A part of the composite view produced by this wrapped metanode is shown in Figure 44.

Last/single category visits:

Parallel Coordinates Plot (JavaScript) node to show the following statistics according to category:

- Count of different categories as last pages
- Count of different categories in sessions of one click
- Average time spent on the last page

You can take a look at the other graphs by executing the workflow available on the EXAMPLES Server at: EXAMPLES/50_Applications/52_Clickstream_Analysis/Clickstream_Analysis

What We Found

We have shown how to read, clean, transform, and visualize the clickstream data. Let's see now what we can discover from exploring our data visually.

User Activity According to Age and Gender

The first questions are always about demographics: How old are the people visiting the web page? Does the web page attract more women than men, or vice versa?

Figure 41 shows four pie charts. The pie charts on the top show the number of user IDs and number of sessions, respectively, according to the age bin. The pie charts on the bottom show the same statistics according to gender.

From the pie charts on the top, we can see the distribution of the four age bins is almost the same in terms of number of users and number of sessions: the users younger than 25 make up about one third of all users and users between 25 and 39 years a bit more than that. Since the number of users reflects the number of sessions, the activity of the users does not depend on their age. The number of sessions is about six times higher than the number of user IDs in each age bin, which means that each user visited the web page in average six times between the first and last timestamp in the web log file.

From the pie charts on the bottom, we can see that the web page has as equally many female and male users. Both genders are also equally active in terms of the number of sessions.

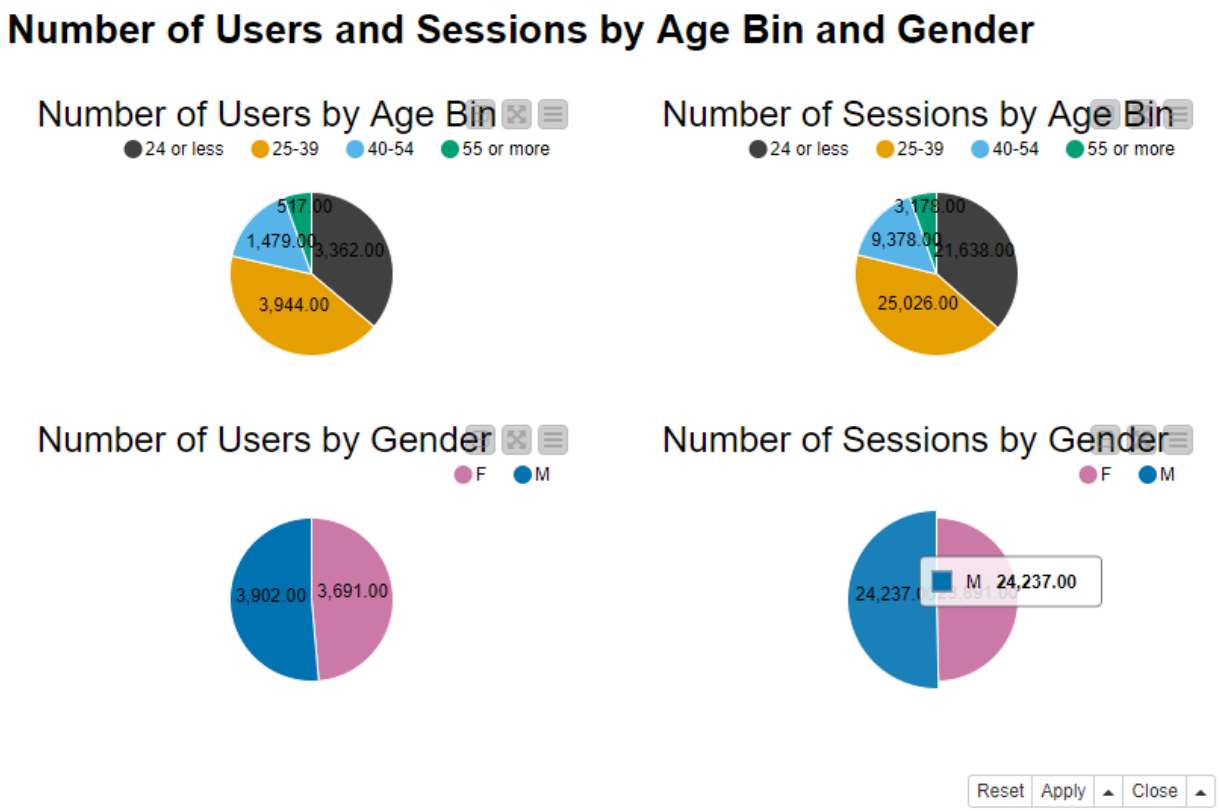


Figure 41. Visualizing number of users and sessions according to age bin and gender

Category Popularity during the Week

We know now that the majority of the users are younger than 40 and men and women are equally active. Next, we explore how the user activity depends on weekday or time of the day and web page category: Is the user activity different depending on which category they visit on which day? Are there more visits on recommendation and review pages at weekends when the users may have more time?

In both graphs in Figure 42, the horizontal axes show the weekdays from Sunday to Saturday. The different page categories define the colors. On the vertical axis, the line plot on the left shows the average session duration, whereas the stacked area chart on the right shows the number of clicks.

Let's start with the graph on the left showing the average session duration (in minutes) according to weekday and page category. On any weekday, the longest time is spent on product page, followed by home page and customer review. The two pages with the shortest session duration times are video review and celebrity recommendation. The page categories have their peaks on different weekdays, though: customer review pages have increasing visit times towards Friday, whereas the visit times increase slightly towards Saturday for all other categories.

If we take a look at the stacked area chart, we see that there is a peak on Monday and decreasing tendency towards Saturday for all categories. If we compare this chart to the line plot on the left, we can see that the number of clicks doesn't correlate with the average session duration in terms of most popular weekdays. It could be that many users just quickly check for new offers or reduced prices after the weekend.

Similar to the average session duration, the most popular categories in terms of the number of clicks are product page and home page, whereas the least popular category is celebrity recommendation. Possibly the web page provides only a few links to the celebrity recommendations. Or, the recommendations don't attract the majority of the users.

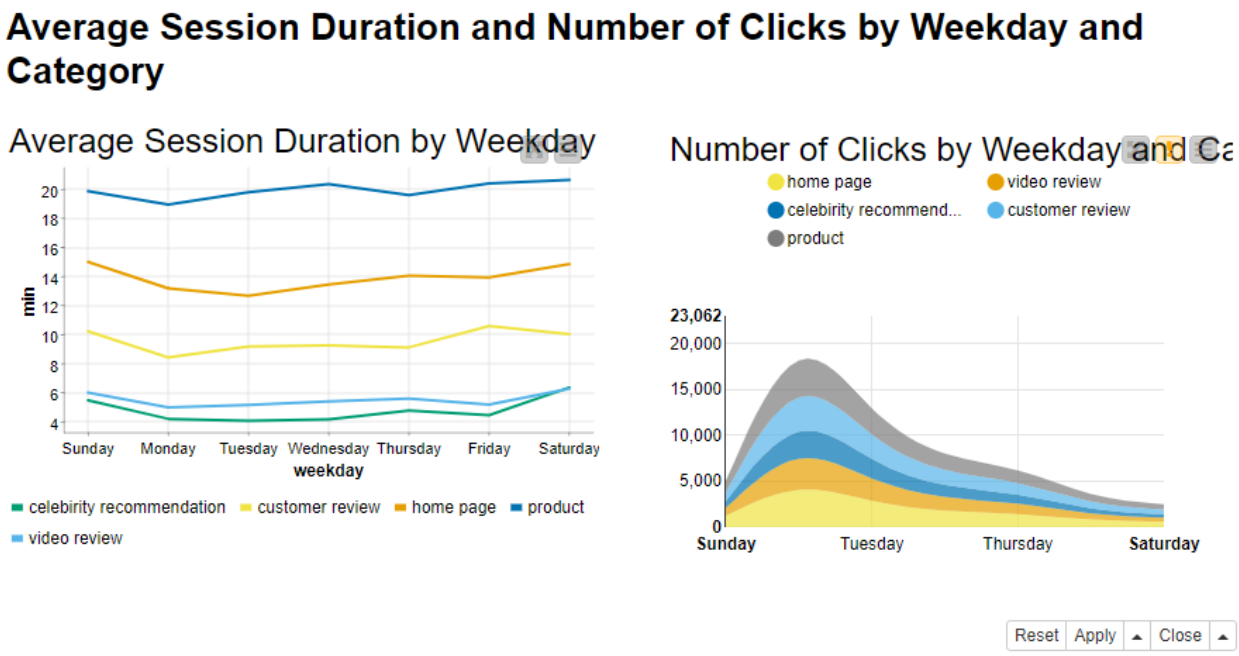


Figure 42. Visualizing average session duration in minutes and number of sessions according to weekday and page category

Purchases During the Day and Week

What we know so far is that some categories are more popular than others, and on some days there's more traffic on the web page than on the others. The majority of the users are younger than 40 years old and male and female are equally represented in the data. This information helps us even more, if we combine it with session purchase information. Does the number of sessions correlate with the number of purchases? Or does visiting the web page at unpopular times mean shopping "on purpose"? Let's check the graphs in Figure 43 describing the absolute and normalized number of sessions by time of the day, weekday and session purchase.

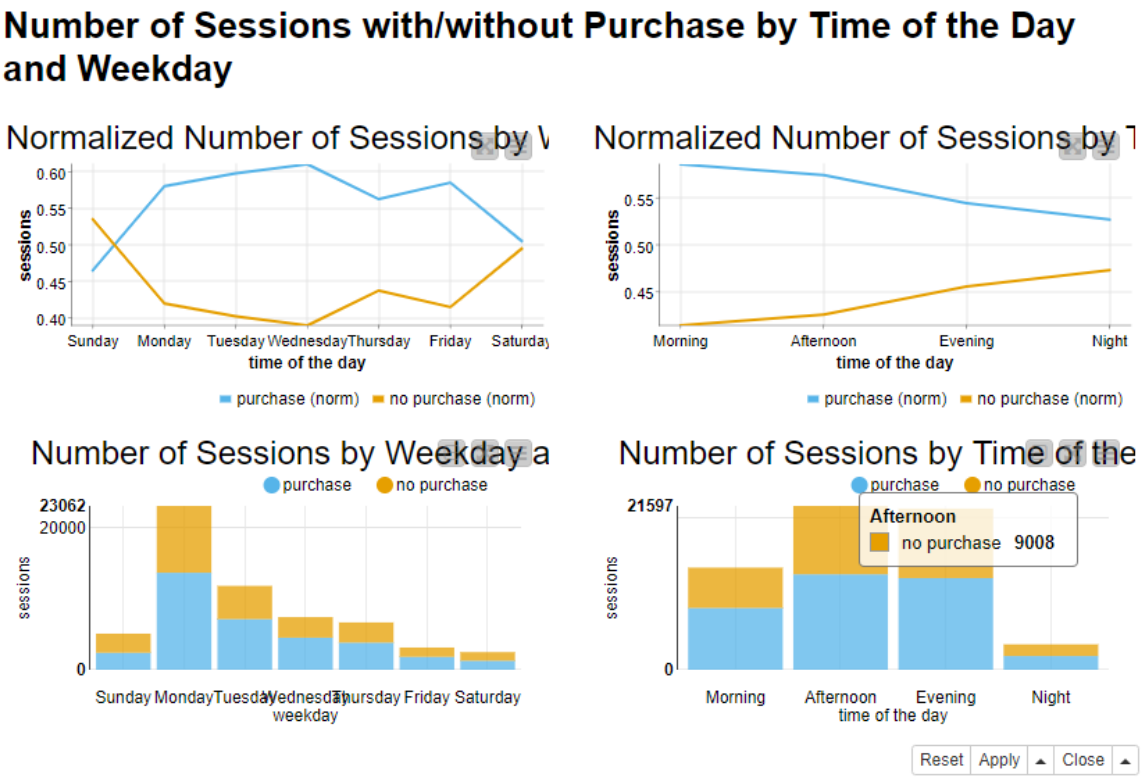


Figure 43. Visualizing number of sessions according to time of the day, weekday and session purchase

In all graphs in Figure 43, the session purchase information defines the colors: blue for "purchase" and orange for "no purchase". The line plots show the normalized number of sessions on the vertical axis. On the horizontal axis, the line plot on the left shows the weekdays from Sunday to Saturday, whereas the line plot on the right shows the times of the day. The bar charts show the absolute number of sessions on the vertical axis. On the horizontal axis, the bar chart on the left shows the weekdays, whereas the bar chart on the right shows the times of the day.

From the bar chart on the left, we can see that Mondays have the most sessions and the number of sessions decreases towards Saturday. Therefore, the number of sessions correlates with the number of clicks in terms of most popular weekdays, as we see in Figure 42.

From the line plot on the left, we see that the share of sessions without purchase is about 0.5 and therefore higher on weekends compared to weekdays. Hence, weekends tend to reduce not only the user activity on the web page but also the willingness to purchase a product. The share of the customers who purchase is the greatest on Wednesdays, when two of three users buy a product.

Let's now take a look at the user and purchase activity at different times of the day. The bar chart on the right shows a peak in the middle, that is, the most users visit the web page in the afternoon or in the evening. Supposedly the least sessions occur at night time. From the line chart above, we can see that the proportion of customers who purchase decreases towards night, but this proportion is approximately 0.5 at all times of the day. That is, in average a half of the users visiting the web page buy a product!

Click Patterns

Let's move on to a more detailed level. What happens during a single session? On which page do the users start? Can we detect a clickstream occurring especially often? How good is the home page in forwarding users to the product pages?

The graph on the left in Figure 44 shows typical click sequences in a session. Colors represent different page categories. The first clicks make the innermost donut. Possible next clicks are attached to each section on the previous donut level.

The orange and blue sections make almost 75 % of the innermost donut. This means that almost three of four visits start at either the home page or a product page. Both the orange and blue sections are divided in two - one part with further clicks and one part without. Therefore, about a half of the users starting a session on the home page or a product page end the session after the first click.

The graph on the right in Figure 44 shows the transition probability between two page categories. The row IDs are the categories of the first click, and the column headers are the categories of the next click. The color of each cell implies how probable the category defined by the column headers is to follow the category defined by the row ID - red for high, white for moderate and violet for low probability.

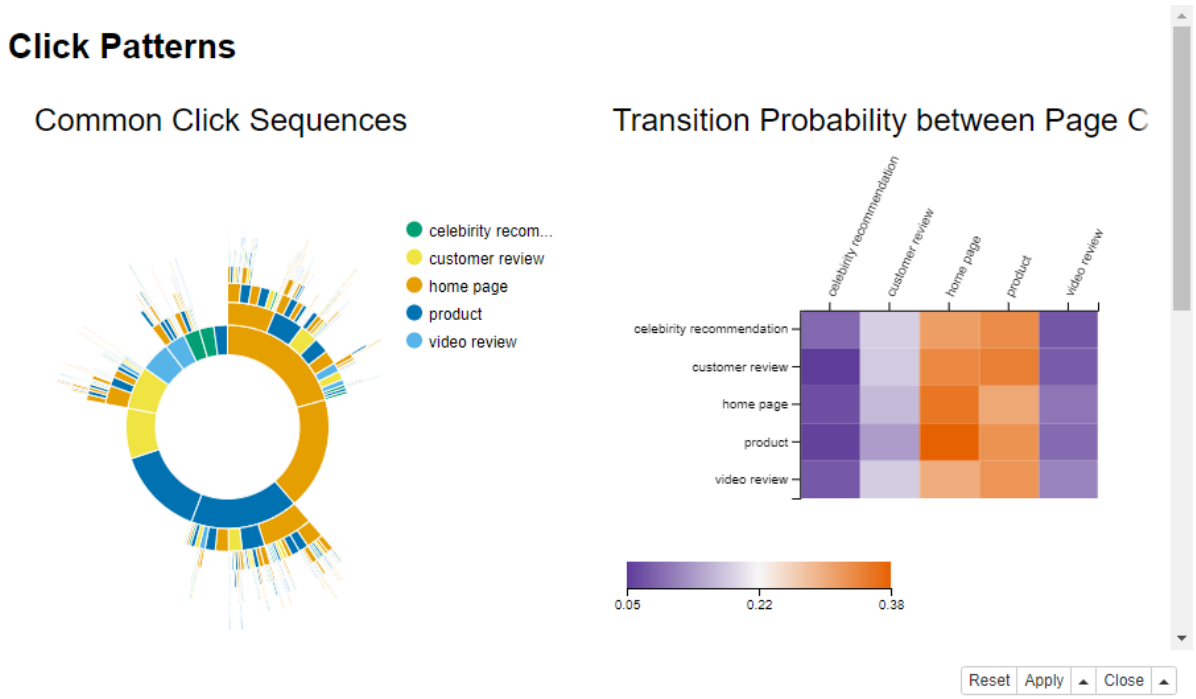


Figure 44. Visualizing typical click sequences and transition probability between two categories

The most probable next categories are home page and product page for all categories, shown by the red columns. Celebrity recommendation and video reviews represent the least probable next clicks for all categories. In general, the transition probabilities reflect the overall popularity of different categories.

Summary

So - who visits your web pages? At what kind of time and how do people get there? Did they buy anything? Applying clickstream analysis enables us to explore and detect patterns and relationships in the mass of data from visitors to the web. We can identify customer trends and analyze the different paths customers take, leading to products. The results, visualized in many different forms, give us the insight we need to enhance the users' experience on websites, increase customer retention, and - promote the decision to buy products!

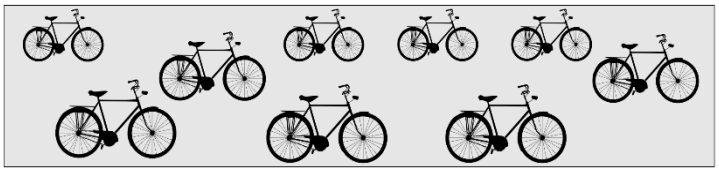
Chapter 6. IoT

6.1. Bike Restocking Alert with Minimum Set of Input Features

By Rosaria Silipo

Access workflow on hub.knime.com – Training Workflow and Deployment Workflow

Or from: EXAMPLES/50_Applications/55_Bike_Restocking_Alert



Demand prediction is a big topic in data analytics. The advantage is clear: knowing the demand in advance can help us better plan the offer. Going beyond demand prediction one step, you meet machine learning based alert systems. An alert system does not predict how many taxi, kilowatts, dinner dishes, beer bottles will be needed tomorrow at a specific hour, but more specifically will alert us when a restocking is required.

Defining an alert system is a simpler problem than demand prediction. Demand prediction has to predict the exact number of items in the future, while an alert system must only predict an unquantified need for adding or removing items to or from the basket. Alert systems are often found in IoT applications. An alert for changing the tires, the toothbrush, refilling the soap, or substituting mechanical parts saves us trouble, time, and money.

In this particular use case, we will be predicting the need for reshuffling (restocking or removing) bicycles at bike stations located throughout Washington DC. The main challenge of a bike share business is the timely restocking of bike stations. It is counterproductive for the business to have customers arrive at an empty bike station –when starting the trip, or to an overfilled bike station – when returning the bike. Both situations delay operations and create a bad customer experience.

What You Need

Capital Bikeshare is a bike service for tourists and residents to move around the Washington DC downtown area, therefore reducing traffic and pollution and promoting a healthier lifestyle.

Some years ago, Capital Bikeshare released a CSV dataset file containing a sample of bike trip history data. The original data included for each bike trip: duration; start date and time; end date and time; starting station name and number; ending station name and number; ID number of bike; membership type, i.e. whether "registered" (annual or monthly) member or "casual" (1 to 5 day pass) rider.

The original dataset has been previously aggregated in a separate workflow to move from each single trip per row to the number of bikes available at a given hour at each station. The final dataset, used in this use case, contains:

- Station information, as Terminal ID and station name
- Time information: Year, Day of Year, Day of Month, Day of Week, Hour

For that station, day, and hour:

- # bikes at station (Count)
- # total docks
- Number of bikes shuffled around: added “+” or removed “-“ (Shifted)
- Cumulative counting of bikes (Cumulative_Sum)
- Cumulative counting of bikes adjusted by the number of bikes added or removed (Adjusted_Cumulative_Sum)
- Performed Action (Flag): “Added bikes”, “Removed bikes”, “No Action”

This is the dataset we will use for this use case. The first 500 points of the time series, “Adjusted_Cumulative_Sum” (green), is plotted together with the “total number of docks” (blue) and with the number of shuffled bikes in column “Shifted” (red) against time (<Year> - <Day of Year> - <Hour>) on the x-axis in Figure 45 for bike station “Calvert & Biltmore St NW”.

“Calvert &Biltmore St NW” is a typical start station, in the sense that riders get a bike there to drive to a different station. The consequence is that this station is constantly depleted of bikes. Without the occasional injection of new bikes (red line), the station would not be able to offer a positive number of bikes (green line). Notice that the bike docks at the station are never completely full (the blue line never meets the green line).

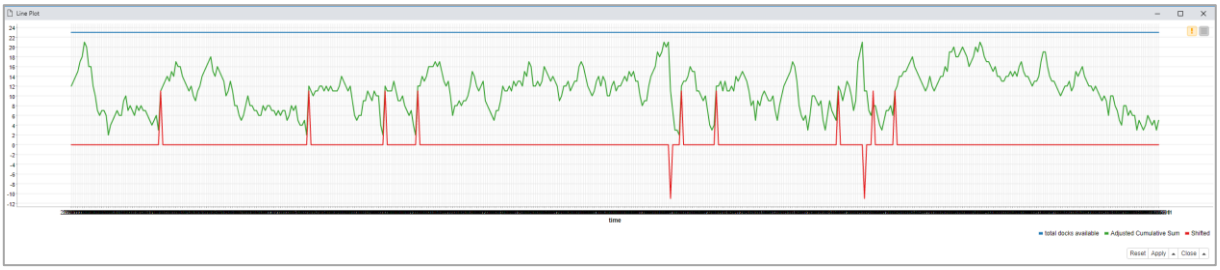


Figure 45. Time series “Adjusted_Cumulative_Sum” (green) is plotted together with the “total number of docks” (blue) and with the number of shuffled bikes in column “Shifted” (red) against time (<Year> - <Day of Year> - <Hour>).

Training Workflow

The goal of this use case is to predict the need of bike reshuffling (either adding or removing bikes) for each station within the next hour. Basically, we want to predict whether an action is needed in column “Flag” based on the data from one hour earlier.

Data Preprocessing

The first step then is to preprocess the data.

- Calculate the bike ratio for each station as *number of bikes available / number of docks available* (in the Math Formula node: *$Adjusted Cumulative Sum$/$total docks available$*). We believe that the ratio can better expose the need for bike reshuffling than just the disjointed number of docks and number of bikes currently available.
- Add calendar and weather information for the same dates from a second file.
- Define the target for the current hour as the value of Flag in the next hour. This is obtained in the “Build Past” metanode by looping on all stations and for the data of each station by shifting the values of Flag one step down with the Lag Column node. Flag(-1) becomes the new target column.
- Build a vector of N past bike ratios. In the same Group Loop in the same metanode, called “Build Past”, used to define the values in column Flag(-1), a second Lag Column node builds the vector of N=10 past values of bike ratios. Following the current philosophy that more data are better than less data, we insert some additional past values to the current bike ratio values. We will see later whether they are really needed or whether the bike ratio at the current hour was just enough for our prediction.
- There are around 200 stations in this dataset and we are considering a time window of 2 years (2011 and 2012) for a total of over one million data rows. Since in the next step we want to check how many and which of the input features are actually contributing to the classifier results, using the forward feature selection, we might need a lighter weight training set. In addition, most of the hours in the dataset are idle, nothing happens in terms of bike reshuffling. So, the number of data rows (hours) with Flag(-1) = “No Action” is definitely more frequent than the other two classes. Do we need all these hours where nothing literally happens? Shall we just reduce the dataset so as to have balanced classes and cut down on the most frequent class? This kind of sub-sampling is performed by the Equal Size Sampling node in the “Build Past” metanode.

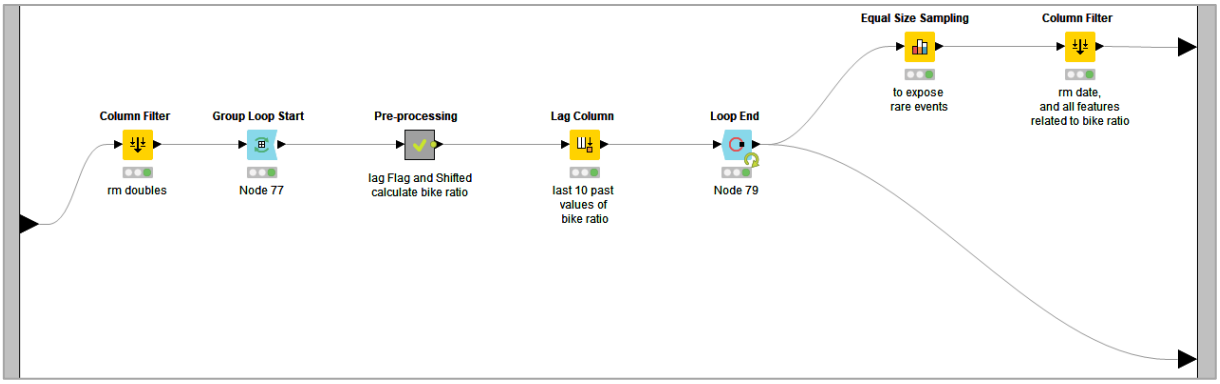


Figure 46. Content of the "Build Past"metanode. Station by station (as defined in the Group loop Start node) for each hour (each data row), the action for the next hour is defined in the "Pre-processing" metanode, and the vector of past 10 bike ratios is built in the Lag Column node.

Backward Feature Elimination

In the spirit of the time, we decided to overfeed our predictive model with more and more input features: weather and calendar features from an external files and past values of bike ratios. Are these additional input features really contributing to the overall success of the model? In order to discover that, we applied the Feature Selection with Backward Elimination procedure.

The "Backward Feature Elimination" metanode contains the loop that iterates on all input features and one by one removes the one least contributing to the model accuracy. The Feature Selection Loop End node offers a summary of the accuracy achieved and the number of features used at each iteration. The configuration window of the Feature Selection Filter node exposes such a summary and allows you to select the subset of the input data with a given number of features and a given accuracy. The model trained in this loop is a decision tree - for the sake of execution speed. As simple as it is, a decision tree can already give us an idea of the importance of the input features.

Notice that the highest accuracy (0.866) with the lowest number of features (4) is obtained with the Terminal ID, the bike ratio at the current hour, the bike ratio at the previous hour, and the current hour (Figure 47). All you need for a decent prediction is the current time (rush hour vs. quiet hours), the station (different stations busy at different times) and just a bit of history of the bike ratio values. All other features do not seem to add much to improving the accuracy. Actually, too many features seem to reduce the model quality in terms of accuracy value.

Our original of introducing more features for better results, and, in particular, introducing more past values, did not really pay off. Apparently, you do not need to go too much back in time to predict the bicycle situation in the next hour.

We proceed only with these four features: Terminal ID, the bike ratio at the current hour, the bike ratio at the previous hour, and the current hour

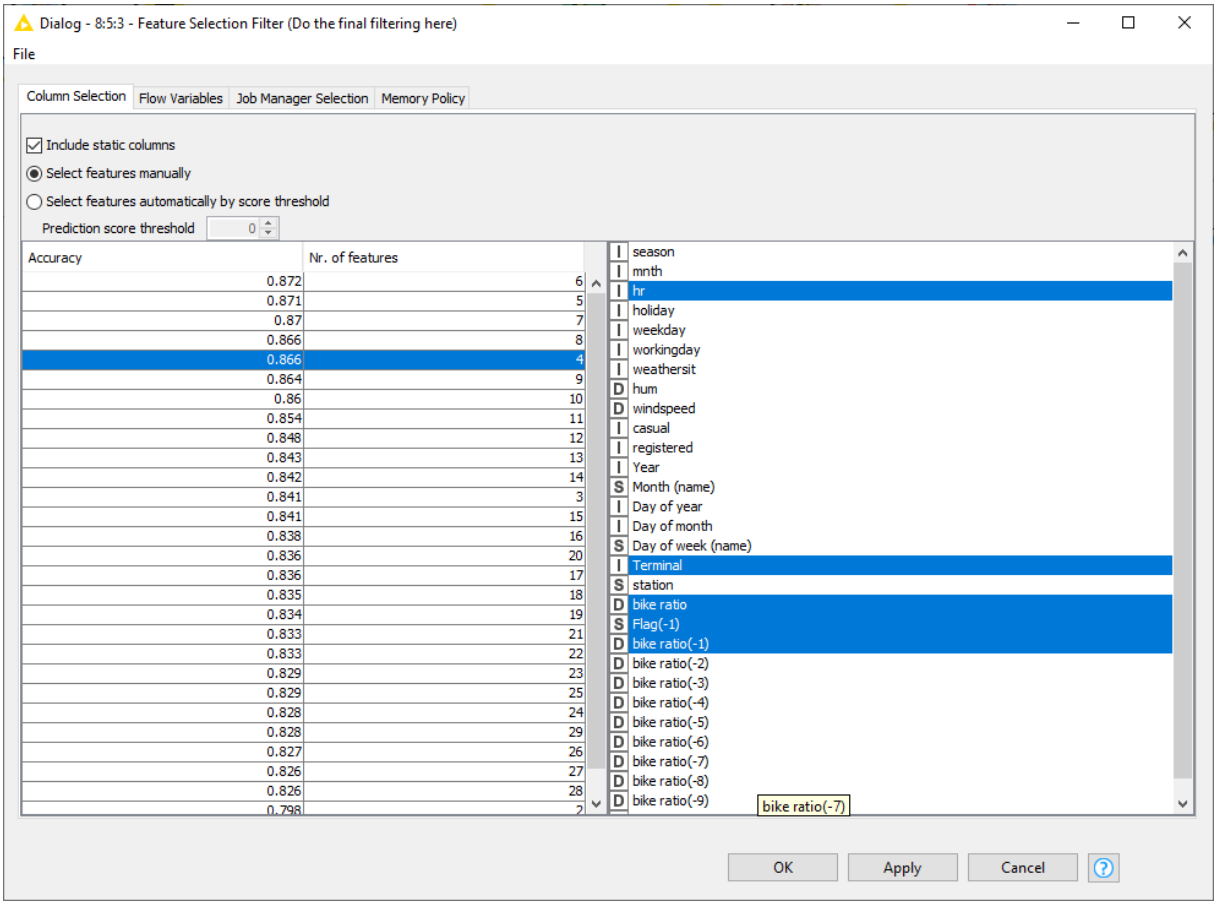


Figure 47. Configuration window of the Feature Selection Filter node following the Feature Backward Elimination loop. A close to best accuracy value is obtained with only four input features: Terminal ID, bike ratio at current hour, bike ratio at previous hour, and current time (hour).

Model Training and Evaluation

Now, we train a random forest model on these four inputs, with 10 decision trees on 90% of the randomly extracted data as training set to predict the need of bicycle reshuffling in the next hour (column *Flag(-1)*). With respect to the single decision tree used in the feature backward elimination procedure, this random forest performs slightly better, achieving 0.88 accuracy. The final model is then saved in a local zip file.

The final workflow is shown in Figure 48 and can be downloaded from the KNIME EXAMPLES server under: *50_Applications/55_Bike_Restocking_Alert/ 01_Bike_Restocking_Alert_Training_w_Feature_Selection* or from the KNIME Workflow Hub at: https://hub.knime.com/knime/workflows/*C0B4aJuqu_JShbYJ .

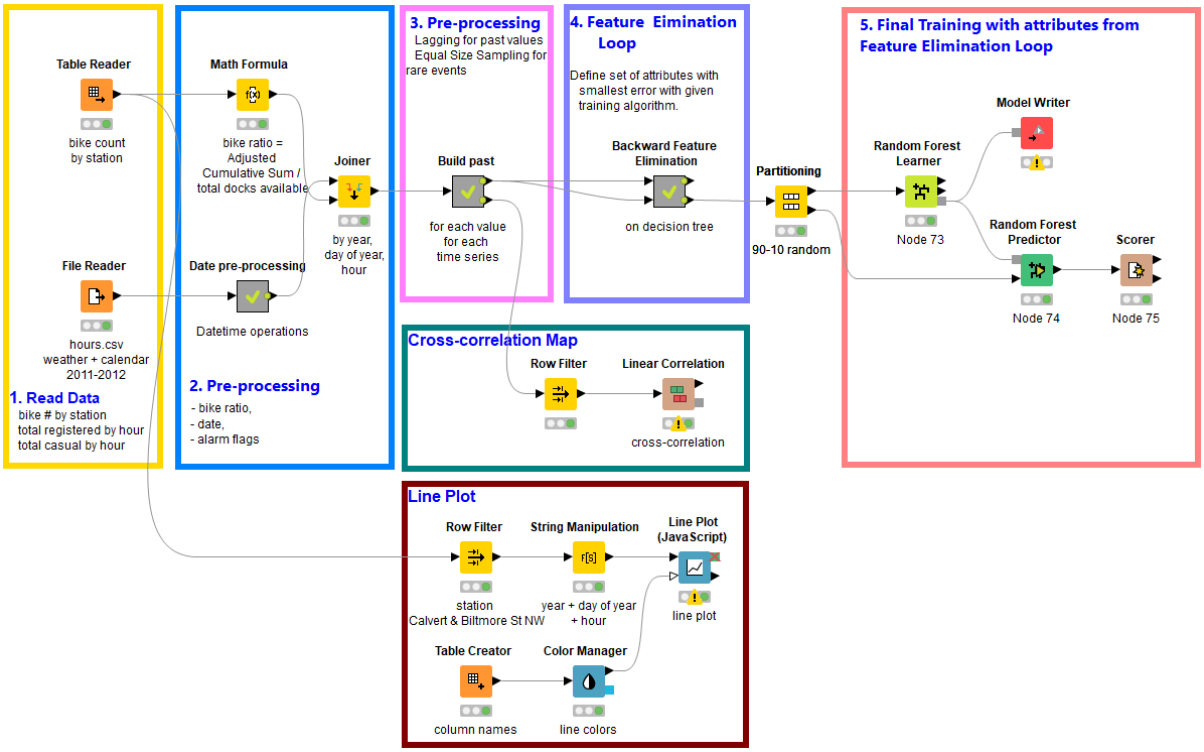


Figure 48. Training workflow 01_Bike_Restocking_Alert_Training_w_Feature_Selection. This workflow removes most of the input features (some joined from an external file and some previously generated) via the Feature Backward Elimination procedure. On the remaining input features a random forest is trained to predict the need for bicycle reshuffling in the next hour.

Deployment Workflow

The deployment workflow is supposed to generate alarms for bicycle reshuffling at each hour for different stations.

In workflow terms, the deployment workflow is supposed to read the current data with their history, read the model previously trained, apply the model to the new data, and generate some kind of a report/response with the prediction. Basically, like any classic deployment workflow.

The deployment workflow is shown in Figure 49 and is available for download from the KNIME EXAMPLES server at *50_Applications/55_Bike_Restocking_Alert/02_Bike_Restocking_Alert_Deploy* or from the KNIME Workflow Hub at https://hub.knime.com/knime/workflows/*NSLUQFTptcYKbVAM.

The data are read, joined again with the calendar and weather information from the external file, and preprocessed the same way as in the training workflow. The data rows for the different stations for the day and hour of interest are subsequently fed into the previously trained model, and the predicted status is produced as Remove Bikes, Add Bikes, No Action required. All stations requiring an action with a confidence higher than 0.7 are then transferred into a data table to be visualized on a web page via the KNIME WebPortal (Figure 50).

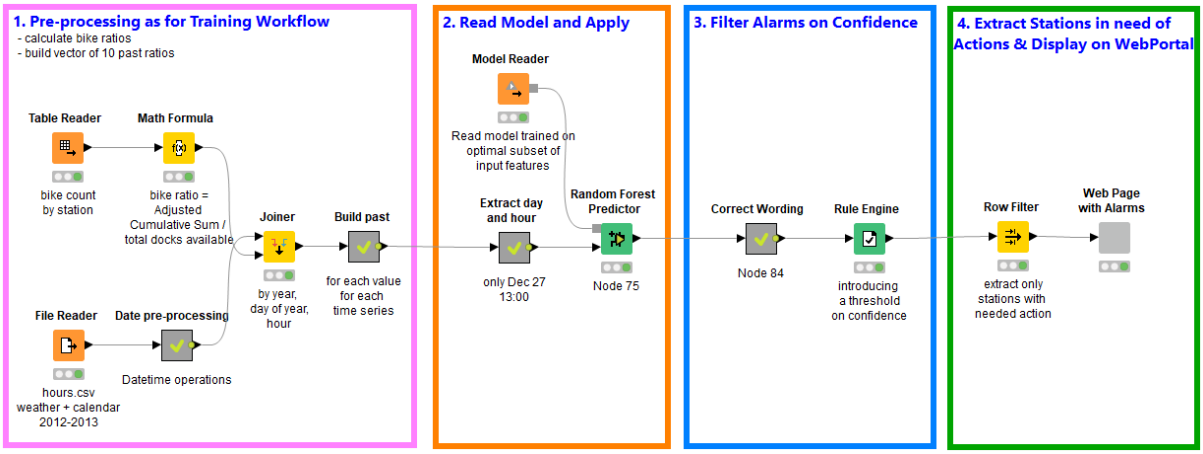
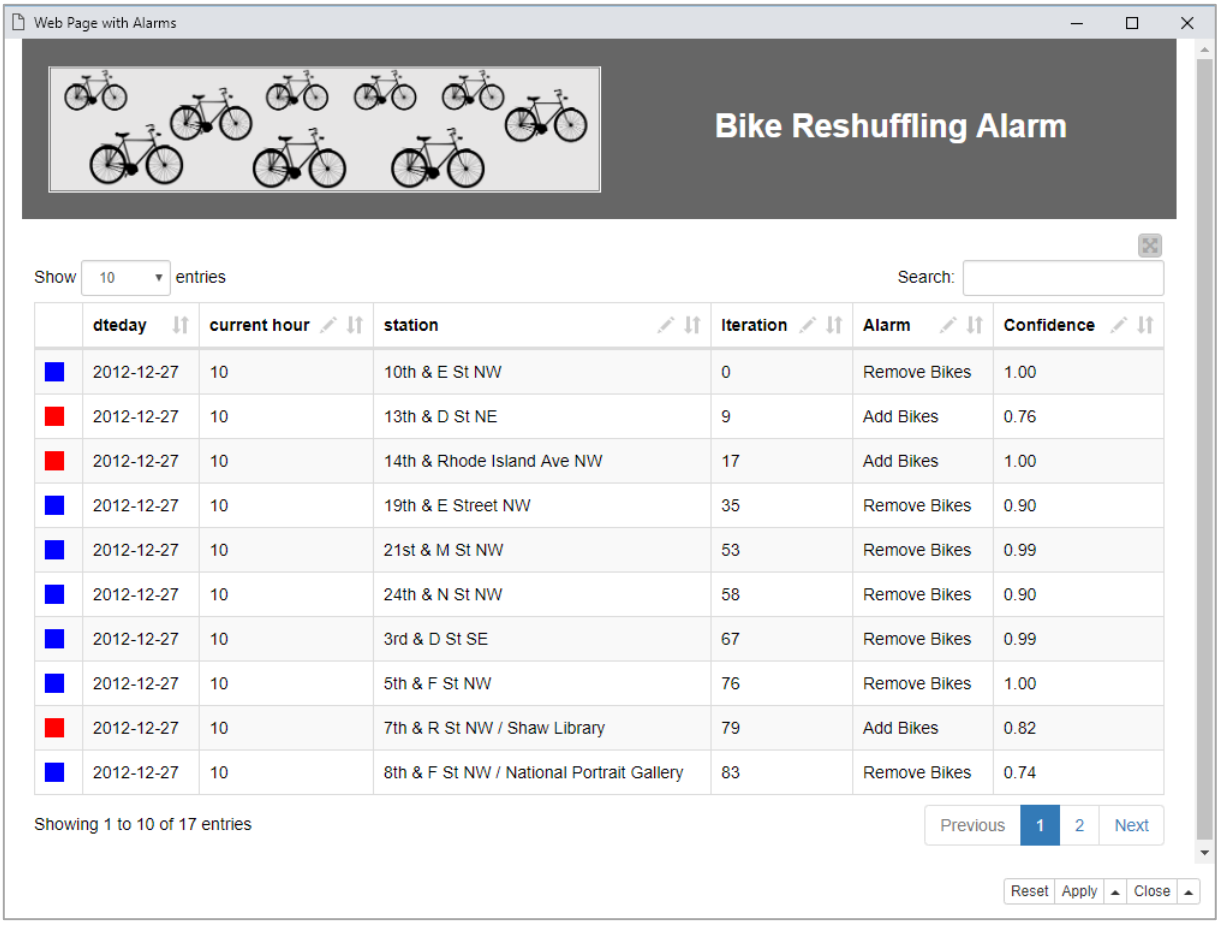


Figure 49. Deployment workflow 02_Bike_Restocking_Alert_Deploy. The workflow produces the predicted actions for all stations for that day and that hour. Stations requiring an action with a confidence higher than 0.7 are shown in a table on the KNIME WebPortal.



	dteday	current hour	station	Iteration	Alarm	Confidence
	2012-12-27	10	10th & E St NW	0	Remove Bikes	1.00
	2012-12-27	10	13th & D St NE	9	Add Bikes	0.76
	2012-12-27	10	14th & Rhode Island Ave NW	17	Add Bikes	1.00
	2012-12-27	10	19th & E Street NW	35	Remove Bikes	0.90
	2012-12-27	10	21st & M St NW	53	Remove Bikes	0.99
	2012-12-27	10	24th & N St NW	58	Remove Bikes	0.90
	2012-12-27	10	3rd & D St SE	67	Remove Bikes	0.99
	2012-12-27	10	5th & F St NW	76	Remove Bikes	1.00
	2012-12-27	10	7th & R St NW / Shaw Library	79	Add Bikes	0.82
	2012-12-27	10	8th & F St NW / National Portrait Gallery	83	Remove Bikes	0.74

Figure 50. List of bike stations currently requiring an action, as output by the deployment workflow.

6.2. Taxi Demand Prediction using Random Forest on Spark

By Andisa Dewi & Tobias Kötter

Access the workflows on hub.knime.com – Training Workflow & Deployment Workflow

Or from: EXAMPLES/10_Big_Data/02_Spark_Executor

Introduction

Taxis play a big part in a city's life. In megacities such as New York, more than 13,500 yellow taxis roam the streets every day [1]. This makes the task of understanding and anticipating taxi demand a crucial task for taxi companies or even city planners, to increase the efficiency of the taxi fleet and minimize waiting time between trips.

In this use case, we will use the NYC taxi dataset and a Random Forest to train a simple time series prediction model to predict taxi demand in the next hour based on data from past hours. For better scalability, we will train and test the model on a Spark cluster.

What You Need

The NYC taxi dataset, which can be downloaded at the NYC Taxi and Limousine Commission (TLC) website, spans over 10 years of taxi trips in NY city with a wide range of information about each taxi trip, such as pick-up and drop-off datetimes, locations, fares, tips, distances, passenger counts, etc. For this study we concentrate on the Yellow taxi dataset for the year 2017.

Given the large size of the dataset, we would like to train and deploy the machine learning model of choice on a Spark cluster. The KNIME Big Data Extension allows you to run a KNIME workflow on the big data platform you prefer, via in-database processing or via Spark. The KNIME Big data Extension can be installed like all other KNIME Extensions from within KNIME Analytics Platform. Check the video "How to install KNIME Extensions" if you want to know more.

Once the KNIME Big Data Extension is installed, you will see special categories in your Node Repository. For example database connector nodes for big data platforms (Hive, HP Vertica, Impala, etc…), nodes to create a Spark context, and other Spark compatible nodes. The Spark category includes a set of nodes for scalable data mining, scoring statistics, machine learning algorithms, and many more running in Apache Spark.

These nodes have all been developed using a similar GUI to the corresponding nodes in KNIME Analytics Platform. For example, the Spark GroupBy node has the same configuration window as the GroupBy node. This of course makes our life much easier, since we can take advantage of the scalability of the big data platform without having to learn a new script.

I would like to draw your attention to one node in particular: the Create Local Big Data Environment. This node allows you to create a simple Hive and Spark cluster locally on your machine. It can be easily used to prototype, or whenever you have no access to the real cluster.

Preprocessing

The goal of this use case is to predict the taxi demand in NY City at the current hour. In order to run this prediction we need the taxi demands from previous hours. Translating this into numbers, the taxi demand in NY City for a given hour is the number of taxi trips during that hour. So, the first step in data preparation is to calculate the total number of taxi trips in New York City for each hour of every single day. This means we have to:

- Separate the pickup_datetime of each taxi trip into year, month, day of month, day of week, and hour
- Group the data by the hour and the date (year, month, day of month), and count the number of rows (i.e. the number of taxi trips), using the Spark GroupBy node
- Store the resulting data set in a /data folder, ready for use!

Data Exploration

Line Plot

The easiest way to explore a time series is always a line plot, with the time on the x-axis and the values on the y-axis. Figure 51 shows this line plot for just two weeks of the data, in June 2017. The line plot shows a clear cycle pattern every 24 hours, i.e. a 24-hour seasonality. Weekly seasonality seems to be present as well, as there is a lower tendency to take taxis on the weekend.

On a larger scale, we can also observe a winter-summer seasonality pattern, with people taking the taxi more often in winter than in summer. However, since we consider the data over only one year, we will ignore this yearly seasonality.

The line plot also offers us a peek into city or weather-related events. For example, the really low drop of taxi trips around March turned out to be due to a blizzard bringing the city to a stop and the high jump in the beginning of November was probably caused by the famous NYC Marathon.

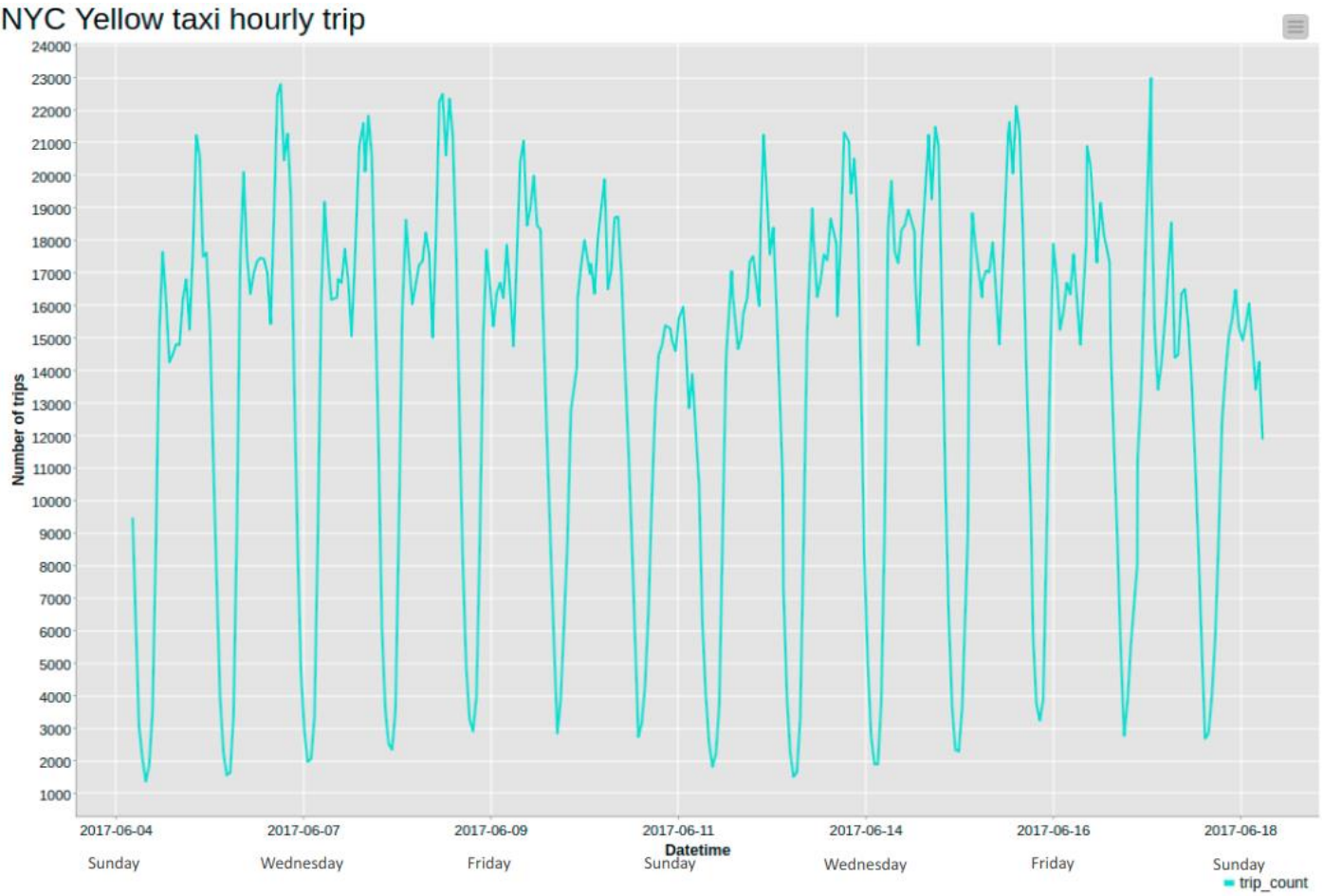


Figure 51. Zoomed in on the first two weeks of June 2017. You can see here the 24-hour and weekly seasonality.

Auto-correlation

From the line plot we have gathered a sense for daily seasonality. Let's confirm this feeling with the auto-correlation map. Auto-correlation is the linear correlation between the value of the time series at time *t* and its past values at *t-1 , .., t-n*.

Thus, in order to create the auto-correlation map we need to create the vector of values at times *t, t-1 , .., t-n*. We create these vectors using a lag=n. Afterwards we calculate the correlation coefficients (Pearson correlation coefficients) among all vector components using the Spark Correlation Matrix node. In general, values between -0.5 and 0.5 would be considered low correlation, while coefficients outside of this range (positive or negative) would indicate a high correlation.

The resulting correlation matrix among the number of trips (trip_count) at different hours (up to n=50) is visualized by a Javascript Heatmap node. In the Heatmap, the stronger the blue shade (higher coefficient) the more the corresponding two variables are positively correlated, and vice versa for the red color (lower coefficient with negative correlation). As we can see in the heatmap, there is a very high positive correlation (0.91) of taxi demand in the current hour with the taxi demand in the immediate past hour (lag=1). More importantly the next highest correlation of taxi demand in the current hour is with the value in the previous 24th hour, the next highest coefficient with the value in the previous 48th hour, etc … indicating a 24h (daily) seasonality.

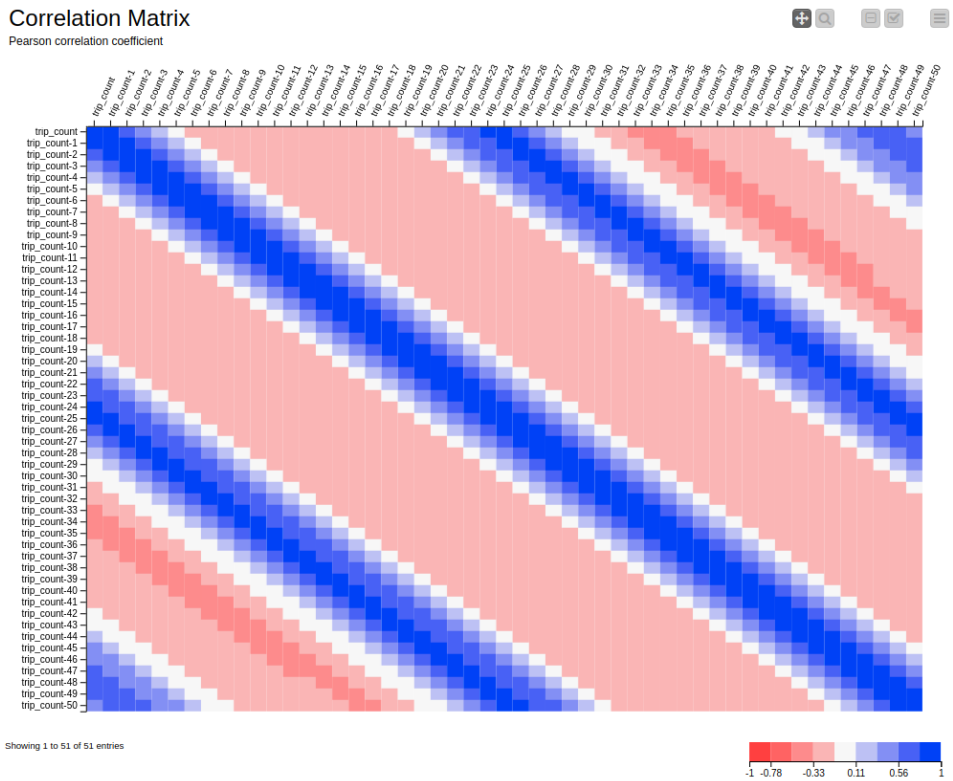


Figure 52. Auto-correlation Matrix (Pearson Coefficients) over 50 hours. The strongest correlation is with the number in the previous hour, in the previous 24th hour, 48th hour, and so on, indicating a daily seasonality.

To run the taxi demand prediction, we decided to use a Random Forest on a Spark cluster. Random forests are very powerful algorithms and running them on a Spark cluster will avoid scalability issues.

The first step as always is to load the dataset. Because we are running everything on a Spark cluster, the Create Local Big Data Environment node comes in handy if we don't have access to an Apache Hadoop cluster right away. This node creates a functional local big data environment right on our machine. If you do have a Hadoop cluster running, with Livy installed, and want to execute the workflow on it, then simply replace this node with a Create Spark Context via Livy node and add the cluster information in the node dialog. The dataset has been previously loaded on the big data platform of choice (local or remote) in the shape of a Parquet file. The Parquet to Spark node loads the dataset onto a Spark dataframe.

The dataset is then partitioned into training (January to November 2017) and test set (December 2017). Notice the split in time. We do not use a random partitioning, but a sequential split in time. This is to avoid data leakage from the training set into the test set in a time series prediction problem, where time is an important variable. This all happens in the metanode named “Split by date and time”.

The Find Lag metanode finds the optimal value of lag n by first creating a certain number of lagged columns (we use an initial lag=50). After creating the lagged columns, the auto-correlation coefficients among all lagged components are calculated. The lag corresponding to the second highest correlation coefficient value is selected as the optimal lag. In our case lag=24 was found as the optimal lag, which corresponds to what found by visual inspection of the auto-correlation matrix in Figure 52.

The “Spark Lag Column” produces a number n of lagged columns from the original column of total number of trips (taxi demand) per hour. The original column is copied and lagged 1, 2, 3, .. n steps via a SQL query. The metanode allows us to set 4 parameters in its node dialog: lag value *n* to create the vector of past and current values, default value to use in case of missing values, name of original column to lag, and name of time column to sort the time series values.

With these four parameters, we build *n* SQL queries, each query creating one lagged version of the original column using the Empty Table Creator and the String Manipulation nodes. The n SQL queries are concatenated together into one big SQL query with the Concatenate and the GroupBy node. The final SQL query is executed with the Spark SQL Query node. At the end, the Spark Missing Value node is applied to handle possible missing values. Figure 54 shows the content of the “Spark Lag Column” metanode.

The Training Workflow

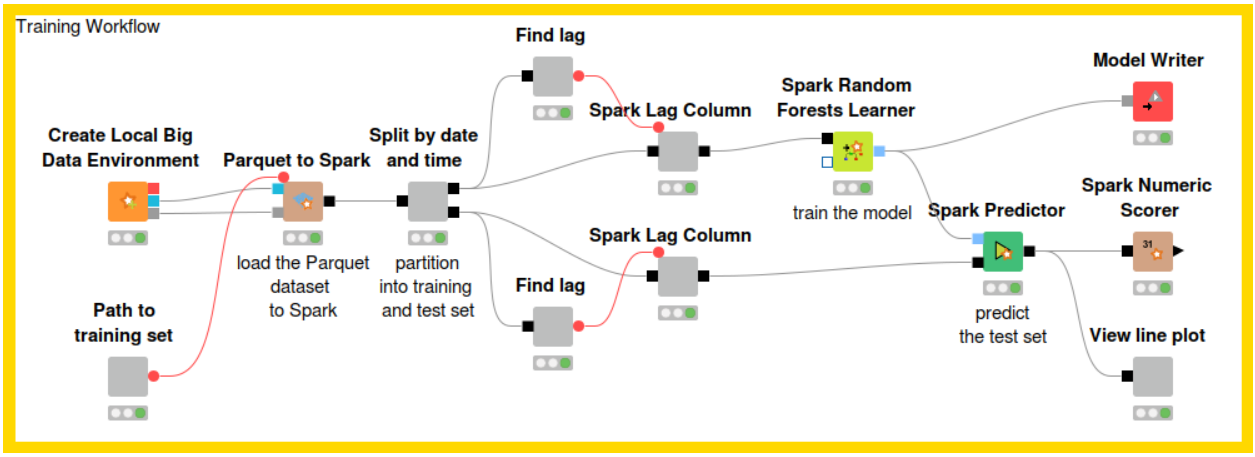


Figure 53. The training workflow

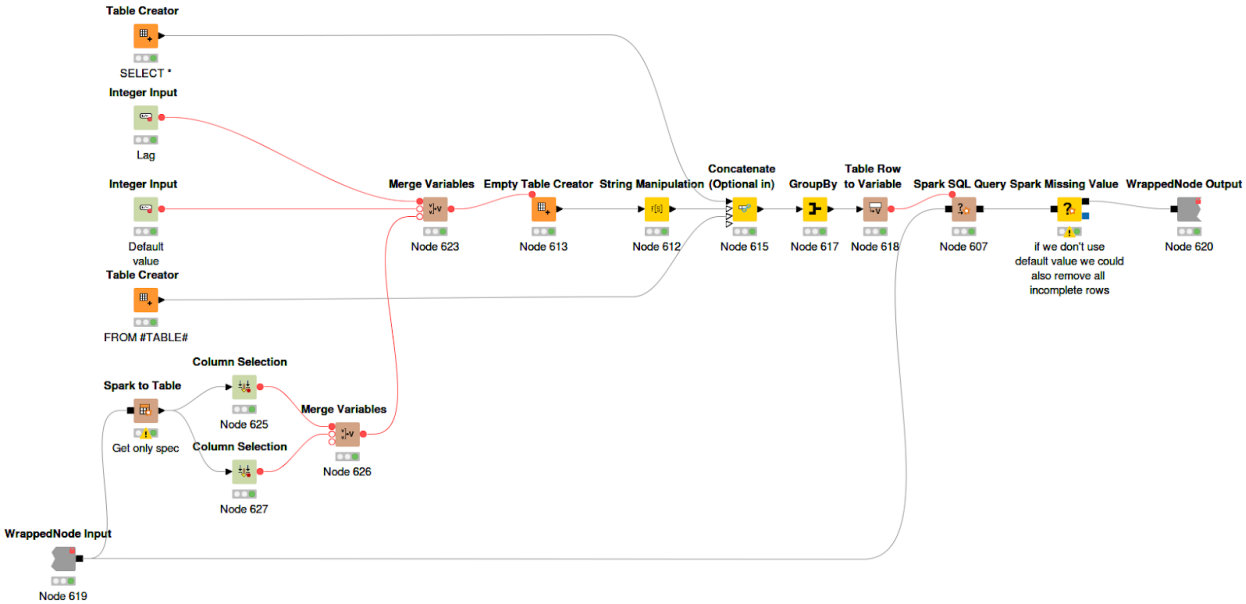


Figure 54. Content of the Spark Lag Column metanode. The node produces n lagged columns from the original taxi demand column.

The next step is to feed the n=24 lagged columns into the Spark Random Forests Learner node to train the random forest using the taxi demand values over the past 24 hours. Additionally, two more temporal features are added: the hour of the day (0-23) and the day of the week (1-7). Default values for the hyperparameters are used for the random forest: 5 trees with maximal depth 10.

After training, the 5 trees produced by the random forest all implement a similar decision path. In all of them, the top split happens on feature trip_count-1, that is the taxi demand in the previous hour (lag=1). This was also the feature with the highest correlation coefficient with trip_count in the correlation map. The split happens on the pickup hour, which provides information on daily human activities, like rush hour, for example.

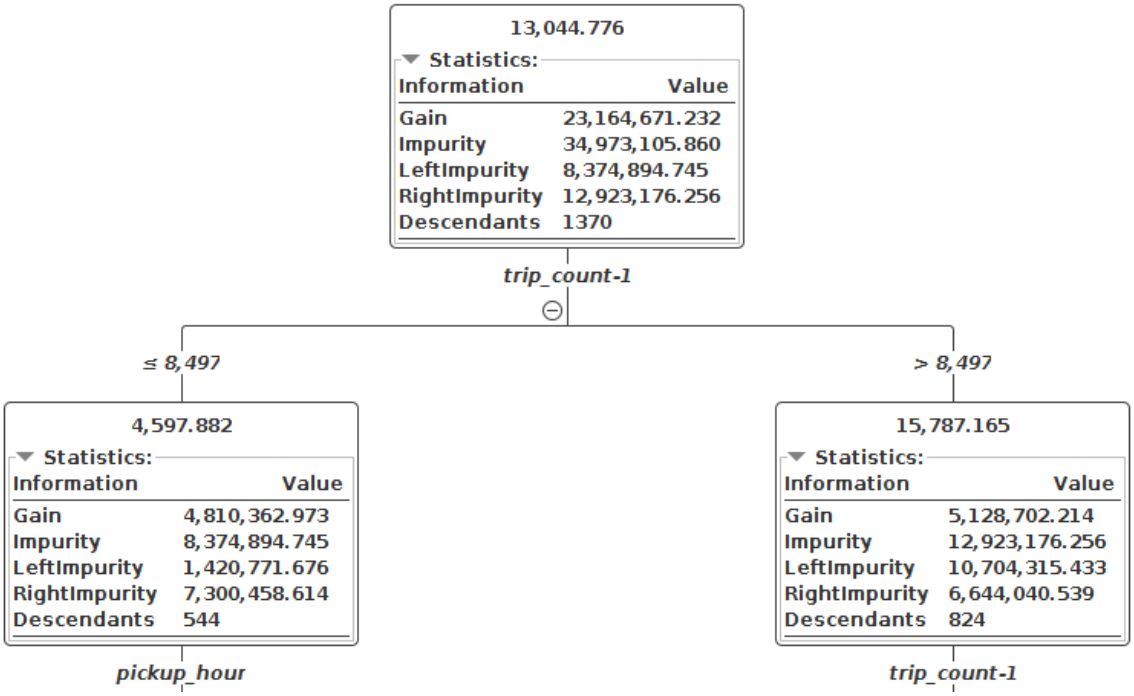


Figure 55. Tree #1 in the Random Forest model (top part). All other 4 trees of the forest follow a similar decision path. Top features seem to be the taxi demand in the hour immediately before the current hour and the time of the day.

Testing the Model

Now it's time to test the model. We use the data from December 2017 as our test set. The data preparation steps are the same as for training.

- A number of lag columns is created in the "Spark Lag Column" metanode based on the optimal value given by the "Find Lag" metanode.
- The lagged columns are then fed into the Spark Predictor node together with the model trained earlier on. The predictor node's output will be the predicted values for taxi demand in the current hour.

As this current case study is a time series prediction problem, the model performance is evaluated calculating a numeric distance between the predicted numbers and the real numbers. This is what the Spark Numeric Scorer node implements through a few different distance measures, such as Mean Absolute Error, R^2, Mean Squared Error or Root Mean Squared Error, and Mean Signed Difference. We opted for the R^2 measure to evaluate the quality of the model's prediction. We arbitrarily set the threshold for model acceptance to a R^2 error of 0.9.

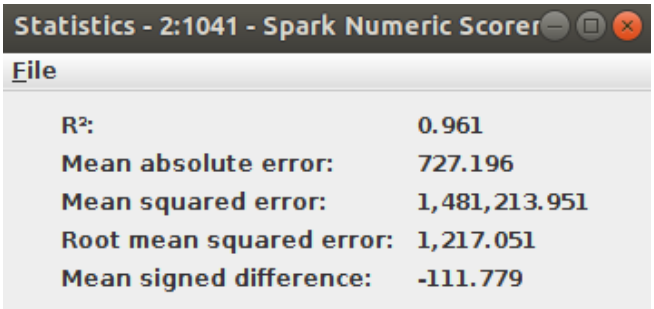


Figure 56. Error measures on predictions of the trained model for the data in the test set, as calculated with the Spark Numeric Scorer node

It is common practice to remove the seasonality pattern before attempting to train a model on a time series. The idea is that predicting the residuals after removing seasonality will hold better predictions. However, we have noticed that for regular time series often a highly parametric algorithm like a random forest produces good results even if trained on the full time series - without seasonality removal. In any case, our acceptance criterion (R^2 > 0.9) has been met and for now we will not try to train the model on the residuals only. The trained model is now stored to a file using the Model Writer node for further usage in deployment.

In Figure 57 you can see the predicted (yellow) and actual (light blue) values of the taxi demand time series that have been plotted. Indeed, the model seems to fit the original time series quite well. For example, it is able to predict a sharp decrease in taxi demand leading up to Christmas time.

Looking again at Figure 51, this hourly taxi demand time series seems to be highly regular and seasonal with a trend that probably can be classified as secular (tendency to grow or decline over a long period of time). It means that even though this simple model is trained only on 11 months data in the year 2017, we can still use it to predict new short-term demands for the next couple of years, provided the trend doesn't change too much.

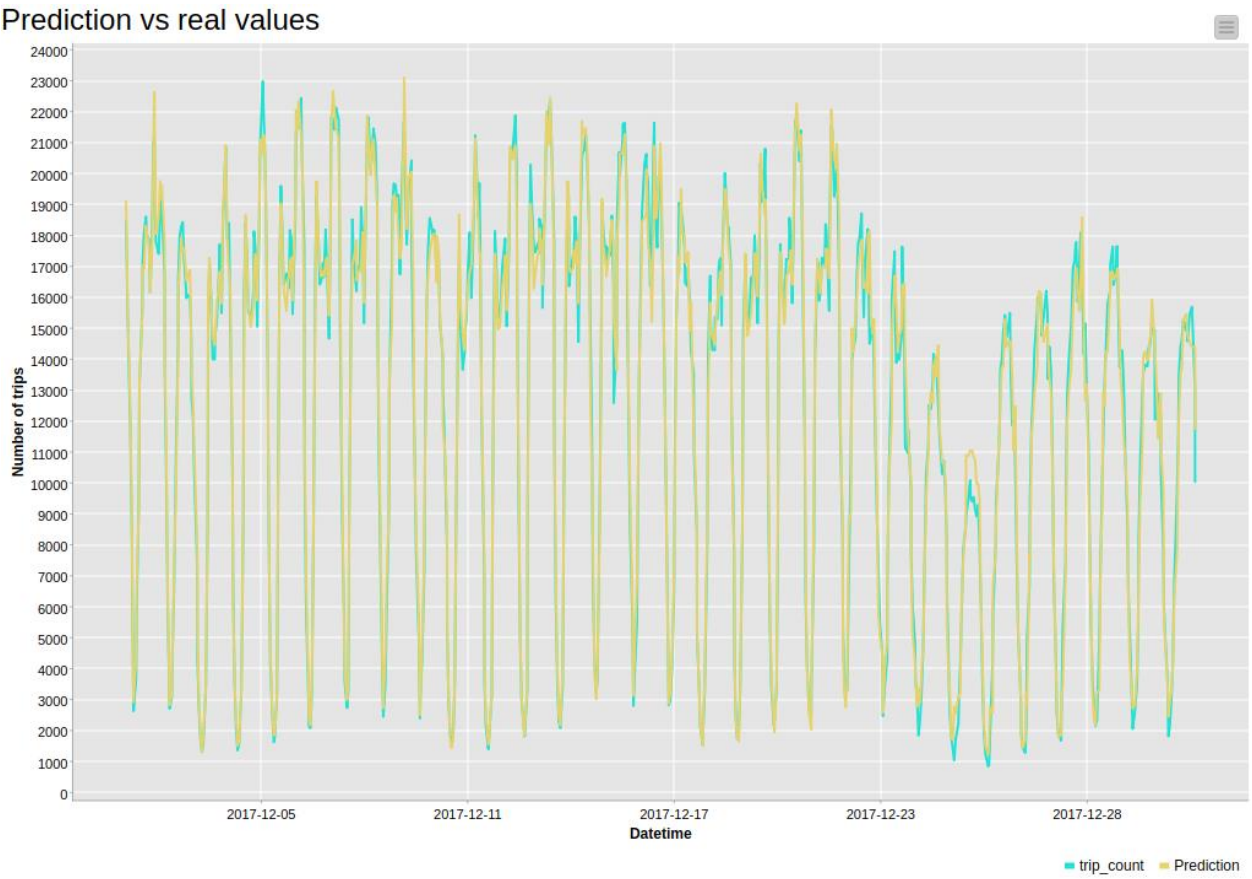


Figure 57. Line plot of the predicted (yellow) vs actual (light blue) values of the taxi demand time series

The training workflow is available on the KNIME EXAMPLES Server under
EXAMPLES/10_Big_Data/02_Spark_Executor/11_Taxi_Demand_Prediction

The Deployment Workflow

Now that we have a model that meets our expectations, we need to use it in a deployment workflow. In the deployment workflow, we use the previously trained model to predict taxi demand on new unseen data.

As deployment data (from the latest dataset the TLC has released at this point of writing (June 2018)), we will use the demand values of the last 24 hours of the month (June 30) to try to predict the demand for the first hour on July 1.

As in the training workflow, first the dataset is loaded into a Spark dataframe. The dataset has not yet preprocessed as the training dataset, so a number of additional steps is needed to prepare the data, e.g counting the number of trips per hour and extracting time components from the timestamp column. At this point, we have 24 data rows, which are the values of taxi demand for the past 24 hours.

Now we create the lagged columns in the “Spark Lag Column” metanode. After the lagged columns have been created, we only use the last vector of past values. At the same time, the Model Reader node acquires the trained model. Next, both the model and the vector of past values are fed into the Spark Predictor node, and the prediction for taxi demand for the next hour is generated.

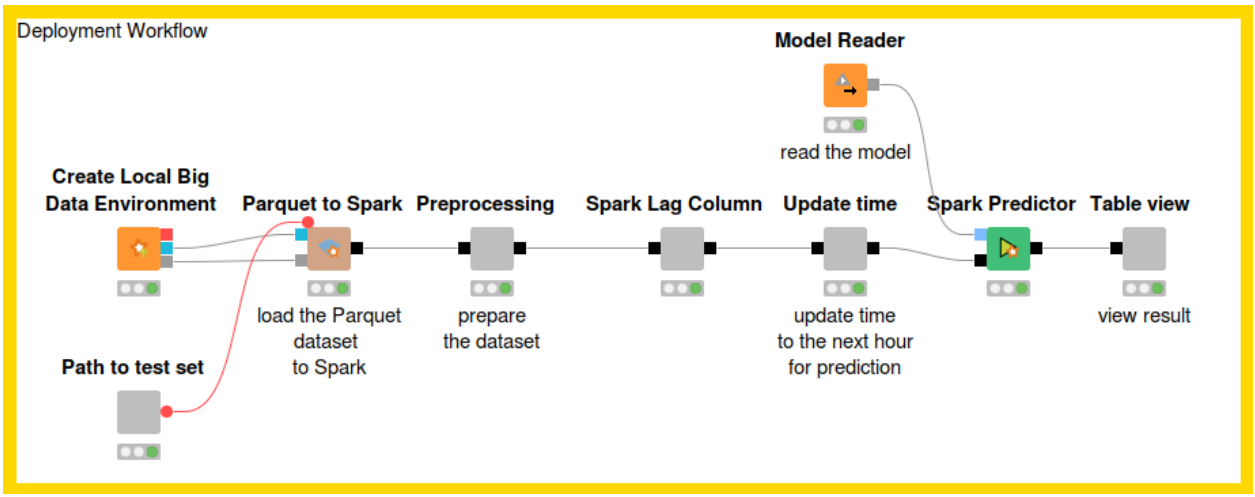


Figure 58. The deployment workflow reads the trained model, prepares the data, and feed the vector of past values and the mode into the Spark predictor node to generate the prediction of taxi demand for the next hour.

Eventually this workflow could be extended by using a recursive loop to predict more hours in the future beyond just the next hour.

The deployment workflow is available on the KNIME EXAMPLES Server under:

EXAMPLES/10_Big_Data/02_Spark_Executor/11_Taxi_Demand_Prediction

Summary

We have trained and evaluated a simple time series model using a Random Forest on the 2017 data from the NYC yellow taxi dataset in order to predict taxi demand for the next hour based on past n values. The whole model training and testing was implemented using a big data Spark framework. Looking at the test result, our simple model seemed to perform well and managed to capture the seasonality pattern, even in December, when there the time series values fluctuate considerably because of the upcoming holidays.

In this example, we decided to overlook the seasonality and train the model on the full time series. Removing seasonality and non-stationarity and training the model only on the residual time series might lead to better results. First order differencing (subtracting current value t with the value from the immediate previous time step t-1) and seasonal differencing (subtracting current value t from the seasonal previous time step, e.g t-24) can help remove non-stationarity and seasonality respectively. In a second experiment, not discussed here, we did train the model on the residual time series, but we got very similar result. So, at least for this time series using the random forest as a model, first order differences did not make a big difference to the final predictions.

To further improve the model, we could try adding more features such as temperature, taxi zones, or rain precipitation.

References

[1] TLC 2018 Factbook. https://www1.nyc.gov/assets/tlc/downloads/pdf/2018_tlc_factbook.pdf

6.3. Anomaly Detection

By Rosaria Silipo

Access workflow on hub.knime.com

Or from: EXAMPLES/50_Applications/17_AnomalyDetection

Anomaly detection covers a large number of data analytics use cases. However, here anomaly detection refers specifically to the detection of unexpected events, be it cardiac episodes, mechanic failures, or fraudulent transactions.

The "unexpected" character of the event means that no such examples are available in a dataset. All classification solutions we have seen so far require a set of examples for all involved classes. So, how do we proceed in a case where no examples are available? It requires a little change in perspective.

In this case, we can only train a machine learning model on non-failure data, i.e. on the data describing the system operating in "normal" conditions. The evaluation of whether the input data is an anomaly or just a regular operation can only be performed in deployment after the prediction has been made. The idea is that a model trained on "normal" data can only predict the next "normal" sample datum. However, if the system is not working in "normal" condition anymore, the input data will not describe a correctly working system, and the model prediction will be far from reality. The error between the reality sample and the predicted sample can then tell us something about the underlying system's condition.

In Internet of Things (IoT) data, signal time series are produced by a number of sensors strategically located on or around a mechanical component. A time series is the sequence of values of a variable over time. In this case, the variable describes a mechanical property of the object and it is measured via one or more sensors. Usually, the mechanical piece is working correctly. As a consequence, we have tons of samples for the piece working in normal conditions and close to zero examples of the piece failure.

Thus, we can only train a machine learning model on a number of time series describing a system working as expected. The model will be able to predict the next sample in the time series, when the system works properly, since this is how it was trained. We then calculate the distance between the predicted sample and the real sample and from there we draw the conclusion as to whether everything is working as expected, or if there is any reason for concern.

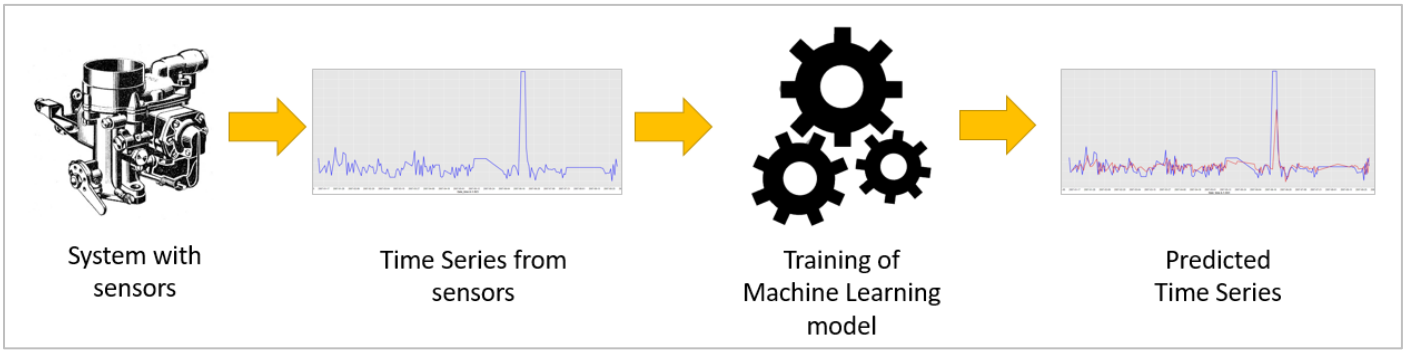


Figure 59. Anomaly detection problems do not offer a classic training set with labeled examples for both classes: a signal from a normally functioning system and a signal from a system with an anomaly. In this case, we can only train a machine learning model on a training set with normal examples and use an error measure between the original signal and the predicted signal to trigger an alarm.

What You Need

Since anomaly detection tries to discover the unknown, it only requires a dataset of data describing a normally functioning system.

Here, we used a twenty-eight-sensor matrix, focusing on eight parts (see the table below) of a mechanical rotor, on a time frame spanning from January 1, 2007 through to April 20, 2009. In total, we have 28 time series from 28 sensors attached to 8 different parts of the mechanical rotor.

The signals reach us after the application of the Fast Fourier Transform (FFT), spread across 28 files, in the form of:

```
[date, time, FFT frequency, FFT amplitude]
```

Spectral frequency and amplitude are generated by the FFT.

A1	input shaft vertical
A2	second shaft horizontal upper bearing
A3	third shaft horizontal lower bearing
A4	internal gear 275 degrees
A5	internal gear 190,5 degree
A6	input shaft bearing 150
A7	input shaft bearing 151
M1	torque KnM

Figure 60. The eight locations of the 28 sensors on the rotor.

The data have been aggregated across frequency bands, time, and sensor channel. This results in 313 time series, describing the system evolution in different locations and frequency bands.

The whole dataset shows only one breakdown episode on July 21, 2008. The breakdown is visible only from some sensors and especially in some frequency bands. After the breakdown, the rotor was replaced, and much cleaner signals were recorded afterwards.

Figure 61. Evolution over time of time series A1-SV3[0, 100] and A1-SV3[500,600]. The rotor breakdown episode on July 21, 2008 is easily visible in the higher frequency band [500, 600] Hz rather than in the lower frequency band [0, 100]. There are 313 such time series in the dataset referring to different frequency bands of the original 28 time series.

Training 313 AR Models

Machine learning models are trained on the portion of the time series running from January to August 2007 when the rotor was still functioning correctly.

One auto regressive (AR) model is trained on 10 past samples for each time series, resulting in 313 AR models, i.e. one model for each one of the time series.

To build one model for each time series, we need to loop on the time series, i.e. on the columns of the dataset. Thus, the workflow to train the AR models is centered on a loop cycle. Within the loop, on each column, for each value, we build a past of 10 previous samples, impute missing values, train a linear regression model on the past values to predict the current value, and finally calculate the error statistics (mean and standard deviation) between predicted and real value. The models and the error statistics are then saved and will be used in deployment.

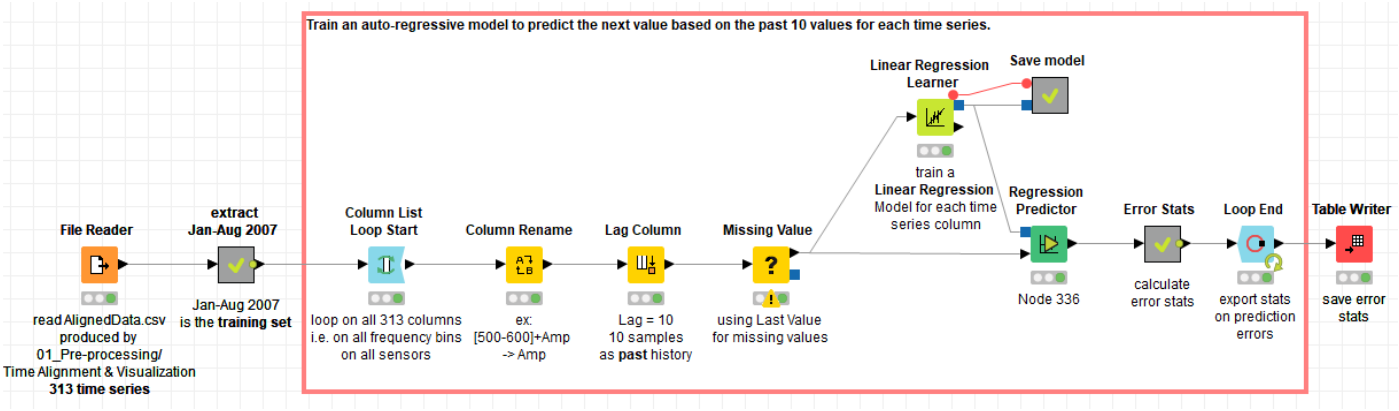


Figure 62. Training workflow 02_Time_Series_AR_Training. This workflow trains an auto regressive model on 10 past samples to predict the current sample for each of the input time series. Notice that the training set consists only of time series for a system correctly working. The model will then predict only samples of time series for correctly working system.

Deployment

The goal of the deployment workflow here is a bit more creative than a classic model application for predictions.

Here, during deployment, for each time series we predict the next value based on the past 10 values, then measure the distance between the predicted value and the current real value, and finally compare this distance with the error statistics generated during training. If the error distance is above (or below) the error mean +(-) 2* standard deviation, an alarm spike is created as large as the distance value, otherwise the alarm signal is set to 0. This alarm signal is "alarm level 1".

The whole prediction, error distance calculation, comparison with mean and standard deviation of training error, and final "alarm level 1" calculation is performed inside the column loop within the deployment workflow. 113 "alarm level 1" time series are calculated, i.e. one for each time series.

The "alarm level 1" is a series of more or less high spikes. A single spike per se does not mean much. It could be due to electricity fluctuation, a quick change in temperature, or some such temporary cause. On the other hand, a series of spikes could mean a more serious and permanent change in the underlying system. Thus, an "alarm level 2" series is created as the moving average of the previous 21 samples of the "alarm level 1" series, on all 113 columns. These are the "alarm level 2" time series and are calculated in the metanode, Alarm Level 2.

Since we need 21 daily values for this moving average, at the start of the deployment workflow we select the day to investigate and then we take at least 21 past values for all of the time series. Because of the many missing values, in order to make sure to have enough samples for the moving average, we exaggerated a bit and used a two-month window of past values.

At this point, all "alarm level 2" values are summed up across columns for the same date. If this aggregated value exceeds a given threshold (0.01) the alarm is taken seriously, and a checkup procedure is triggered in the metanode, Trigger Checkup if level 2 Alarm = 1.

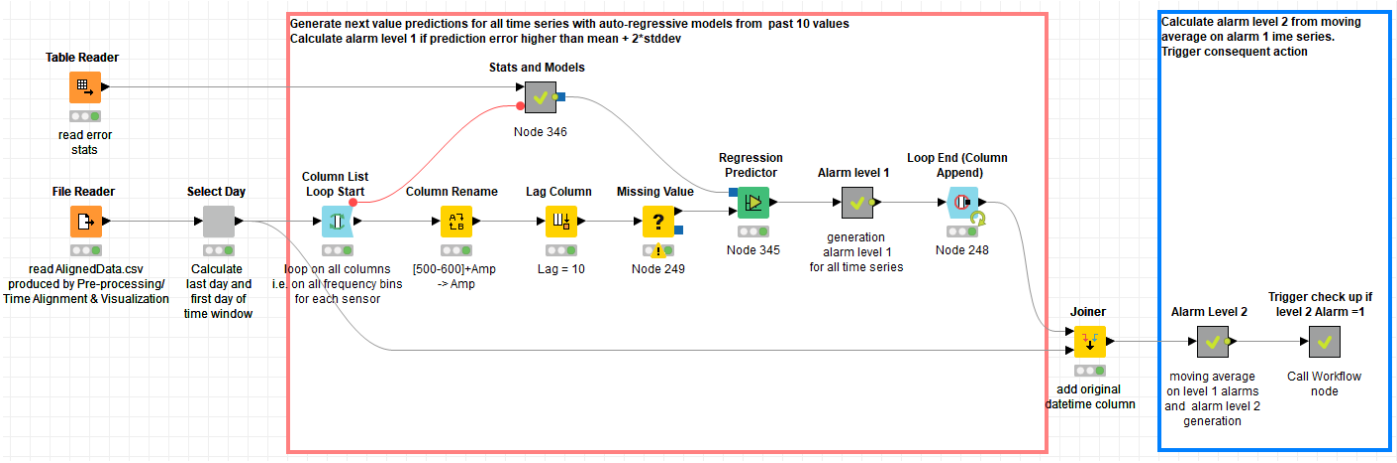


Figure 63. Deployment workflow *03a_Time_Series_AR_Deployment*. Here we read the models trained and saved in the training workflow, we apply them to the data in a new time window (at least 2 months long), we calculate the distance between predicted samples and original samples and we generate two level alarms. If alarm level 2 is active, a checkup procedure is triggered.

The trigger agent in that metanode is a CASE Switch Data (Start) node. The second port is enabled only when "alarm level 2" is active and starts an external workflow via the node "Call Workflow (Table Based)". This node in its configuration window is set to start the external KNIME workflow "Send_Email_to start_checkup".

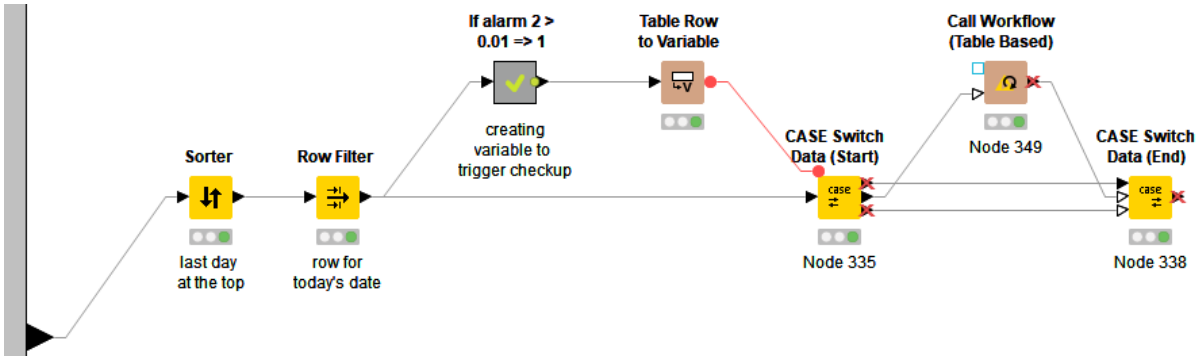


Figure 64. Content of metanode "Trigger check up if level 2 Alarm =1". The trigger node, Call Workflow (Table Based) node, calls and executes an external workflow named "Send_Email_to_start_checkup" and that is exactly what it does: it sends an email to trigger the check up procedure.

The workflow "Send_Email_to start_checkup" has just one central node: the Send Email node. The Send Email node - as the name says - sends an email using a specified account on an STMP host and its credentials.

The other node in the workflow is a Container Input (Variable) node. While this node is functionally not important, it is required to pass the data from the calling to the called workflow. Indeed, the Call Workflow (Table Based) node passes all the flow variables available at its input port to the Container Input (Variable) node, if any, of the called workflow. In summary, the Container Input (Variable) workflow is a bridge to transport the flow variables available at the caller node into the called workflow.

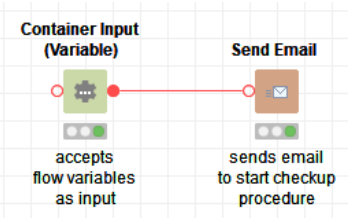


Figure 65. The workflow “Send_Email_to_start_checkup” sends an email to trigger the check up procedure. Notice the Container Input (Variable) node to pass flow variables from the caller workflow to the called workflow.

Testing Results

Just barely modifying the deployment workflow, we get the chance to test this strategy on a number of data points and therefore observe the evolution over time of the “alarm level 2” time series.

In the modified version, we read all data after the training set portion, i.e. from Sep 2007 until July 2008. The “alarm level 2” time series is visualized for each frequency band for each sensor in a stacked area chart. As you can see, “alarm level 2” values rise at the beginning of March 2008 already across all frequency bands and all sensors. However, the change in the system becomes evident at the beginning of May 2008, especially in some frequency bands of some sensors (see [200-300] A7-SA1 time series).

Considering the rotor broke off on July 22, 2008, this would have been a fairly advanced warning time!

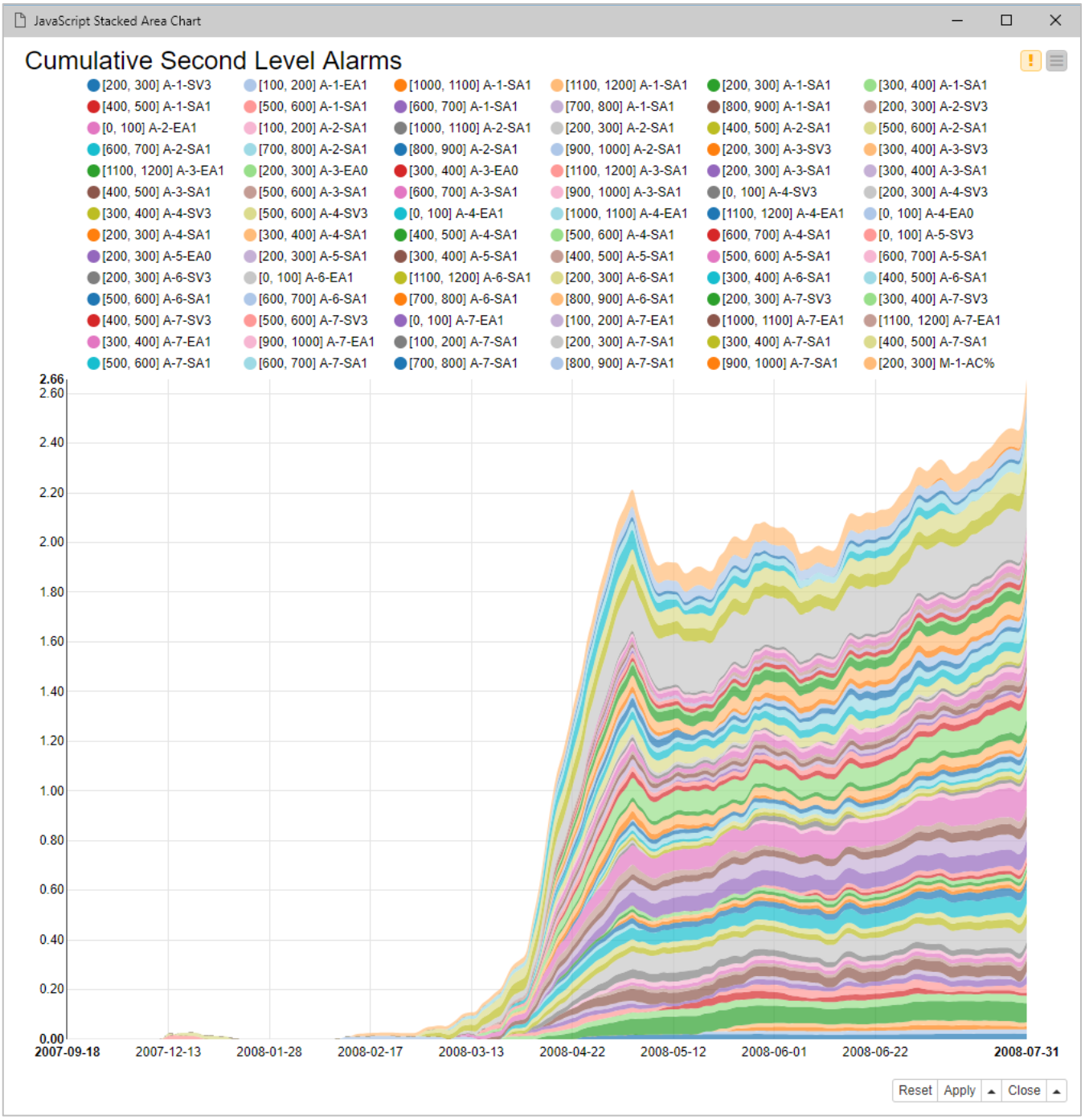


Figure 66. The deployment workflow has been slightly modified to run a test on the remaining time window until the breakup episode (Workflow *03b_Time_Series_AR_Testing*). The result is a stacked area chart piling up all "alarm level 2" signals from Jan 2007 until July 2008. You can see the alarm signal rising in March 2008 and more in May 2008.

Chapter 7. Life Sciences

7.1. DNA Sequence Similarity Search with BLAST

By Jeanette Prinz

Access workflow on hub.knime.com

Or from: EXAMPLES/50_Applications/48_BLAST_from_the_PAST/Blast_from_the_PAST

In this use case, we will have a blast by travelling ~270,000 years back in time. And we won't even need a DeLorean, just our trusty KNIME Analytics Platform. We will return to the Ice Age and examine genetic material retrieved from a cave. We will solve the mystery as to which species the DNA came from.

To investigate this ancient DNA, we will utilize one of the most widely used bioinformatics applications: BLAST (Basic Local Alignment Search Tool). We will do that by running the BLAST RESTful interfaces to submit BLAST searches inside KNIME Analytics Platform.

This post speaks directly to people interested in bioinformatics. At the same time, because it covers the technical aspect of handling asynchronous REST operations, it will also be useful for those in other fields. Thus, if you have an interesting web application you want to use within KNIME - especially one where the resource method execution takes longer - keep reading!

What You Need

Bioinformatics without BLAST is like a spaceship without rocket fuel. BLAST is a registered trademark of the National Library of Medicine. The original paper[1] has been cited over 70,000 times, not to mention many highly cited follow-up papers[2,3], BLAST is a fast and robust algorithm that finds regions of similarity between biological sequences. The sequences can be DNA or protein sequences from diverse organisms. Biological sequences can be viewed as very large strings like this: AGTCGCAGAGT... The human genome, for example, consists of a sequence comprised of over 3 billion letters. Those sequences change over time, and this drives evolutionary processes (e.g. when one letter is exchanged for another or is removed completely). Hence, if we want to compare many sequences, it is important to use fast and clever algorithms that allow us to find similar sequences even if changes have taken place. BLAST enables a user to compare a query sequence with a huge database of sequences to find the most similar ones.

So what does any of this have to do with our time-travel journey? If we find e.g. a random bone on that journey, we can extract the DNA, and then use BLAST to compare it with a database of DNA sequences and find out which organism the bone belongs to. That's exactly the sort of problem we will solve now.

Analyzing 270,000 Year Old DNA

In our use case, we want to analyze DNA extracted from an archaic femur (thigh bone) from Hohlenstein–Stadel cave in southwestern Germany, which is at least 270,000 years old. We use part (~50%) of the DNA sequence published last year by a research team led by scientists from the Max Planck Institute for the Science of Human History and the University of Tübingen4. More specifically, we want to investigate mitochondrial DNA (mtDNA). Sequences of mtDNA sequences are relatively short and, since they evolve faster than nuclear genetic markers, they accumulate differences comparatively quickly. This makes the analysis of mtDNA one of the most important tools for phylogenetics and evolutionary biology.

The Workflow

We want to compare the mtDNA sequence with a huge set of DNA sequences from diverse organisms. For that, we create a workflow that receives a sequence as input, queries BLAST and shows the BLAST results in an interactive table (see Fig. 1).

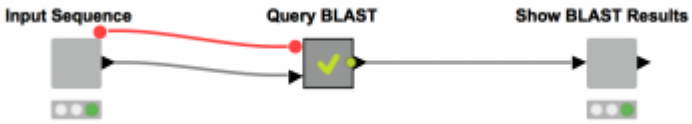


Figure 67. BLAST workflow. The workflow receives an input sequence, queries BLAST, and subsequently shows the BLAST results in an interactive table. The views are accessible via KNIME WebPortal or via the node view.

We compare our sequence to the extensive Nucleotide database, a collection of sequences from several sources, including GenBank, RefSeq, and PDB. In July 2018 RefSeq alone covered genetic data from more than 81,000 organisms. Comparing the input sequence with the vast number of sequences in the database can take a while, hence the BLAST RESTful interface handles queries asynchronously.

Handling Asynchronous REST Operations

Calling a web application synchronously implies that you have to wait for the response to return before you can continue. In cases where the response takes longer this blocks your workflow. In contrast, asynchronous operations are non-blocking, i.e. client execution continues immediately after the request, and the response is then processed later, by a different execution context or thread[5].

Handling asynchronous REST operations introduces some challenges, as we receive a return value even when the execution is not finished. What that means for us here is that after starting our BLAST job our workflow will repeatedly call the web service to check whether or not it has finished. This workflow is contained in the metanode “Query BLAST”. We first create a new resource using the PUT method. We then extract the “Request Identifier” (RID) and the “Request Time of Execution” (RTOE). Afterwards, we build a loop where we keep waiting RTOE seconds until the status of the request equals READY. Now that we know that the BLAST results are ready for retrieval, we can fetch them via a final GET request.

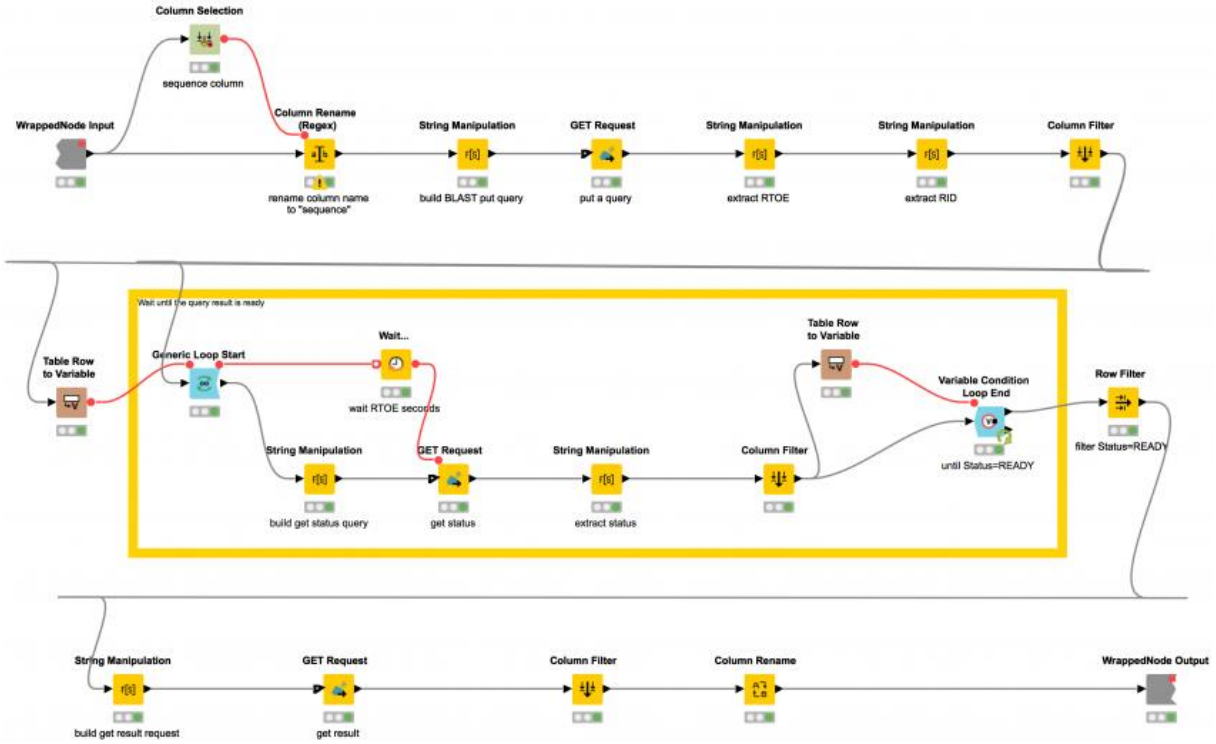
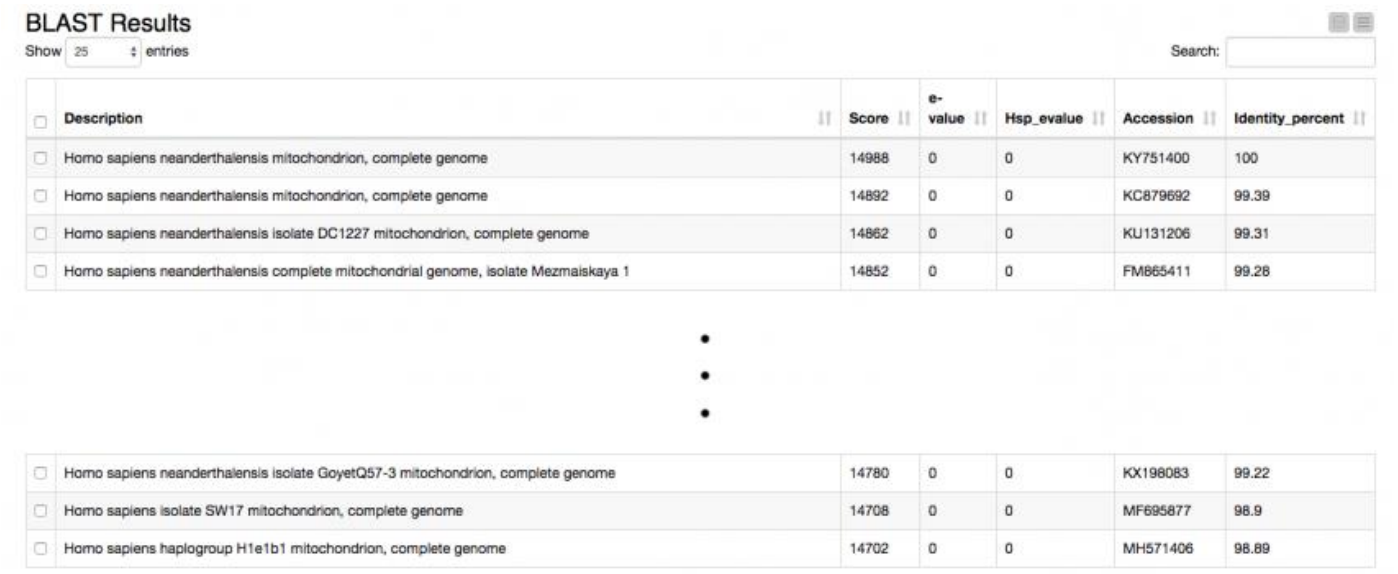


Figure 68. Retrieval of results from an asynchronous GET Request: First, we extract the 'Request Identifier' (RID) and the “Request Time of Execution” (RTOE). We then build a loop where we keep waiting RTOE seconds until the status of the request equals READY and subsequently retrieve the BLAST results. This workflow is contained in the metanode “Query BLAST”.

BLAST Result

The BLAST query gives us an XML document, which we parse using the XPath nodes in KNIME. We extract the description of the hit, the accession ID, the length of the sequence, the score, and the e-value. The score adds up the positions where the two sequences are the same and penalizes those where they differ. The e-value describes the expected number of hits with an equal or higher score one would expect to see by chance when searching a database of a particular size. The lower the e-value (the closer it is to zero), the more “significant” the match is. In addition, we collect the field identity, which signifies the extent to which two sequences have the same residues (letters) at the same positions. By dividing by the length of the sequence and then multiplying by one hundred we receive the percentage of sequence identity that is shared between the input and the hit sequence. For details of the calculation and interpretation of the extracted results, see the BLAST Glossary.

Now: What will we find? Who did the mysterious bone belong to?



BLAST Results

Show 25 entries

Search:

Description	Score	e-value	Hsp_evalue	Accession	Identity_percent
Homo sapiens neanderthalensis mitochondrion, complete genome	14988	0	0	KY751400	100
Homo sapiens neanderthalensis mitochondrion, complete genome	14892	0	0	KC879692	99.39
Homo sapiens neanderthalensis isolate DC1227 mitochondrion, complete genome	14862	0	0	KU131206	99.31
Homo sapiens neanderthalensis complete mitochondrial genome, isolate Mezmaiskaya 1	14852	0	0	FM865411	99.28
Homo sapiens neanderthalensis isolate GoyetQ57-3 mitochondrion, complete genome	14780	0	0	KX198083	99.22
Homo sapiens isolate SW17 mitochondrion, complete genome	14708	0	0	MF695877	98.9
Homo sapiens haplogroup H1e1b1 mitochondrion, complete genome	14702	0	0	MH571406	98.89

Figure 69. JavaScript Table View of the BLAST Result. The Description, score, e-value, accession, and Identity_percent are displayed. The e-value describes the expected number of hits with an equal or higher score and Identity_percent signifies the percentage of sequence identity that is shared between the input and the hit sequence. This is the output of the metanode "Show BLAST results".

Our top hit, with 100% identity and a significant e-value, is "homo sapiens neanderthalensis mitochondrion". This is the DNA that was submitted by the scientist from the Max Planck Institute. The following hits also manifest Neanderthal mtDNA from different archaeological sites. Our bone, conclusively, was a Neanderthal bone. Different isolates of homo sapiens mtDNA are also among the significant hits with an identity of ~98.9% showing once more that we are not as different from the Neanderthals as we might think.

Deployment

We used an interactive KNIME workflow that is able to handle asynchronous REST operations to investigate ancient DNA. This workflow makes use of the RESTful interfaces offered by the NCBI to programmatically submit BLAST searches.

The workflow can be easily deployed to the KNIME WebPortal, as we used wrapped metanodes containing quickforms and JavaScript views for all user interactions. This means that the end user does not have to know about the analytics going on under the hood. Moreover, you can also reuse these metanodes in other workflows with different inputs and process the results further. Last but not least, this can be seen as a template to handle (asynchronous) REST operations for diverse webservices that have nothing to do with sequence similarities.

7.2 Automatic Tagging of Disease Names in Biomedical Literature

By Jeanette Prinz
Access workflow on hub.knime.com
Or from:
EXAMPLES/08_Other_Analytics_Types/02_Chemistry_and_Life_Sciences/03_Fun_with_Tags

Introduction

The rapid growth in the amount of biomedical literature becoming available makes it impossible for humans alone to extract and exhaust all of the useful information it contains. There is simply too much there. Despite our best efforts, many things would fall through the cracks, including valuable disease-related information. Hence, automated access to disease information is an important goal of text-mining efforts[1]. This enables, for example, the integration with other data types and the generation of new hypotheses by combining facts that have been extracted from several sources[2].

What We Need

In this use case we use KNIME Analytics Platform to create a model that learns disease names in a set of documents from the biomedical literature. The model requires two inputs: an initial list of disease names and the documents. Our goal is to create a model that can tag disease names that are part of our input as well as novel disease names. Hence, one important aspect of this project is that our model should be able to autonomously detect disease names that were not part of the training.

To do this, we will automatically extract abstracts from PubMed and use these documents (the corpus) to train our model starting with an initial list of disease names (the dictionary). We then evaluate the resulting model using documents that were not part of the training. Additionally, we test whether the model can extract new information by comparing the detected disease names to our initial dictionary.

Subsequently, we interactively inspect the diseases that co-occur in the same documents and explore genetic information associated with these diseases.

The Workflow

Our workflow has three main parts (see Fig 1.), which will be described in detail in the following sections:

1. Dictionary and Corpus Creation
2. Model Training and Evaluation
3. Co-occurrence of Tagged Disease Names

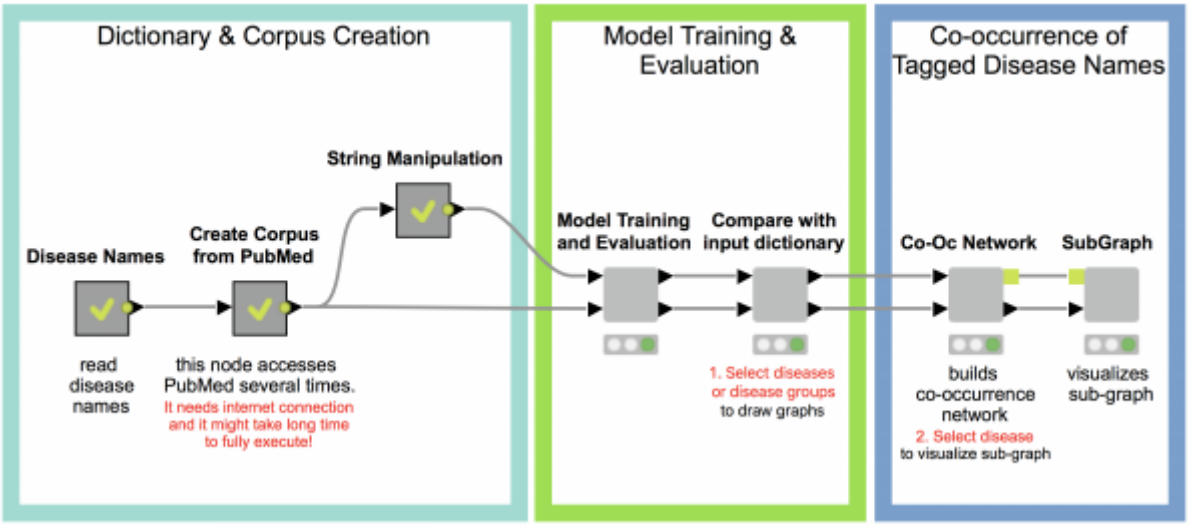


Figure 70. Overview of the workflow to automatically extract disease related information from biomedical literature. First, the literature corpus as well as the dictionary of known disease names are gathered. Next, the model is trained and evaluated. Last, the results are investigated in a network graph.

1. Dictionary and Corpus Creation

Dictionary creation (Disease Names)

For the initial input, we create a dictionary that contains disease names from Ensembl Biomart. For that, we downloaded phenotypes (diseases and traits) that are associated to genes or variants. These diseases and traits are assembled from different sources such as OMIM, Orphanet, and DDG2P. To create a dictionary that contains commonly used disease names, we filter the documents for disease names that are contained in at least three sources. The model we eventually train is case sensitive, thus, we add variations such as capitalizing the disease names, lowercase, and uppercase. The resulting disease names comprise our initial input dictionary. The dictionary creation is contained in the metanode named “Disease Names” in the main workflow.

Corpus creation

One of the most important steps for creating a NLP (natural language processing) model is to gather a corpus of documents on which to train and test the model. For our purpose of automatically accessing disease information, we use abstracts from the database PubMed. The KNIME node “Document Grabber” enables us to automatically search the PubMed database according to specific queries. This query takes a disease name from our dictionary and searches for it in the PubMed data. We only keep the results from diseases with at least 20 hits in PubMed, and we collect a maximum of 100 documents per disease. This Corpus is created in the metanode named “Create Corpus from PubMed” metanode.

2. Model Training and Evaluation

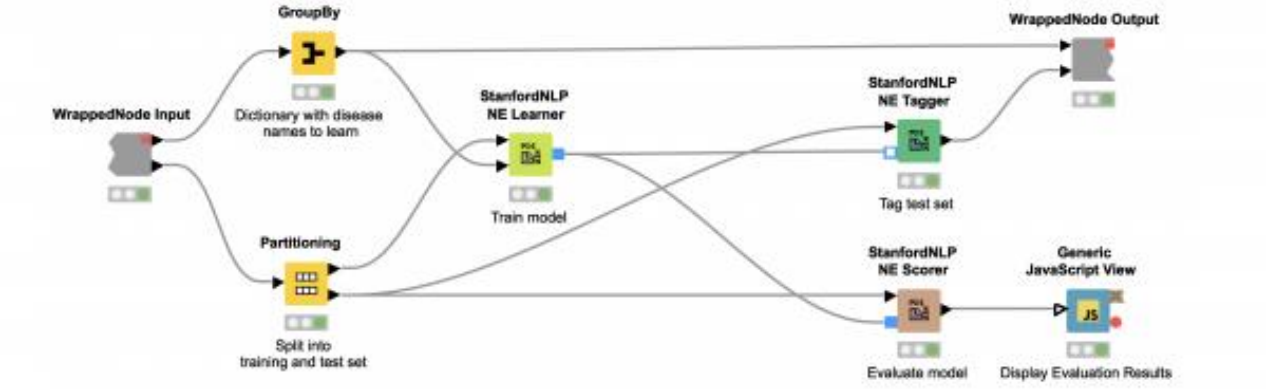


Figure 71. Workflow contained in the wrapped metanode "Model Training and Evaluation". We use the StandordNLP NE Learner and Tagger nodes to tag disease names in our corpus. The evaluation is done using the StanfordNLP NE Scorer and displayed using a Generic JavaScript View.

We can now use the dictionary and our corpus as input for the StanfordNLP NE Learner. The StanfordNLP NE Learner creates a Conditional Random Field (CRF) model based on documents and entities in the dictionary that occur in the documents. CRFs, a type of sequence model, which takes context into account, are often applied in text mining. If you are interested in the StanfordNLP toolkit, please visit http://nlp.stanford.edu/software/.

Figure 71 depicts the workflow contained in the wrapped metanode "Model Training and Evaluation". As the figure shows, we first split our collected documents into a training (10%) and a test set (90%) and train the model using the training data. We use the default parameters, with the exception that we increase maxLeft (the maximum context of class features used) to two and Max NGram Length (maximum length for n-grams to be used) to ten. Additionally, we select the Word Shape function dan2bio.

Next, we tag the documents in our test data with our trained model. Subsequently, we use the same test data to score our model. This is done with the node "StanfordNLP NE Scorer", which calculates quality measures like precision, recall, and F1-measures and counts the amount of true positives, false negatives, and false positives. Note that it does not make sense to calculate true negatives, as this would be every word that is correctly *not* tagged as a disease. Internally, the "StanfordNLP NE Scorer" node tags the incoming test document set with a dictionary tagger using our initial disease dictionary. After that, the documents are tagged again via the input model, and then the node calculates the differences between the tags created by the dictionary tagger and the tags created by the input model.

The "Generic JavaScript View" node helps us to generate a view summarizing the results. As can be seen in Figure 72, we achieve a Precision of 0.966, Recall of 0.917, F1 of 0.941.

Confusion Matrix

Actual \ Predicted	positive	negative
positive	8212	743
negative	287	xx

Performance:

Precision: 0.966
Recall: 0.917
F1: 0.941

Figure 72. Confusion matrix containing the true positives, false positives, and false negatives. Precision, Recall and F1 are also shown.

Comparison with input dictionary

Now comes the interesting part. We are not only interested in how well our model recaptures disease names that we already know, but also if we are able to find new disease names. Therefore, we divide the diseases we have found in the test set depending on whether or not they were in our initial dictionary. We flag these either as “disease name contained in the input dictionary” or, alternatively, as “disease name NOT contained in the input dictionary.”

We then create an interactive view that allows the user to investigate and filter the results. To select all data that were (or were not) contained in the input we use a “GroupBy” node to group according to the attribute we just created (i.e., if the data were part of the input dictionary or not). Here we use a small trick: it is important to “Enable highlighting” in the GroupBy node. If we show that in a composite view using a Table View (JavaScript) alongside another JavaScript view, it is now possible to make selections in one view which affect the other as well. If the user does not select anything, we will use all diseases by default.

This part is included in the metanode named “Compare with input dictionary”

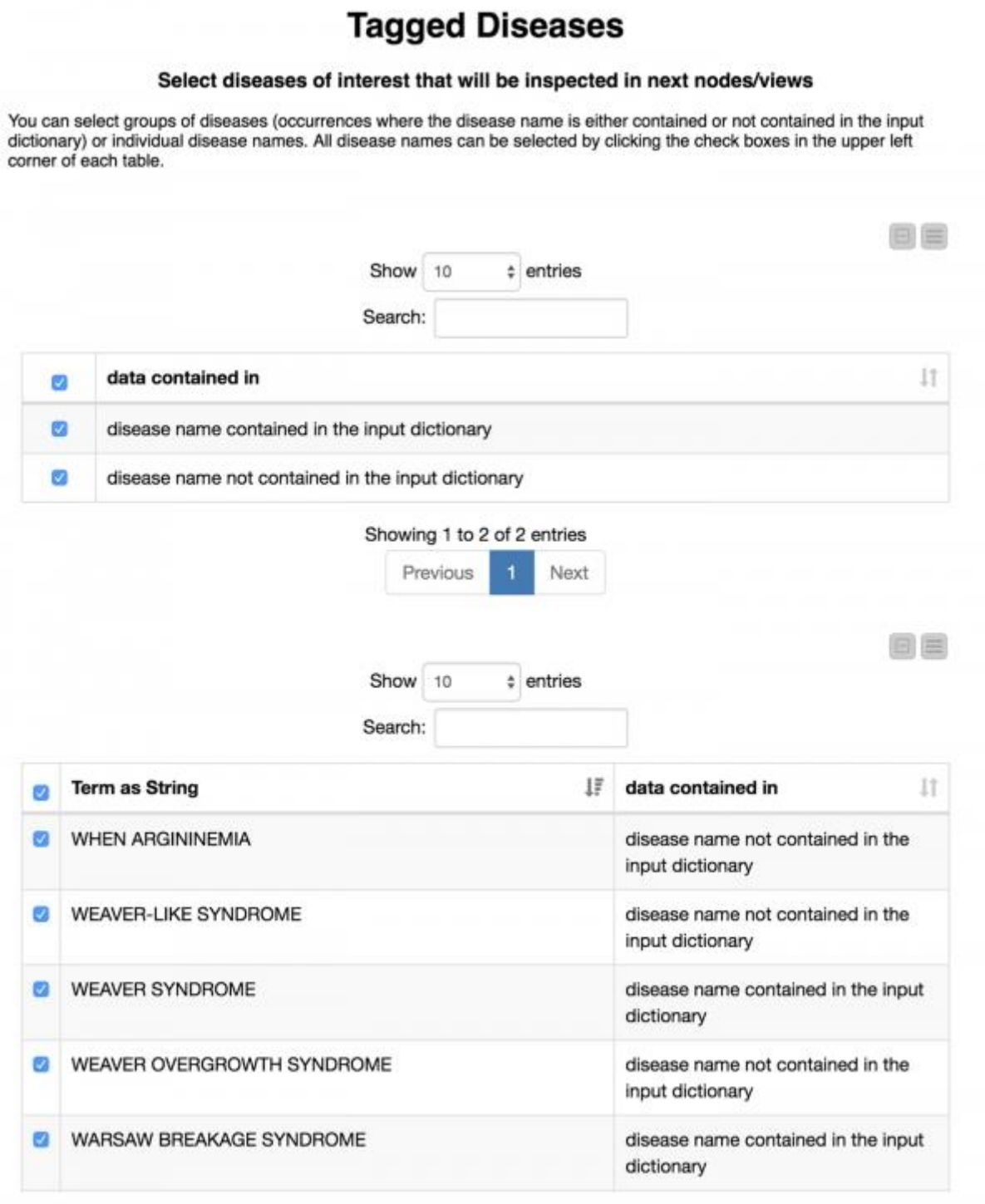


Figure 73. Interactive view of the results. The user can select one or more diseases or even all diseases that were (not) part of the input in the lower table view. This affects the first table and shows the corresponding diseases appearing in the test set. This is the output of the metanode "Compare with input dictionary".

The disease names detected in the test set that are not contained in the input dictionary can be very similar to the ones that we used for the training. For example, PYCNODYSOSTOSIS is contained in our input dictionary and we detect the new name PYCNODYSOSTOSIS SYNDROME as well as the misspelling PYCNODYSOTOSIS, which we flag as not part of the input dictionary. The similarity of the tags shows us that the tagged disease names that are not part of the input dictionary actually do make sense. These alternate tags can be valuable, for example, in normalization efforts where we need to determine synonyms and/or spelling variants of disease names.

To learn more about the newly detected disease names where the relationship to our input is less clear, we investigate whether or not tagged disease names co-occur in the same documents. This enables us to, for example, infer information from known diseases to the ones that were not in the input dictionary.

3. Co-occurence of Tagged Disease Names

In this final part, we utilize the node “Term Co-Occurrence Counter” to count the number of co-occurrences for the list of tagged diseases within the documents. We add the information to the resulting disease pairs pertaining to whether or not each term was part of the input dictionary.

Co-occurrence network

To facilitate the investigation of the results, we created a network graph with diseases as nodes, which were connected if they co-occurred in the same document. We colored the nodes according to their flag specifying whether or not they were contained in the input dictionary. We then created a view containing the network as well as a table with the disease names and the annotation stating whether it was part of the input dictionary. The creation of the network graph, node assignment, coloring, and edge definition is all computed in the metanode named “Co-oc Network”. The view is shown in Figure 74.

The user can now select nodes or rows of interest for further inspection either in the network or in the table. The subgraph surrounding the selected nodes/rows will be extracted and displayed in the next metanode.

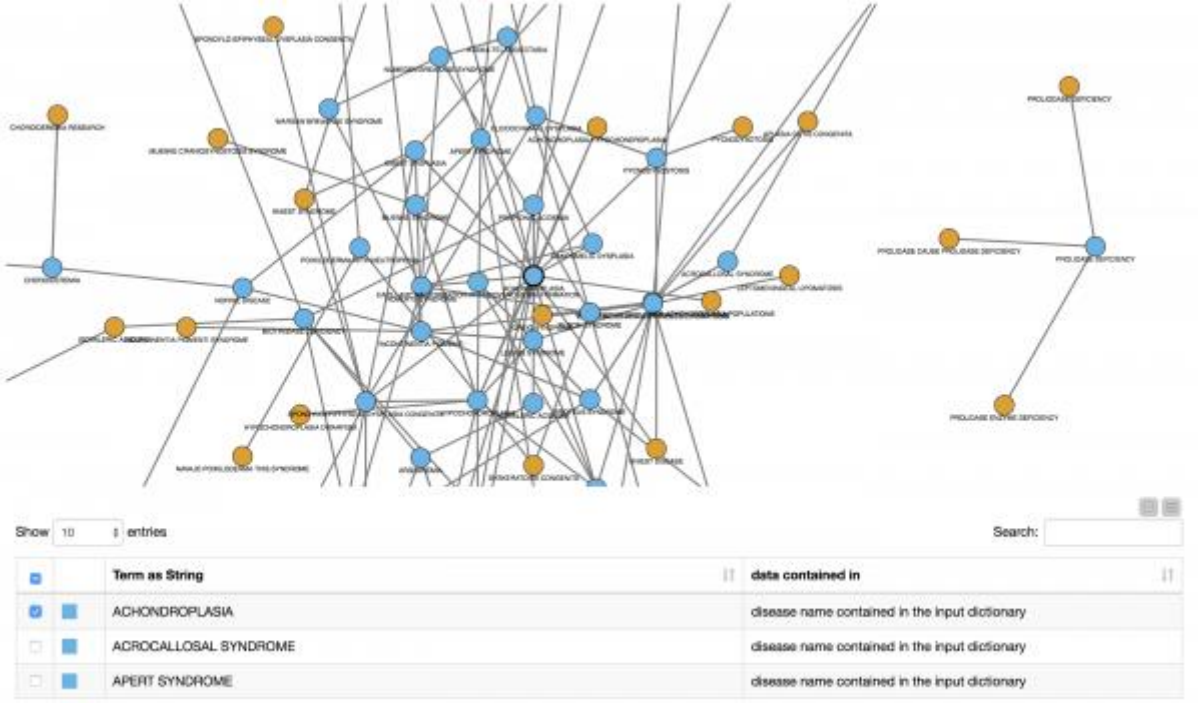


Figure 74. Network view of co-occurring disease names. Each node is a disease names. Nodes are connected if the disease name co-occur at least once in a document. The node color refers to the presence (blue) or absence (yellow) of the disease name in the original dictionary.

Subgraph

The KNIME node “SubGraph Extractor” enables us to focus on a specific subset of diseases and their neighbors in the co-occurrence network. For that, the user needs to select a disease of interest. If nothing is selected we display a message stating that one did not select a disease to inspect in subgraph. We utilize the “Try” and “Catch Error” nodes to check if the selection is empty.

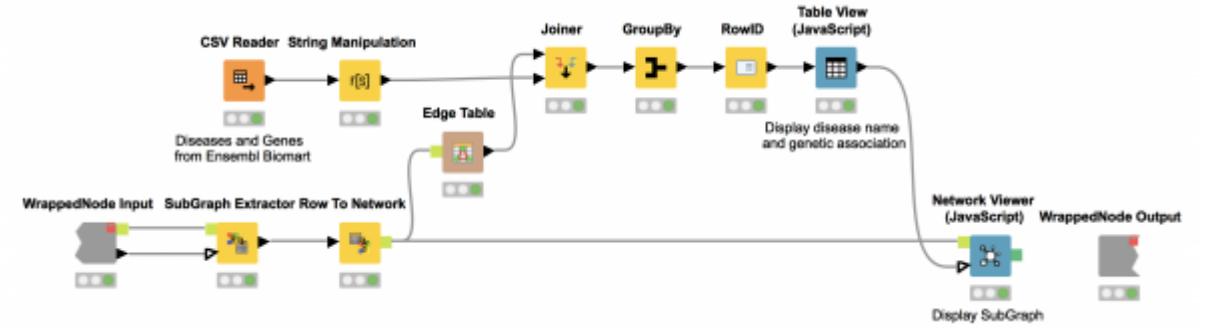


Figure 75. Workflow to extract subgraphs of interest in the network of co-occurring disease names. This workflow is contained in the last metanode, "SubGraph".

Again, we show the network (Network Viewer node) in an interactive view along with a table (Table View node) containing the disease names. Furthermore, we display additional information about genetic associations of the disease in the same table. We do this by joining the edge table of our network with genetic information about the diseases that we collected from Ensembl Biomart (see Figure 69). This allows us to derive hypotheses about the genetic basis of the diseases that were not in our input dictionary.

For example, we select OHDO SYNDROME, which was not part of our input dictionary. In the resulting view (Fig. 7), we see that this disease co-occurs with GENITOPATELLAR SYNDROME. Using the Ensembl information, we know that GENITOPATELLAR SYNDROME is associated with the gene, KAT6B. This could lead to the working hypothesis that OHDO SYNDROME is also associated with KAT6B. Indeed, mutations in the KAT6B gene have been associated with the Say-Barber-Biesecker Variant of Ohdo Syndrome[3].

Figure 76. Subgraph connecting GENITOPATELLAR SYNDROME and OHDO SYNDROME. Blue nodes indicate that the disease was part of our input dictionary, whereas yellow nodes indicate that the disease was not included.

Summary

In our workflow, we successfully trained a model to tag disease names in biomedical abstracts from PubMed. We started with a set of well known disease names in a dictionary and went on to interactively investigate these known diseases as well as diseases that were not in our original dictionary, checking their co-occurrence in the collected documents. From these co-occurrences, we created a co-occurrence network we could use to zoom easily into connected subgraphs and their underlying genetic associations.

Deployment

All views in this workflow are JavaScript based, which makes it an excellent candidate for deployment on KNIME Server and use through the KNIME WebPortal. You can also utilize the KNIME Model Writer node to write the created model to a file and then reuse it in different workflows using the Model Reader node.

7.3 Creating and Deploying a Self-testing Prediction Service

By Greg Landrum

Access workflow on hub.knime.com

Or from:

EXAMPLES/50_Applications/51_Model_Deployment_and_Validation/01_Deploying_and_Validating_models_as_WebServices

In this article, we'd like to demonstrate how to create and deploy a prediction service that tests itself. We will build and deploy a web service in KNIME that can be validated both in KNIME Analytics Platform itself and on the Server once it has been deployed. When a workflow enters "production" and becomes an essential part of your work, it's important to always be sure that it's still doing what it's supposed to do. This is particularly true when something in your system environment changes, e.g. you install a new version of KNIME, move to a new computer, or update some of your nodes.

This example is based upon some previous work, involving building and deploying a set of machine learning models for safety-related endpoints using data from ChEMBL. In this particular case, the plan was to deploy the models as RESTful web services using the KNIME Server. Since the web services were intended for production use (and since there's no escaping Murphy's Law) it's important to be able to validate that the deployed models are working properly. *Note*: although the example discussed here deals with machine learning models for bioactivity prediction, the techniques shown here can be used for any type of predictive model.

What You Need

Ok, let's get started! For this exercise we decided to build the models using the XGBoost integration that was added to KNIME in v3.7. There are, of course, many other things that could have been used, but gradient boosting normally works well for building models for bioactivity. Once the work to build and validate the models was finished - the workflow for this is linked at the end of this post - we filtered out the models that performed poorly in validation and saved the remaining eight models into a KNIME table. KNIME's "Model to Cell" node is very useful for things like this!

The Prediction Workflow

Having filtered out the eight models, we were ready to build the prediction workflow. This is done by reading in some new molecules, generating fingerprints and then looping over the models to generate predictions. Figure 77 shows the basic prediction workflow.

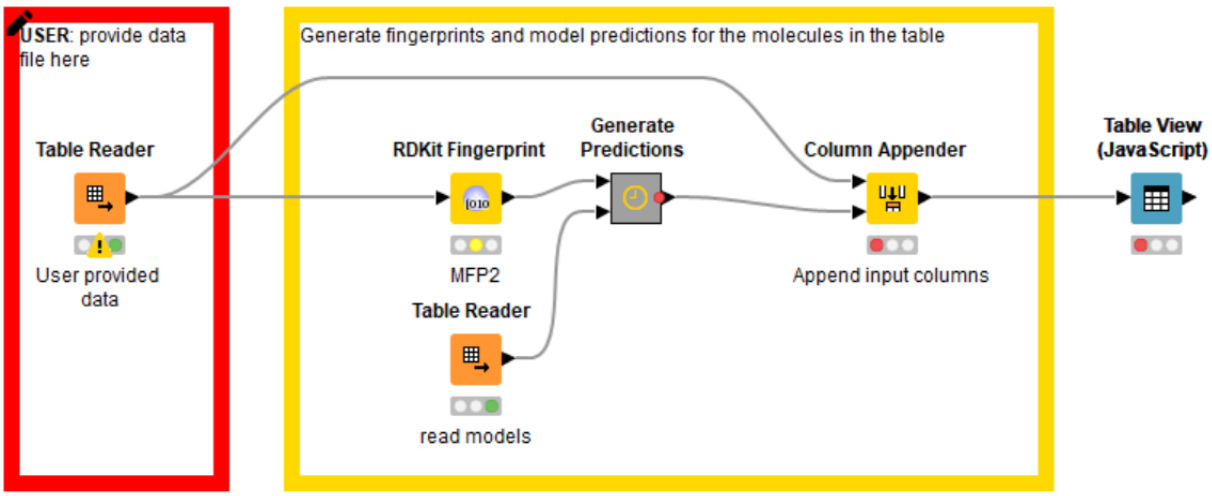


Figure 77. The basic prediction workflow

The only small bit of complexity in this is the loop over models, where the prediction columns need to be renamed so that the user can see what the predictions are for. This is all collapsed into the “Generate Predictions” metanode.

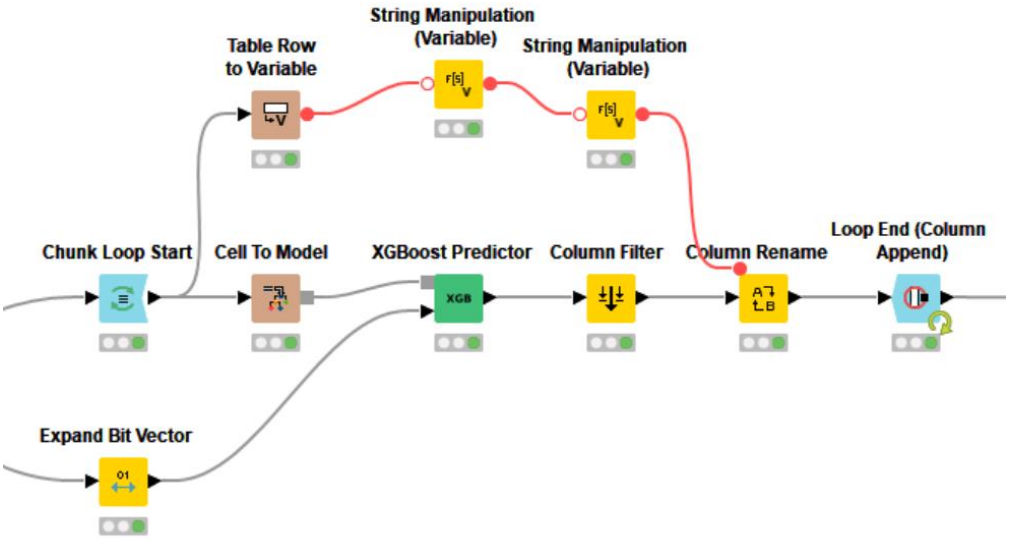


Figure 78. The “Generate Predictions” Metanode. The metanode contains a loop over all models and renaming of the prediction column.

At this stage, it might be possible to declare the work done and move on to the next item on your ToDo list. This time, however, a little more work was needed: the goal was to be able to generate predictions as a web service, so the task now was to adapt the workflow to work as a RESTful web service when deployed to the KNIME Server. If your prediction service is going to be in use for a while, it really is important to be able to make sure that everything is still working correctly with the workflow when something changes with the infrastructure (for example, deploying on a different operating system, updating the version of the RDKit, etc.). Fortunately, KNIME simplifies the testing to make sure that workflows are functioning correctly.

Preparing the Workflow for Testing

The prediction workflow for automated testing is prepared exactly the same way as was demonstrated in the earlier article - by adding the two wrapped metanodes in the blue boxes in Figure 79. The first of these loads a set of reference input data and the second compares the results generated for that input against a table of reference results. If the user provides input data when running the workflow (instead of the empty table that is the default), these wrapped metanodes just pass the data along. This workflow can now operate in two modes: if the user provides data, then it runs normally and generates predictions, but if the input table is empty it runs the validation tests and verifies that everything is still working as expected.

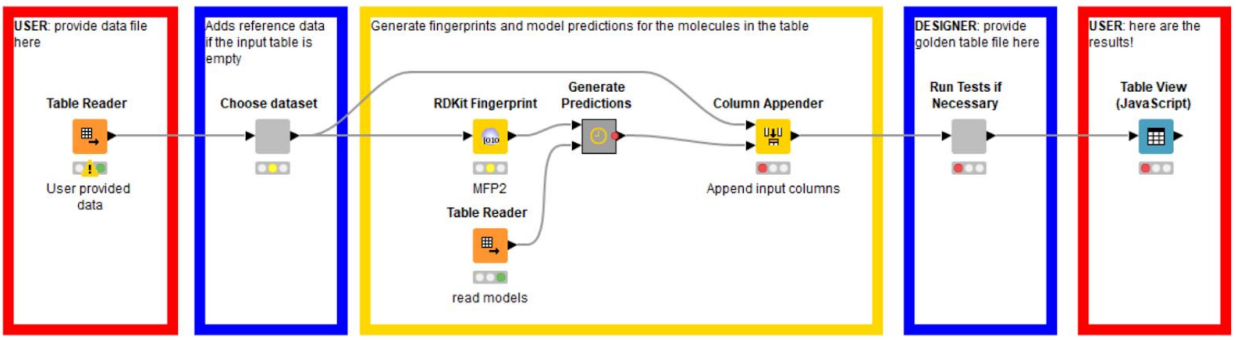


Figure 79. The prediction workflow modified to allow automated testing

Great! Now we have a testable workflow and now need to set it up as a web service.

Deployment - Making it a Web Service

In an article, “Creating Restful Services with KNIME” on the KNIME Blog, we explained how to get a standard KNIME workflow ready to be a web service. Since that post appeared we’ve made a few changes to make this process even easier. In this case all that was needed was to add the “Container Input (Table)” and “Container Output (Table)” nodes to the workflow, Figure 79. These are general purpose nodes indicating that a workflow will either be receiving data from an outside source or providing data to an outside consumer[1].

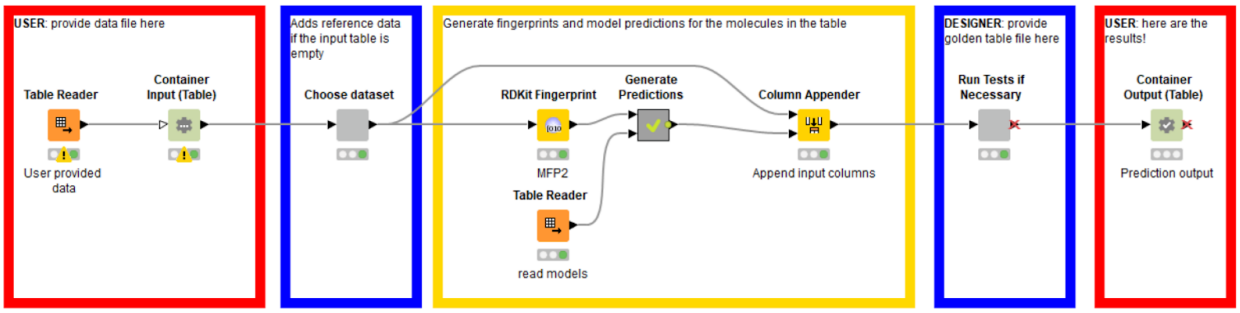


Figure 80. The prediction workflow modified to allow use as a web service

In this case, adding the Container nodes to the workflow does not have any effect at all when the workflow is run in KNIME Analytics Platform. Once deployed to the KNIME Server as a web service these nodes become the places where the data provided by the caller enters the workflow and where data from the workflow is passed back to the caller. The nodes also provide nice examples of the data expected/returned. We can see these by looking at the OpenAPI definition of the prediction service, Figure 81 and Figure 82.

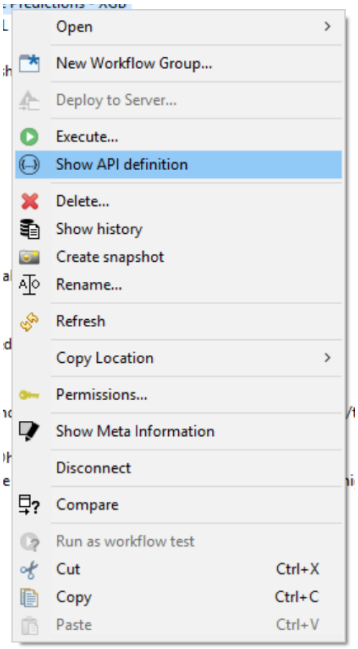


Figure 81. Opening the OpenAPI definition for a web service

[1] Though the “Container Input (Table)” and “Container Output (Table)” nodes were used here to make the workflow “webservices-ready”, the same nodes were used to get a workflow ready to be called from Python in another article, KNIME and Jupyter https://www.knime.com/blog/knime-and-jupyter.

Inline input parameters for the job

Example Value | Model


```
{
  "molecule_input-250": {
    "table-spec": [
      {
        "compound_id": "string"
      },
      {
        "smiles": "Smiles"
      }
    ],
    "table-data": [
      [
        "ChEMBL_36_A_1",
        "CN(C)c1ccc(C2CC3(C)C(CCC3(O)C#Cc3ccc(C(C)(C)C)cc3)C3OCC4=CC(=O)CCC4=C23)cc1"
      ],
      [
        "ChEMBL_36_A_2",
        "CCc1cn(CCN)c(CC)c1Oc1ccc(C#N)cc1"
      ]
    ]
  }
}
```


Figure 82. The sample input provided by the OpenAPI definition for our prediction web service. Compund_id and smiles are provided for each molecule.

Once the prediction workflow was deployed to the server, it could be invoked by POSTing JSON similar to what is shown in Figure 82 to the appropriate URL (this URL is from the OpenAPI definition page that Figure 82 is clipped from). The figure below shows a piece of the output table as provided in the JSON that was returned from calling the service. We can see that, in addition to the compound_id and smiles I provided, there are also 8 predictions and 8 predicted probabilities of being active. The order of these data is provided a bit higher up in the JSON output (not shown here).


```
"table-data": [
    [
        "ChEMBL_36_A_10",
        "N#Cc1ccc(-c2ccccc2O)cc1C(F)(F)F",
        0.9999991655349731,
        "active",
        0.00002162553028028924,
        "inactive",
        0.0011914604110643268,
        "inactive",
        3.304523044089791e-10,
        "inactive",
        3.868060318118309e-14,
        "inactive",
        1.5059975488895816e-9,
        "inactive",
        5.6007577048378l1e-12,
        "inactive",
        0.000024705152554815 87,
        "inactive"
    ],
```


Figure 83. First row of the output table when the prediction web service is called with input data

At this point the workflow was shaping up nicely and able to be used in KNIME Analytics Platform to apply the eight models to generate predictions for new data, we also had automated tests running in KNIME Analytics Platform to verify that the workflow was generating correct predictions for a set of reference input. A web service was also deployed via KNIME Server that could be used from applications to quickly get predictions for new compounds.

What makes the whole setup even more useful was the fact that the RESTful web service deployed on the KNIME Server also had tests built in. All we now had to do was call the web service with an empty input table and the automatic tests would run. Going back to the workflow, some adjustments were made to the "Run Tests if Necessary" wrapped metanode to add a simple report if the workflow tests pass, then re-deployed it to KNIME Server, and ended up with a web service that tests itself if it is called with an empty input table. The figure below shows a snippet from the output of the prediction service when it's called in testing mode.


```
"outputValues": {
    "Test-results-17:20": {
        "table-spec": [
            {
                "Result": "string"
            }
        ],
        "table-data": [
            [
                "Tests passed"
            ]
        ]
    },
```


Figure 84. Test results in the output from the prediction web service when it is called with an empty input table, causing the validation tests to be run

In this way it is easy to test that the prediction workflow was generating the correct results *without even having to start KNIME Analytics Platform*!

Conclusion

We've demonstrated how to build and deploy a web service in KNIME that can be validated both in KNIME Analytics Platform itself and on the Server once it has been deployed. The KNIME workflow used here is available on the KNIME Public EXAMPLES server:

EXAMPLES/50_Applications/51_Model_Deployment_and_Validation/01_Deploying_and_Validating_models_as_WebServices

Chapter 8. Cybersecurity

8.1. Fraud Detection

By Kathrin Melcher

Access workflow on hub.knime.com

Or from: EXAMPLES/50_Applications/39_Fraud_Detection

According to the Nilson Report, a publication covering global payment systems, the global card fraud losses equaled $22.8 billion in 2016, with an increase of 4.4 percent over 2015. This shows the importance of the detection of credit card fraud. A number of options can be used to solve this problem.

1. Rely on domain experts and translate their knowledge into rules
2. Based on a set of historical data, train a machine learning model to quantify the probability of a transaction to be fraudulent.
3. Based on a set of legitimate historical transactions, train a model to predict them and then apply an anomaly detection technique to discover outliers, in this case fraudulent transactions

The first approach calls for expert knowledge, which we do not have. The conversion process from knowledge to rules can also prove to be a tricky one. The second approach requires a labeled dataset and the last approach relies on techniques for anomaly detection. The dataset we have available for this use case is a labeled dataset from Kaggle. Therefore, we will use the second approach. That is we train a model on historical transactions, labeled as either fraudulent or not, with a supervised learning algorithm.

Another important factor in predicting fraudulent transactions is the execution time by transaction during deployment. A fraudulent transaction must be detected in real time, or close to real time, to either block the credit card right away or call the customer to double check whether the transaction is legitimate.

What You Need

Even though episodes of credit card fraud are happening more and more frequently, and the money loss associated with them is usually very high, they are still rare events in terms of a priori probability. This means that any transaction dataset is highly unbalanced, since in general less than 1% of all transactions are actually frauds.

To train a machine learning model on a set of labeled examples, we need a dataset with historical transactions, some of which have been identified as fraudulent.

For the aim of this example, we use a dataset from Kaggle with credit card transactions over two days on September 2013 by European cardholders. This dataset shows that fraudulent transactions are rare, as only 0.173% of the transactions are fraudulent.

The dataset consists of a CSV file with 31 columns. 28 of them are the output of a Principal Component Analysis (PCA). Unfortunately, we do not have any information about the original features used to create the Principal Components (PCs). In addition to the PCs, we have the transaction amount, its time, and its fraudulent character as a binary feature (1 = fraudulent, 0 = legitimate).

The Workflow

Reading

First of all, we need to read in our data table. To read in the CSV file we use the File Reader node.

In the lower part of the configuration window of the File Reader node there is a preview of the data table we are reading. Next to the column header, the datatype of the column is reported. As the class column has only two values – 0 and 1 - it is by default set to be a numerical column. However, we will need to feed these two classes into the model training as String values. Thus, we change the column type, by double clicking the column header in the preview to change the datatype to String in an additional dialog.

Partitioning

As usual, we split the data in training set to train the model and test set to evaluate the model performance. We use the Partitioning node to divide the dataset in two separate partitions: the training set (usually 70-80 % of the dataset) and the test set.

For classification problems, like the problem at hand, it is important to ensure that all classes are present in the training set and the test set. Stratified sampling retains the value distribution from the original dataset for the selected column. In this particular case, where one class is much less frequent than the other, it is recommended to use the "Stratified sampling" relatively to the target column (Class). A random sampling indeed does not guarantee the presence of the least frequent class in both sets, with consequent bias in training and/or testing the model.

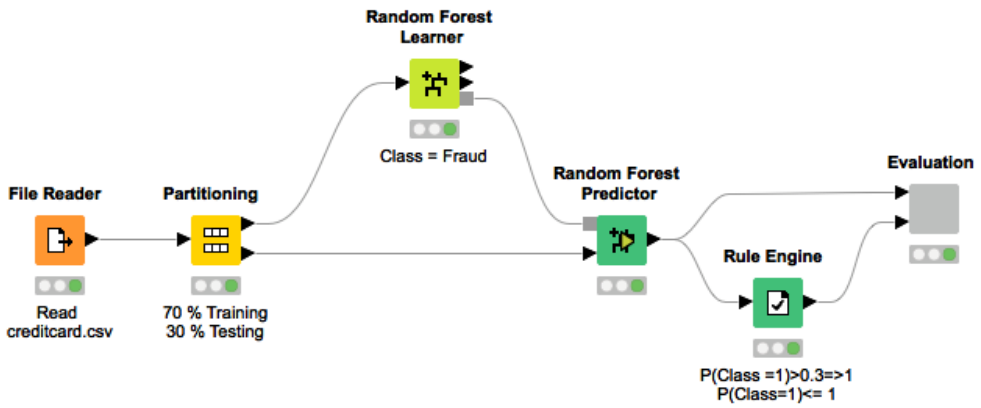


Figure 85. This workflow reads in the creditcard.csv file, trains and evaluates a Random Forest model to classify transactions as either fraudulent or not. Notice the final Rule Engine node. This node accepts all transactions where fraud probability above 0.3 as fraudulent.

Training the Model

Training a model in KNME Analytics Platform always uses the same motif. We have a learner node to train the model and a predictor node to apply the model. Classification problems can be solved by using supervised learning algorithms like the logistic regression, the decision tree, or an ensemble of decision trees, like the random forest, which was used here.

First, we feed the training set into the Random Forest Learner node to train the model. Then we feed the model, from the gray squared model output port of the Learner node, and the test set into the Random Forest Predictor node.

Note. The predictor node calculates the class probabilities and assigns each data row to one of the two classes applying a ***default threshold of 0.5*** on one of the class probabilities.

Evaluating the Model on a Class Unbalanced Test Set

To evaluate the model we use the Scorer (Javascript) node. As the data set is highly unbalanced, the accuracy mainly reflects the correct predictions of the most frequent class. Indeed, the accuracy has a very high value (99.95%) since

the most frequent class is often correctly predicted. In order to have a more realistic measure of the classification performance, also covering the least frequent class, we need to change the metric to the Cohen's Kappa (0.851), for example, or to an accuracy evaluated on a balanced test set (87.84%).

Alternatively, we can just take a look at the confusion matrix in Figure 86 on the left. Given the disparity in a priori distribution of the two classes, the model is actually performing quite satisfactorily. However, we can do better.

Find a Better Prediction Threshold

One option to improve our model's classification performance is to make it more biased towards the least frequent class, by changing the default threshold on the prediction probability. To do so we activate the checkbox "Append individual class probabilities" in the configuration window of the predictor node and then use a Rule Engine node to define a new threshold of 0.3 on the probability of the fraud class. Indeed, we assume a more conservative attitude towards fraud. Now a transaction with a probability of fraud just above 0.3 is considered potentially dangerous and requires double checking.

In Figure 86, we see the confusion matrix when using a threshold of 0.5 and a threshold of 0.3. It shows that by changing the threshold we detect six more fraudulent transactions in the test set. It also shows that we would need contact seven or more customers about a transaction that turned out to be legitimate. A considerable loss on a fraudulent transaction, though, might justify this conservative approach. This strategy is used in all of those use cases where one class carries a risk of an extremely negative consequence. In this case, a cost value is associated with the decision for the class with no negative consequences; i.e. a lower threshold is used to detect the riskier class.

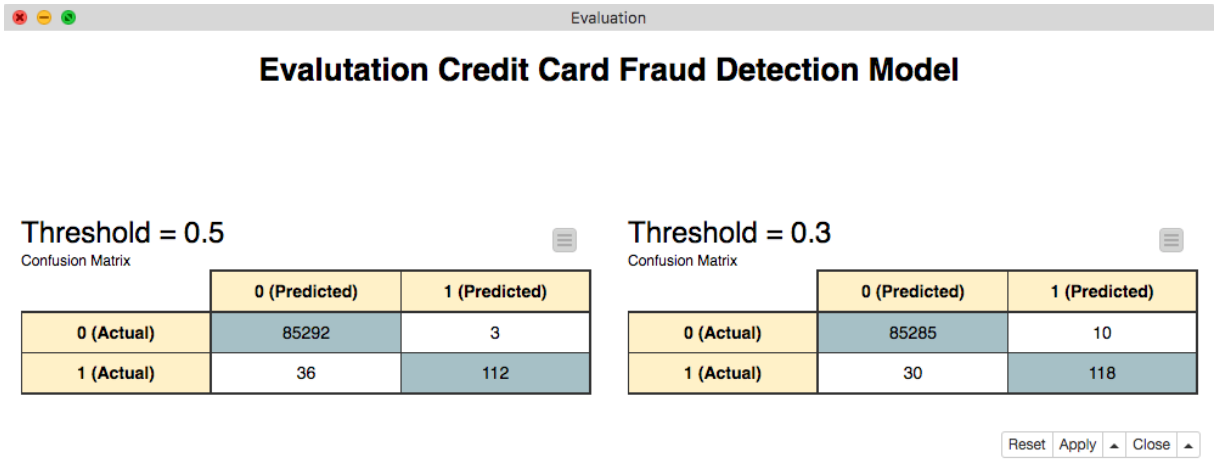


Figure 86. The two confusion matrices show the performance of the model for two prediction thresholds: 0.5 and 0.3. The model detects 6 more fraudulent transactions if the lower threshold is used. At the same time seven non-fraudulent transactions are wrongly classified as fraudulent.

Deployment Workflow and Real Time Performance

To apply the model to new transactions we use a second workflow: the deployment workflow. In this workflow, we read one new transaction with the Table Reader node, and the trained model with the Model Reader node. Then, we apply the model to the new transaction with the Random Forest Predictor node. In the next step, we use the Rule Engine node to apply the customized threshold. If the transaction is classified as fraudulent we send an email to the customer, otherwise nothing happens.

It is important that fraudulent transactions are detected before the transaction is approved. The Timer Info measures the execution time of each component in this workflow. The Math Formula node after that sums up the total execution time for the deployment workflow; i.e. how long it takes to classify one transaction. On a MacBook Pro (2016) with 16 GB RAM and CPU Intel Core i7-6920HQ CPU @ 2.90GHz, it took 45 milliseconds, which is way below 1 second and it feels like real time from a user perspective.

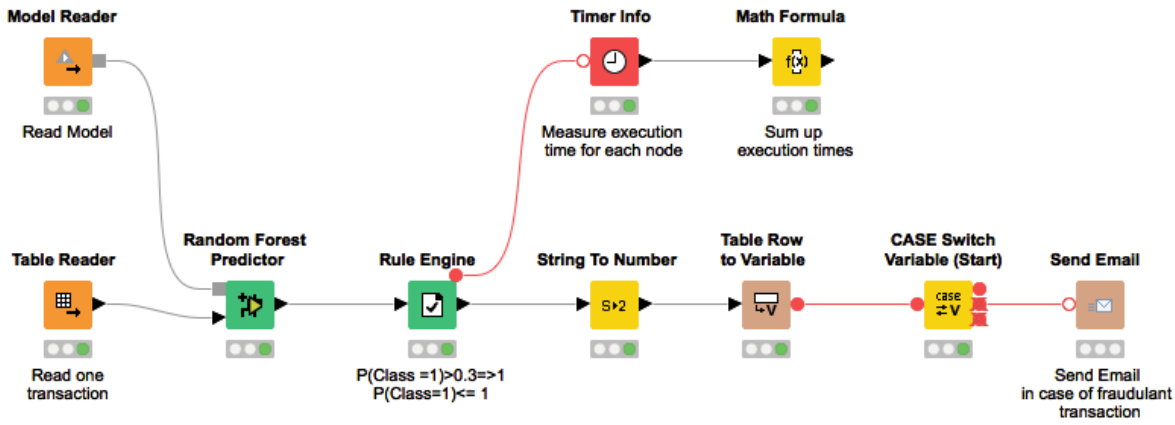


Figure 87. The deployment workflow reads the trained model, as well as the new transaction and applies the model to classify it. Like in the training workflow we use a Rule Engine node to apply the custom threshold. In case a transaction is classified as fraudulent the workflow sends an email to the customer.

8.2. Fraud Detection Using a Neural Auto-encoder

By Maarit Widmann, Kathrin Melcher

Access workflow on hub.knime.com

Or from: EXAMPLES/50_Applications/39_Fraud_Detection

Introduction

Fraud detection belongs to the more general class of problems, which is the anomaly detection. Anomaly is a generic, not domain specific concept. It refers to any exceptional or unexpected event in the data, be it a mechanical piece failure, an arrhythmic heartbeat, or a fraudulent transaction as in this study. Indeed, to identify a fraud means to identify an anomaly in the realm of a set of legitimate "normal" credit card transactions. Like all anomalies, we are never sure of the form a fraudulent transaction will assume. We need to be prepared for all possible "unknown" forms.

There are two main approaches to fraud detection.

- Based on the histograms or on the box plots of the input features, a threshold can be identified. All transactions with input features beyond that threshold will be declared fraud candidates (*discriminatory* approach). Usually, for this approach a number of fraud and legitimate transaction examples are necessary to build the histograms or the box plots.
- Using a training set of just legitimate transactions, we teach a Machine Learning algorithm to reproduce the feature vector of each transaction. Then we perform a reality check on such a reproduction. If the distance between the original transaction and the reproduced transaction is below a given threshold, the transaction is considered legitimate, otherwise it is considered a fraud candidate (*generative* approach). In this case, we just need a training set of "normal" transactions and we suspect an anomaly from the distance value.

Fraud detection using auto-encoders belongs to the *generative* approach for anomaly detection, where the anomalies appear as deviations from the reconstruction of the input data.

Let's apply this approach to fraud detection using a neural auto-encoder, following the approach described in the blog post "Credit Card Fraud Detection using Autoencoders in Keras — TensorFlow for Hackers (Part VII)" by Venelin Valkov.

What You Need

Credit Card Transactions Dataset

In this example, we use the credit card dataset provided by Kaggle, containing 284 807 credit card transactions performed in September 2013 by European cardholders. The transactions have two labels: 1 for fraudulent and 0 for legitimate (normal) transactions. 492 (0.2 %) transactions in the dataset are fraudulent.

Each transaction is represented by:

- 28 principal components extracted from the original data
- the time from the first transaction in the dataset
- the amount of money

Notice that the data contain principal components instead of the original transaction features, for privacy reasons. You could of course use any other normalized, numeric features to train the auto-encoder.

The Auto-encoder

An auto-encoder is a neural network, with as many output units as input units, and trained with the BackPropagation algorithm to reproduce the input data onto the output layer. During the training phase, only "normal" data are used, that is, data generated by the underlying system in normal conditions, without anomaly examples.

The idea is that, during deployment, on new data, the network will reproduce the input data of the "normal" class sufficiently well, while it will fail in reproducing input vectors of anomalies. Therefore, if during deployment we calculate the distance between the input and output vector, we will observe an exceptionally high deviation in case of an anomaly.

Figure 88 below shows the structure of a simple auto-encoder with only one hidden layer as introduced in this video Neural networks [6.1] : Autoencoder - definition by Hugo Larochelle.

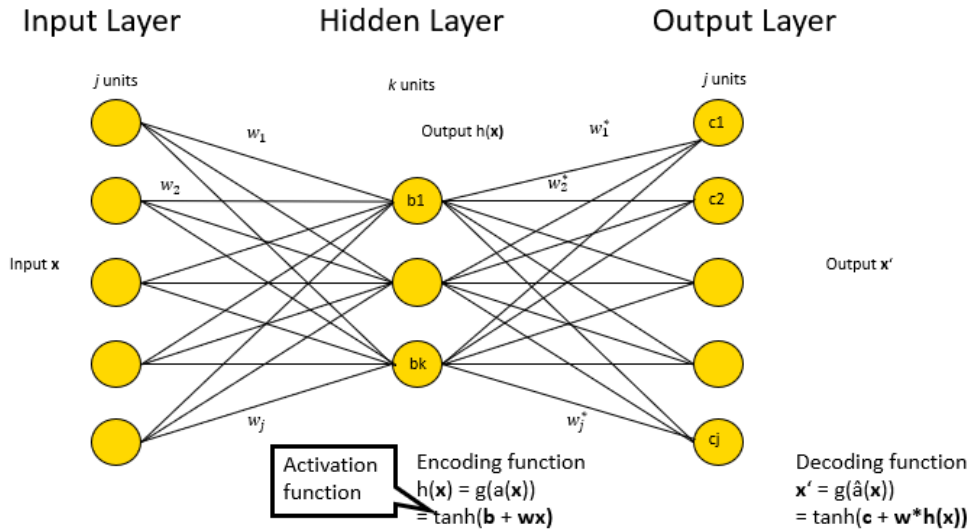


Figure 88. Architecture of a three-layer auto-encoder with one input layer with *j* units, one output layer with also *j* units, and one hidden layer with *k* units.

For this case study, we built an auto-encoder with 3 hidden layers, with number of units 30-14-7-7-30 and *tanh* and *reLU* as activation functions. The network was trained using the BackPropagation algorithm with the Mean Squared Error (MSE) as loss function. Many loss functions are available to train the network. You can even use a custom loss function if you don't want to use any of the standard ones.

The Anomaly Detection Rule

After the model has been trained, we apply it to new transactions. For each transaction x_k we calculate the distance between the original vector and the reconstructed vector, ε_k. Thus, a transaction x_k is a fraud candidate according to the following rule:

$$x_k \rightarrow \text{"normal" IF} \varepsilon_k \leq K$$
$$x_k \rightarrow \text{"anomaly"} \; IF \; \varepsilon_k > K,$$

where ε_k is the reconstruction error value and *K* is the selected threshold. The MSE was chosen again as reconstruction error measure. The threshold *K* has been optimized against sensitivity and specificity on a validation set.

Optimization of threshold *K* was made possible here by a few fraudulent transactions available in the original dataset. This optimization phase can be skipped if no labelled anomalies are available. In such case, the threshold can be defined as a high percentile of the reconstruction errors over the validation set.

The KNIME Keras Deep Learning Extension

One of the KNIME Deep Learning extensions integrates functionalities from Keras libraries, which in turn integrate functionalities from TensorFlow within Python.

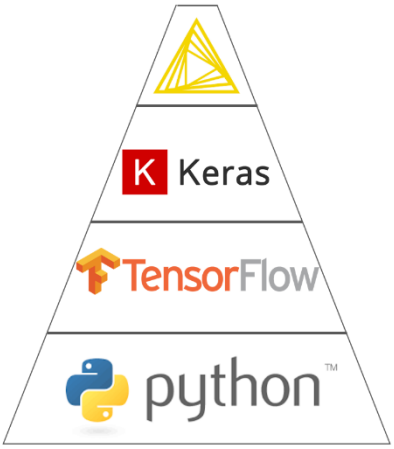


Figure 89. The Deep Learning Keras integration in KNIME Analytics Platform 3.7 encapsulates functions from Keras built on top of TensorFlow within Python.

In particular, the KNIME Deep Learning Keras integration utilizes the Keras' deep learning framework to read, write, create, train, and execute deep learning networks. This KNIME Deep Learning Keras integration has adopted the KNIME GUI as much as possible. This means that a number of Keras library functions have been wrapped into KNIME nodes, most of them providing a visual dialog window to set the required parameters.

The advantage of using the KNIME Deep Learning Keras integration within KNIME Analytics Platform is the drastic reduction of the amount of code to write, especially for pre-processing operations. Just by dragging and dropping a few nodes, you can build or import the desired neural architecture, which you can subsequently train with the Keras Network Learner node and apply with the DL Network Executor node. Just a few nodes with easy configuration rather than calls to functions in Python code.

Installation

In order to make the KNIME Deep Learning Keras integration work, a few pieces of the puzzle need to be installed:

- Python (including TensorFlow and Keras),
- KNIME Deep Learning Keras Extension

More information on how to install and connect all of these pieces can be found in the "KNIME Deep learning - Keras Integration" documentation page.

A useful video explaining how to install KNIME extensions can be found on the KNIME TV channel on YouTube.

Training the Auto-encoder

Data Preprocessing

Even though the data are ready to be analyzed, we still need a few preprocessing steps before training the network.

We need to:

1. Extract the “normal” transactions for the training set (Row Splitter node)
2. Create a validation set with ⅓ of the “normal” transactions and all remaining fraudulent transactions to optimize threshold *K* (first Partitioning node + Concatenate node)
3. Create a test set to evaluate the network reconstruction performances using 10% of the remaining “normal” transactions (second Partitioning node)
4. Normalize the input vectors from the training set to fall in [0,1] (Normalizer node), since neural networks perform better on normalized input vectors. The same normalization is then applied to all remaining transactions (Normalizer (Apply) node) and saved as a .model file to apply later to deployment data (Model Writer node).

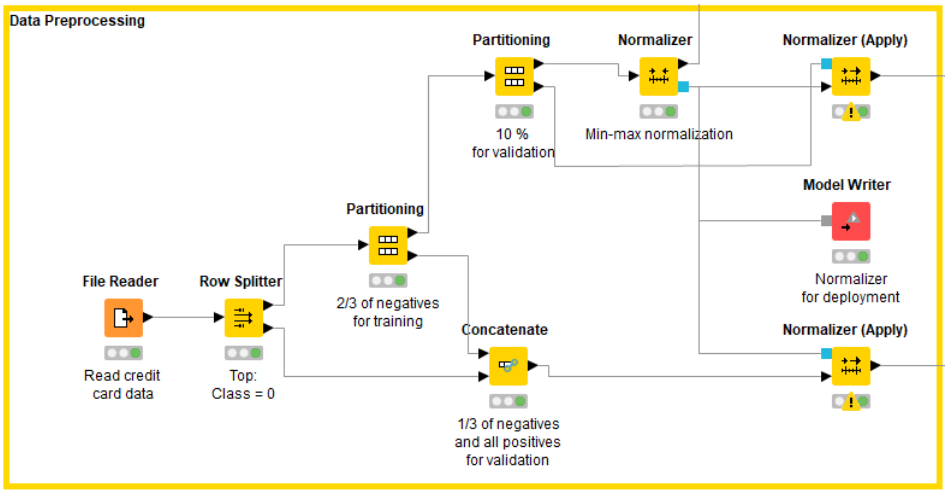


Figure 90. Data preprocessing to feed the neural auto-encoder. First we extract all “normal” data and then we partition them (⅔ -⅓). The ⅓ part together with all remaining fraudulent transactions forms the validation set. Next we perform the last partitioning (90%-10%) to create the training set and the test set for the auto-encoder. Then we normalize the data for the auto-encoder to fall into the [0,1] range. Finally we apply the normalization to the validation set and the test set for auto-encoder, and save the normalization parameters as a .model file.

Building the Auto-encoder Architecture

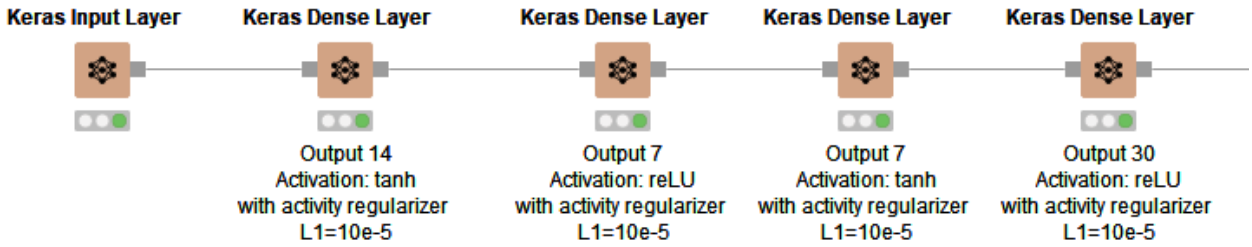


Figure 91. Structure of the neural network (30-14-7-7-30) trained to reproduce legitimate credit card transactions on the output layer

We built the network (30-14-7-7-30) completely codeless using the nodes from the KNIME Deep Learning - Keras Integration as:

- An input layer with as many dimensions (30) as the input data (Keras Input Layer node)
- A hidden layer that compresses the data into 14 dimensions using the *tanh* activation function and activity regularizer L1=0.0001 (Keras Dense Layer node)
- Another hidden layer that further compresses the data into 7 dimensions using the *reLU* activation function and activity regularizer L1=0.0001 (Keras Dense Layer node)
- A third hidden layer that transforms the 7 dimensions into other 7 dimensions using the *tanh* activation function and activity regularizer L1=0.0001 (Keras Dense Layer node)
- A final output layer supposed to reconstruct the input data into the original dimension (30) using the *reLU* activation function and activity regularizer L1=0.0001 (last Keras Dense Layer node)

The activity regularization parameter is a sparsity constraint, which makes the model less likely to overfit to the training data.

The number of training epochs is set to 50, the batch size is also set to 50, the loss function is set to the MSE, and Adam - an optimized version of the BackPropagation algorithm - is chosen as the training algorithm.

Training & Testing the Auto-encoder

The network is then trained and tested within the Keras Network Learner node, ending with final loss values in the range [0.070, 0.071], calculated using the following formula:

$$\varepsilon = \frac{1}{N}\frac{1}{n}\sum_{i=1}^{N}\sum_{j=1}^{n}(\hat{x}_{ij} - x_{ij})^2,$$

where *N* is the batch size and *n* is the number of units on the output layer.

After training, the network is saved for deployment as a Keras file using the Keras Network Writer node.

The value of the loss function though does not tell the whole story, because it just tells how well the network is able to reproduce "normal" input data onto the output layer. To have a full picture of how well this approach performs in detecting anomalies, we need to apply the anomaly detection rule from above to new data including anomalies: the validation set.

Optimizing the Threshold

The transactions from the validation set are now passed through the auto-encoder in the DL Network Executor node, in order to obtain the reconstructed feature vector.

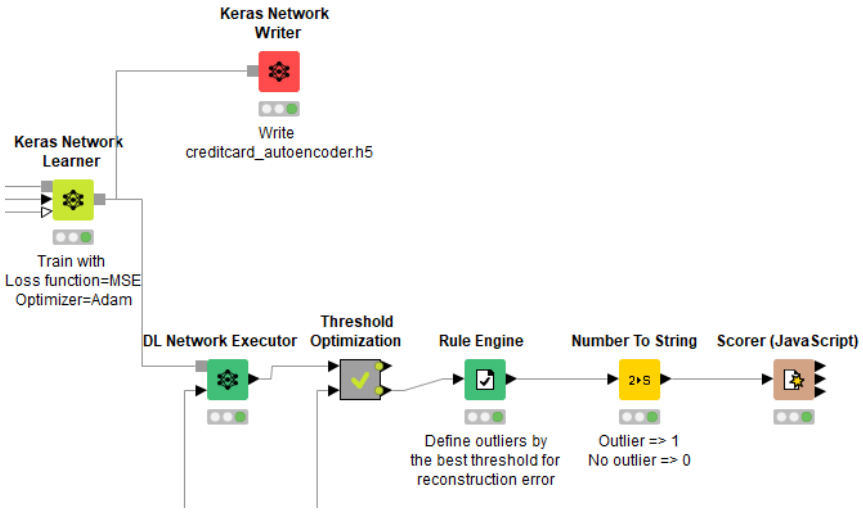


Figure 92. Training the auto-encoder and applying it to the validation set to optimize the value of threshold *K*.

Inside the "Threshold Optimization" metanode, the reconstruction errors ε between the true values x and the reconstructed values $\hat{x}$ are calculated using the following formula for MSE:

$$\varepsilon = \frac{1}{n}\sum_{i=1}^{n}(\hat{x}_i - x_i)^2,$$

where *n* is the number of units on the output layer. In the same metanode, threshold *K* is optimized against sensitivity and specificity to an optimal value of 0.009, in an optimization loop with threshold values ranging between 0.001 and 0.02 with step size 0.001.

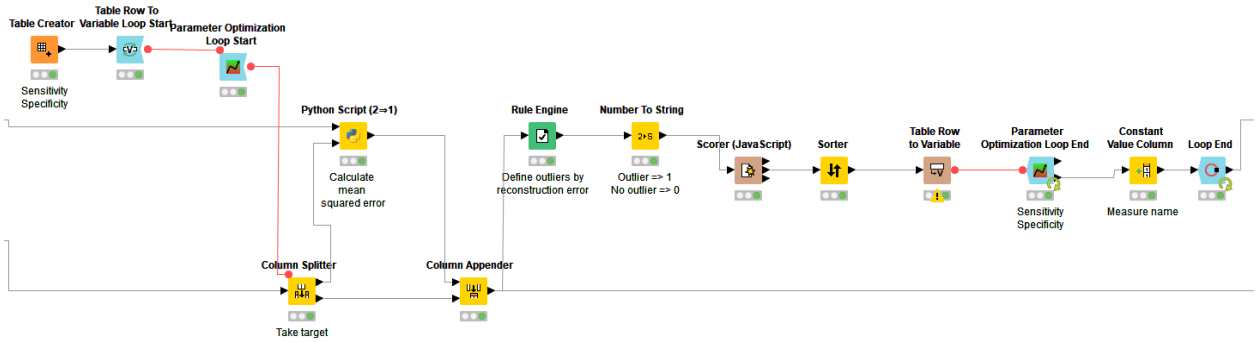


Figure 93. Content of "Threshold Optimization" metanode. The parameter optimization loop finds the optimal threshold *K* for the anomaly detection rule on the validation dataset. The parameter optimization loop runs two iterations: one to maximize the sensitivity and one to maximize the specificity. The optimal threshold is then defined as the value that results in equal performance using both criteria.

The parameter optimization loop has an outer loop which runs the parameter optimization loop twice: once to maximize the sensitivity and once to maximize the specificity.

Final performance for threshold *K*=0.009, as produced by the Scorer (JavaScript) node, is shown below.

Scorer View

Confusion Matrix

	0 (Predicted)	1 (Predicted)	
0 (Actual)	81784	14884	84.60%
1 (Actual)	65	427	86.79%
	99.92%	2.79%	

Class Statistics

Class	True Positives	False Positives	True Negatives	False Negatives	Recall	Precision	Sensitivity	Specificity	F-measure
0	81784	65	427	14884	84.60%	99.92%	84.60%	86.79%	91.63%
1	427	14884	81784	65	86.79%	2.79%	86.79%	84.60%	5.40%

Overall Statistics

Overall Accuracy	Overall Error	Cohen's kappa (κ)	Correctly Classified	Incorrectly Classified
84.61%	15.39%	0.045	82211	14949

Reset Apply Close

Figure 94. Performance metrics of the fraud detection model based on an auto-encoder network (30-14-7-7-30) and the threshold *K*=0.009.

As we can see, the auto-encoder captures 87 % (sensitivity) of the fraudulent transactions and 85 % (specificity) of the normal transactions in the validation set. Considering the high imbalance between the normal and fraudulent transactions in the validation data for threshold optimization, the results are promising.

The Final Workflow

The final workflow to build, train, and evaluate the neural auto-encoder, to prepare the data, and to optimize the value of threshold *K* is shown in Figure 95 and is downloadable for free from the KNIME EXAMPLES Server under: EXAMPLES/50_Applications/39_Fraud_Detection/Keras_Autoencoder_for_Fraud_Detection_Training

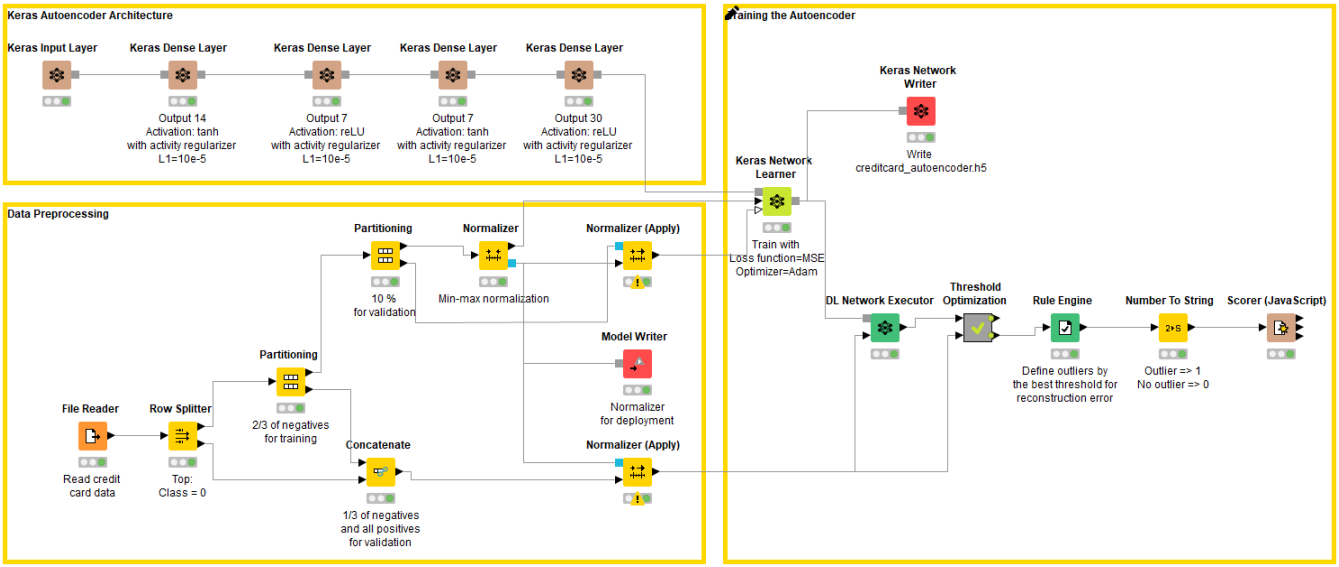


Figure 95. Final workflow to build, train, and evaluate the neural auto-encoder, to prepare the data, and to optimize threshold *K*.

Deployment

Now that we are left with a process (network + rule) with acceptable performance in reproducing the input data and in detecting anomalies, we need to implement the deployment workflow. Like all deployment workflows, this workflow will read new transaction data, pass them through the model and anomaly detection rule, and finally predict whether the current transaction is a fraud candidate or a legitimate transaction.

The deployment workflow is downloadable for free from the KNIME EXAMPLES Server under

EXAMPLES/50_Applications/39_Fraud_Detection/Keras_Autoencoder_for_Fraud_Detection_Deployment.

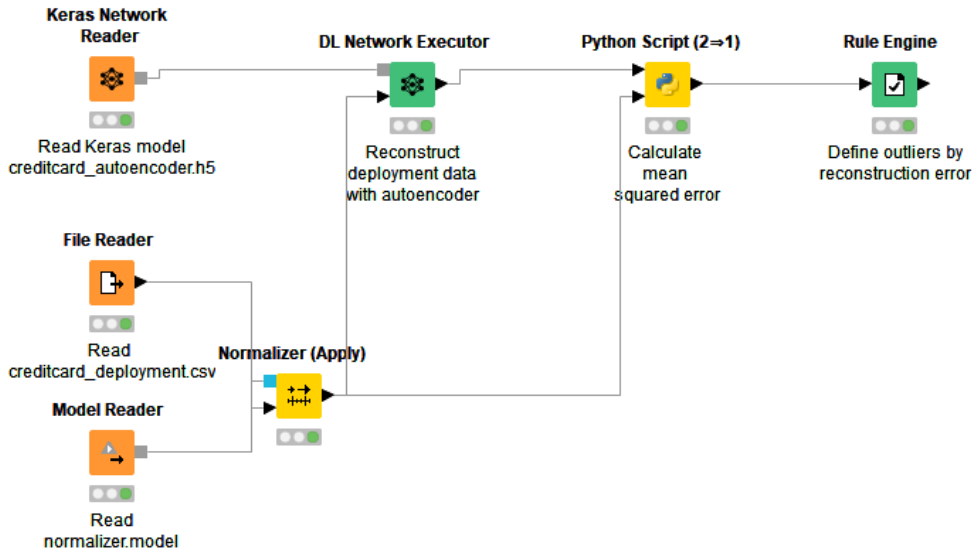


Figure 96. The deployment workflow reads the auto-encoder network from a Keras file and applies it to new credit card transactions. The data are normalized using the same parameters as for the training data. Reconstruction errors are calculated as mean squared errors between the original and the reconstructed data. If the reconstruction error is greater than threshold *K* = 0.009, the transaction is considered fraudulent.

First we read the auto-encoder model from the Keras file with the Keras Network Reader node, the deployment data with the File Reader node, and the normalization parameters with the Model Reader node. In order to reconstruct the

data, we pass them through the auto-encoder in the DL Network Executor node. Then, we calculate the MSE between the original features and the reconstructed features using the Python Script (2 ⇒ 1) node with the following script inside:

```
import pandas as pd
import numpy as np

table_1 = input_table_1.as_matrix()
table_2 = input_table_2.as_matrix()

mse = np.mean(np.power(table_2 - table_1, 2), axis=1)

output_table = pd.DataFrame({'reconstruction_error': mse}
           )
```

The workflow closes with the anomaly detection rule implemented in the Rule Engine node.

Conclusions

The auto-encoder approach to anomaly detection, and more precisely here to fraud detection, is an example of a generative approach.

The model is a neural network trained to reconstruct the input data onto the output layer, using only legitimate "normal" transactions (just because we have many of those). A distance measure is calculated to evaluate the quality of the data reconstruction and a rule is defined to recognize possible fraudulent transactions when the distance measure exceeds a threshold. The few fraud transactions available are used to optimize this threshold value.

The whole approach, starting from data preprocessing to training the model and optimizing the value of threshold *K*, was relatively straightforward. The innovative value of this approach is all in the usage of only "normal" transactions to train the network and use the distance to detect fraud candidates.

As usual, to improve performances, we could optimize the network parameters: try different activation functions and regularization parameters, a higher number of hidden layers and units per hidden layer, along with many other options.

The whole process could also be forced to lean on more towards frauds, by introducing an expertise-based bias in the definition of threshold *K*. Sometimes, in fact, it might be preferable to tolerate a higher number of check-ups on false positives than to miss even one fraudulent transaction!

Chapter 9. Text Generation

9.1 Neural Machine Translation with RNN

By: Rosaria Silipo, Kathrin Melcher, Simon Schmid

Access workflows on hub.knime.com - Training Workflow and Deployment Workflow

Or from. EXAMPLES/04_Analytics/14_Deep_Learning/02_Keras/12_Machine_Translation

Automatic Machine Translation has been a popular subject for machine learning algorithms. After all, if machines can detect topics and understand texts, translation should be just the next step.

Machine translation can be seen as a variation of natural language generation. In a previous project we worked on the automatic generation of fairy tales (see post "Once upon a Time … by LSTM Network" on KNIME blog [1]). Here a Recurrent Neural Network (RNN) with a Long Short Term Memory (LSTM) layer was trained to generate sequences of characters on texts from the Grimm's fairy tales. The result was a new text in a Grimm's fairy tale style.

In this use case, we want to extend this example of language generation to language translation. In particular, we want to address the need to make communication between the German and the US officer easier, by attempting an English to German translation experiment.

What You Need

A dataset that is suitable for this task can be downloaded free of charge from http://www.manythings.org/anki/. This dataset contains sentences that are used commonly in everyday life in both languages. It consists of two columns only: the original short text in English and the corresponding translation in German.

JavaScript Table View

10 entries — Search:

RowID	English	Translation
Row990	Why did you lie?	Warum fürgen Sie?
Row991	Write something.	Schreibe doch mal.
Row992	You almost died.	Du wärest fast gestorben.
Row993	You can make it.	Du kannst das machen?
Row994	You have cancer.	Sie haben Krebs.
Row995	You look stupid.	Du siehst dumm aus.
Row996	You should come.	Sie sollten kommen.
Row997	You'd better go.	Du solltest besser gehen.
Row998	You'll get lost.	Du wirst es helank angreiben.
Row999	You'll meet Tom.	Du wirst Tom treffen.

wing 991 to 1,000 of 1,000 entries — Previous 1 … 96 97 98 99 100 Next

Figure 97. An excerpt from the English <-> German translation dataset.

As for language generation, machine translation can be implemented at word level or at character level. In the first case, the next word is predicted given the previous sequence of words. In the second case, the next character is predicted given the previous sequence of characters. Since we have accumulated some experience in text generation at character level, we will continue here with machine translation at character level.

Again, as for language generation, an RNN with one (or more) LSTM layer(s) might prove suitable for the task. Here, however, we are dealing with two languages. Thus, we adopt a slightly modified neural architecture with two LSTM layers: one for the original language text and one for the target language text. This architecture is known as the encoder-decoder RNN structure.

The Encoder-Decoder RNN Structure

The task of machine translation consists of reading text in one language and generating text in another language. When neural networks are used for this task, we talk about Neural Machine Translation (NMT) [2][3].

Within NMT, the encoder-decoder structure is quite a popular RNN architecture. This architecture consists of two components: an encoder network that consumes the input text and a decoder network that generates the translated output text [2]. The task of the encoder is to extract a fixed sized dense representation of the different length input texts. The task of the decoder is to generate the corresponding text in the destination language, based on the dense representation from the encoder (Fig. 2). Usually both networks are RNNs, often LSTMs.

Figure 98. The Encoder-Decoder neural architecture.

Let's choose an LSTM layer for the encoder and decoder networks. An LSTM network is a particular type of RNN, which relies on internal **gates**, to selectively remove (*forget*) or add (*remember*) information from/to the cell state *C(t)* based on the input values *x(t)* and the hidden state *h(t−1)* at each time step *t* (see post "Once upon a Time … by LSTM Network" on the KNIME blog for more details on RNNs and LSTM layers [1]).

A common way of representing RNNs is to unroll them into a sequence of copies of the same static network A, each one fed by the hidden state of the previous copy *h(t−1)* and by the current input *x(t)*. The unrolled RNN can then be trained with the Back Propagation Through Time (BPTT) algorithm [4]. To implement, train, and deploy the RNN with the LSTM layers using BPPT we use the "KNIME Deep learning - Keras Integration".

The Training Workflow

In theory, training should be performed using the predicted character from the previous step. In practice, however, you can obtain better training performances using the so-called "teacher forcing" technique. Here, the actual previous character instead of the previous prediction is fed into the next network copy, which greatly benefits the training procedure [3, p.372]. This forced feeding of the actual characters must be removed in the deployment workflow.

Therefore, we need three different character sequences to train the network:

1. The input sequence into the encoder, i.e. the index-encoded input character sequence for the original language.
2. The input sequence into the decoder, i.e. is the index-encoded character sequence for the translation language.
3. The target sequence, i.e. the input sequence to the decoder shifted by one step.

Remember that a network doesn't understand characters, but only numerical values. The character input sequences need to be transformed into numerical input sequences via one of the many text encoding strategies available.

The training workflow covers all the required steps:

- English and German text pre-processing
- Network structure definition for the encoder and the decoder
- Network training
- Network structure modification for both encoder and decoder via Python editing
- Model deployment and evaluation

This training workflow can be found on the KNIME EXAMPLES server under
04_Analytics/14_Deep_Learning/02_Keras/12_Machine_Translation/NMT_Training

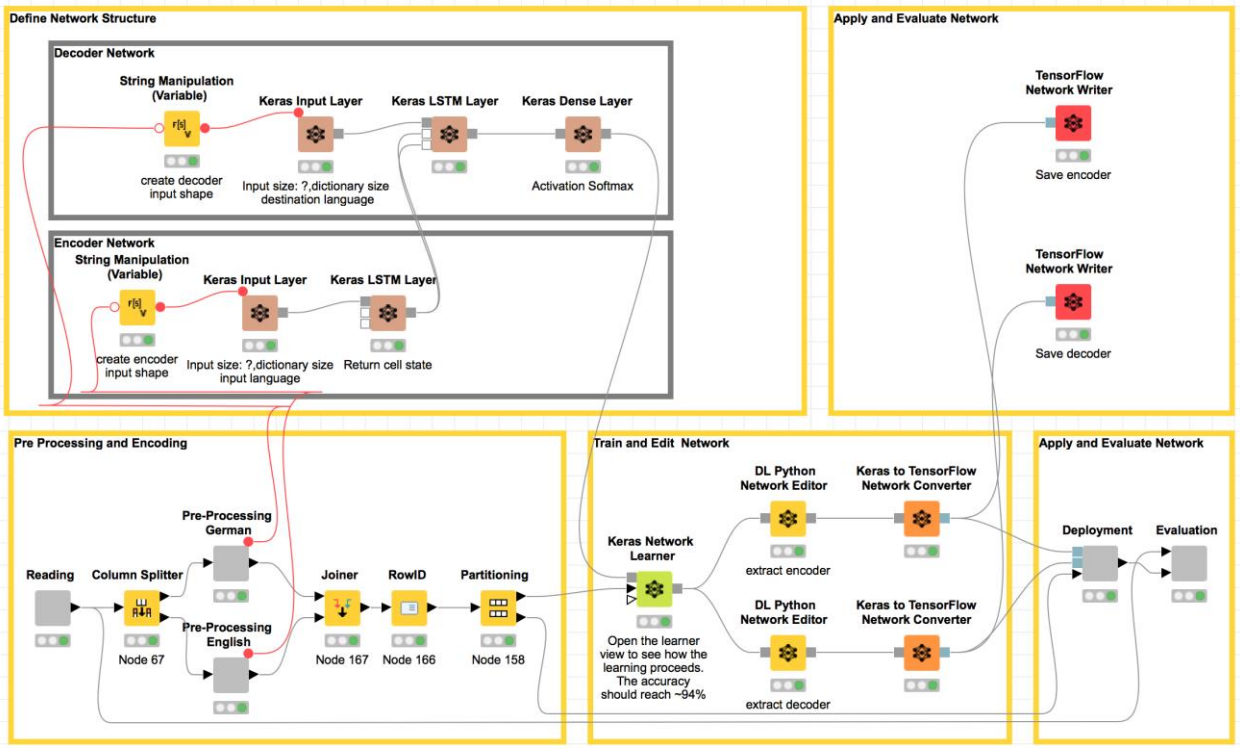


Figure 99. Workflow to build, train, and evaluate a neural network for automatic translation from English to German. In the top left frame, the brown nodes, each add one layer to the neural network for the training phase: an input layer and an LSTM layer for the English (encoder) and German (decoder) text respectively, one dense layer to produce character probabilities at the end. The frame under the network construction nodes transforms the English and German texts into index based encoding sequences. Still in the lower part, nodes in the central frame train the whole network and then split it into encoder and decoder via DL Python Network Editor nodes. Finally, Deployment and Evaluation nodes apply the network to the test set and evaluate the translation results.

Defining the Network Structure

The upper part of the workflow, including the brown nodes, defines the neural network structure. The brown nodes each build one type of layer in the neural network.

The **encoder** network is made of two layers:

1. An **input layer** implemented via the Keras Input Layer node. This layer accepts input tensors of integer indexes, with variable sequence length and same size as the character dictionary size of the input language (English). In our example, the input tensor for English has size [?, dictionary size input language], i.e. [?,70].
2. An **LSTM Layer** via a Keras LSTM Layer node: In this node, the checkbox “return state” must be enabled, to pass the hidden states to the upcoming decoder network.

The **decoder** network is made of three layers:

1. An **input layer** via a Keras Input Layer node. This layer defines the shape of the input tensor, with variable sequence length and same size as the character dictionary size of the input language (German). In our example, the input tensor for German has size [?, dictionary size destination language], i.e. "?, 85".
2. An **LSTM layer** via a Keras LSTM Layer node. This layer has two inputs: the character sequence for the text in the destination language and the hidden states from the encoder LSTM layer. In its configuration window, the checkboxes "return sequence" and "return state" are both enabled to return the hidden state as well as the next character prediction.
3. Last a **dense layer** via a Keras Dense Layer node to produce the probability vector for all dictionary characters. In the configuration window, the activation function softmax is selected to have 85 units, i.e. the size of the dictionary of the destination language.

Text Preprocessing and Encoding

In the lower left corner of the workflow, we find the nodes dedicated to text pre-processing and encoding. Remember that we have two texts to pre-process and encode: one in English and one in German. We use index based encoding for both of them.

In addition, all input sequences need a Start Token, set to 1, to mark the beginning of the character sequence. All target sequences need an End Token, set to 0, at the end, to mark the end of the character sequence. In addition, since all character sequences must have the same length as defined by the corresponding input layer, zero-padding and truncation are also applied where needed. All of these operations are implemented in the gray nodes named "Pre-processing German" and "Pre-processing English" for German and English respectively.

These same encoding and pre-processing steps are described in detail in the "Once Upon A Time … by LSTM Network" blog post [1]. You will also find information about these steps in Chapter 9.1 on Product Naming.

Training and Editing the Network

In the middle lower part of the training workflow, the network is trained with the BPTT algorithm by executing the Keras Network Learner node.

Note that the output of the dense Softmax layer gives the probability distribution for all index-encoded characters of the German language. The predicted character can be selected as the one with the highest probability or via an additional lambda layer.

After the network has been trained, it needs a bit of a post-processing before deployment, e.g.:

- Separate the encoder and decoder part of the network
- Introduce a lambda layer with an argmax function to select one of the characters with the highest probability in the softmax layer. The lambda layer adds noise into the selection process, which then leads to a less deterministic character selection.

The DL Python Network Editor nodes allow to make those changes to the network structure with a relatively simple Python code snippet. The Python code to extract the decoder network and to introduce the lambda layer is reported here.

```
# variable name of the input network: input_network

# variable name of the output network:  output_network

from keras.models import Model

from keras.layers import Input, Lambda
```

```
from keras import backend as K

state1 = Input((256,))

state2 = Input((256,))

new_input = Input((1,85))

decoder_lstm = input_network.layers[-2]

decoder_dense = input_network.layers[-1]

x, out_h, out_c = decoder_lstm(new_input, initial_state=[state1, state2])

probability_output = decoder_dense(x)

argmax_output = Lambda(lambda x: K.argmax(x, axis=-1))(probability_output)

output_network = Model(inputs=[new_input, state1, state2], outputs=[probability_output,
argmax_output, out_h, out_c])
```

For faster deployment execution, the encoder and decoder Keras networks are converted to TensorFlow and saved separately.

The Deployment Workflow

In the deployment workflow (Fig. 5) English sentences have to pass through the encoder and the decoder networks in two subsequent steps. Two TensorFlow Network Reader nodes each read the encoder and the decoder network respectively.

Two DL Network Executor nodes feed the two networks with the input data table of English sentences.

The first DL Network Executor node presents an English character to the encoder and produces the corresponding hidden state. The second DL Network Executor node presents this hidden state from the encoder to the decoder network to produce the predicted index-encoded character for the German sentence.

Finally, the German dictionary is applied to extract the character from index-based encoding to get the final translated sentence.

Notice the last node: the “Container Output (JSON)” node. This node produces a JSON structured output containing the translation text. This deployment workflow was indeed used as a REST service, producing a JSON structured response with the translation text.

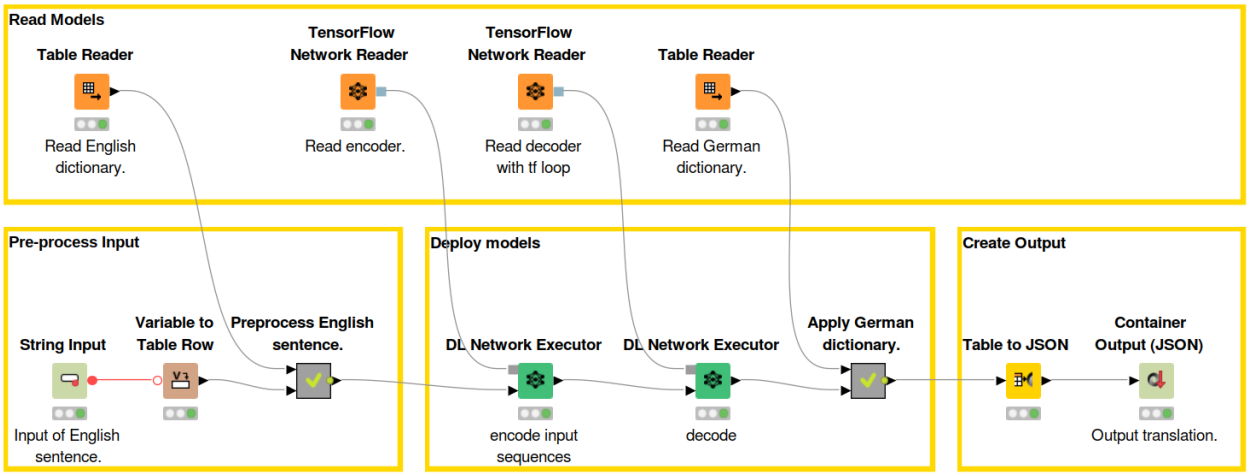


Figure 100. The deployment workflow applies the encoder and decoder network to a new English sentence. Notice that the encoder and decoder networks are retrieved from two separate TensorFlow files.

The deployment workflow can be found on the KNIME EXAMPLES server under *04_Analytics/14_Deep_Learning/02_Keras/12_Machine_Translation/NMT_Deployment.*

Below is an excerpt of the translation results.

English	German	Translation
You run.	Du läufst.	Sie laufen.
You won.	Du hast gewonnen.	Sie gens!
She blushed.	Sie errötete.	Sie lief rot an.
Sit with me.	Setz dich zu mir!	Wirgen Sie mir!
Start again.	Fang noch einmal an.	Beginnen Sie noch einmal.
Stop trying.	Probieren Sie es nicht länger.	Probier es nicht länger.
Take a walk.	Geh spazieren!	Geht spazieren!
Talk to Tom.	Sprich mit Tom!	Rune mir es Tom!
That's cool.	Das ist cool.	Das ist geil.
Be brave.	Sei tapfer!	Seid tapfer!

Figure 101. Final results of the deployed translation network on new sentences. "English" is the input sentence, "German" is the proposed target, and "Translation" the actual German text produced by the network.

Conclusion

We explored the topic of Neural Machine Translation, specifically on a translation task from English to German. In order to solve this problem we used the KNIME Deep Learning Keras integration and implemented an almost codeless deep learning workflow via the KNIME GUI.

Translation results are so far satisfactory, but could still be improved with more data, parameter optimization, and longer network training. Cheers! Zum Wohl!

References

[1] Kathrin Melcher, “Once upon a Time … by LSTM Network”, KNIME Blog, Nov 26 2018 https://www.knime.com/blog/text-generation-with-lstm

[2] Yonghui Wu, et al. “Google’s Neural Machine Translation System: Bridging the Gap between Human and Machine Translation”, arXiv:1609.08144v2 [cs.CL] 8 Oct 2016 https://arxiv.org/pdf/1609.08144.pdf

[3] Prakhar Mishra, “A Concise Introduction to Neural Machine Translation“ https://hackernoon.com/a-concise-introduction-to-neural-machine-translation-d925ee1ff5df

[4] Ian Goodfellow, Yoshua Bengio and Aaron Courville, “Deep Learning”, The MIT Press, 2016

9.1 Product Naming with Deep Learning for Marketers and Retailers – Keras Inspired

By Kathrin Melcher and Rosaria Silipo
Access workflows on hub.knime.com – Training Workflow and Deployment Workflow
Or from:
EXAMPLES/04_Analytics/14_Deep_Learning/02_Keras/10_Generate_Product_Names_With_LSTM

Usually, data science is applied to automatize repetitive tasks or to predict new classes based on a set of examples. Let's move to a more creative usage of data science and, in particular, of artificial intelligence.

Let's take a classic creative marketing task: product naming. The moment a product is ready to be pushed out onto the market, the most creative minds of the company come together to generate a number of proposals for product names that must sound familiar to the customers and yet are new and fresh too. Of all those candidates, ultimately only some will survive and be adopted as the new product names.

To help the marketing department, we asked a many-to-many Recurrent Neural Network (RNN), with a hidden layer of LSTM (Long Short Term Memory) units, to generate name candidates for a new outdoor clothing line. As we want to find names for outdoor apparel, the idea is to generate mountain inspired names, as many other outdoor fashion labels do.

A requirement in product naming is to generate names that are on one side realistic enough to stand out in the market and on the other side original enough to differentiate from similar competitor products. Fictive mountain names should do.

What You Need

First of all we need a set of mountain names on which to train the network.

We built a list with 33,012 names of mountains in the US, as extracted from Wikipedia through a Wikidata Query. Mountain names are strings and presented to the network as sequences of characters, one character per "time step". Each sequence needs a trigger character, named Start Token, and an end character, named End Token. We arbitrarily chose "1" as the Start Token and "0" as the End Token.

The neural network would be fed with a list of characters and will be trained to predict the next character in the name. If for example "Red Hill" is in the training set, the input could be ("R", "e", "d", " ", "H") and the network should learn to predict "i" from that input sequence; "l" from the input sequence ("e", "d", " ", "H", "i"); and so on. In this example we have implicitly used an input sequence of 5 past characters. However, the number *m* of past characters can be arbitrary. We set the number of past input characters to *m=256* for this particular use case. This can be changed and optimized to better fit similar new use cases.

Still, a neural network doesn't understand characters but operates on numbers. Therefore, all characters must be encoded into numbers, for example one-hot encoded. In one-hot encoding, each character is represented via a vector with size as the number of total characters in the dataset. Each character is assigned a position in the vector, which assumes value 1 or 0 depending on the presence or absence of that character.

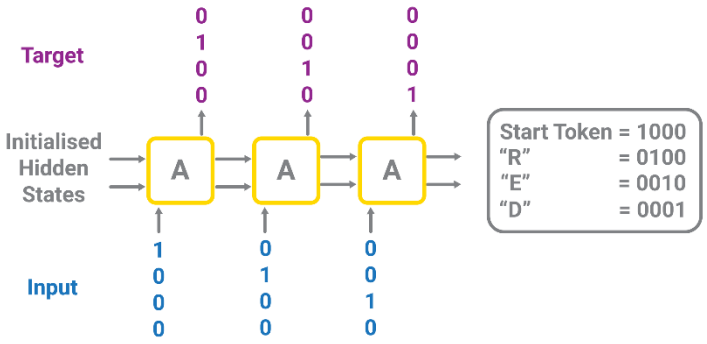


Figure 102. Neural networks operate on numerical vectors/tensors. Therefore, characters are one-hot encoded.

Now, we need a neural network architecture to train. We adopted the following Recurrent Neural Network (RNN) structure, with *n* input units, *256* LSTM units, *n* dense linear units, and *n* Softmax output units, where *n=95* is the number of the characters in the dataset (aka the dictionary) and the size of the one-hot encoding vector [1].

The output layer consists of the same number of units as the input layer, because we want it to produce the probabilities for all characters to follow the input sequence. We can then choose the predicted character from the set of characters with the highest probability for example via an additional Lambda Layer introduced for deployment.

Indeed, RNNs and especially LSTM units, thanks to their recurrent connections, have been reported as performing well in time series prediction problems, such as for example the prediction of character sequences [2] [3] [4] [5].

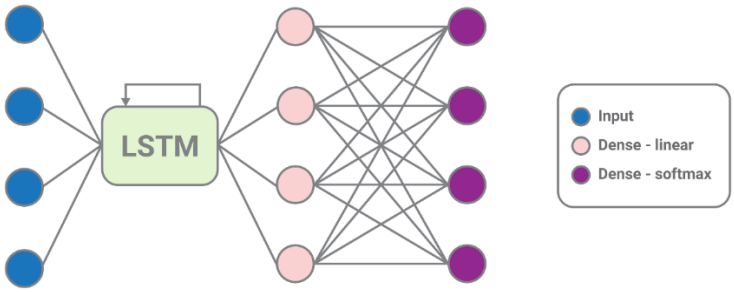


Figure 103. The adopted network structure, with an input layer, an LSTM layer, a linear dense layer, and a softmax dense layer. The last two layers have been inserted to allow for the introduction of the temperature parameter

The Training Workflow

In order to read and encode the mountain names, and to define, train, and execute the many-to-many LSTM network, we used the Keras Integration available within KNIME Analytics Platform.

The workflow that implements the example described in this article is shown in Figure 104. At the top you can see the brown nodes that build the layers in the RNN structure. Below, from left to right, you can see a grey node that reads, pre-processes, and index-encodes the mountain names in the input data, the node implementing the network training algorithm. Finally the last three nodes prepare the network for deployment, transform it to TensorFlow format, and save it to a file.

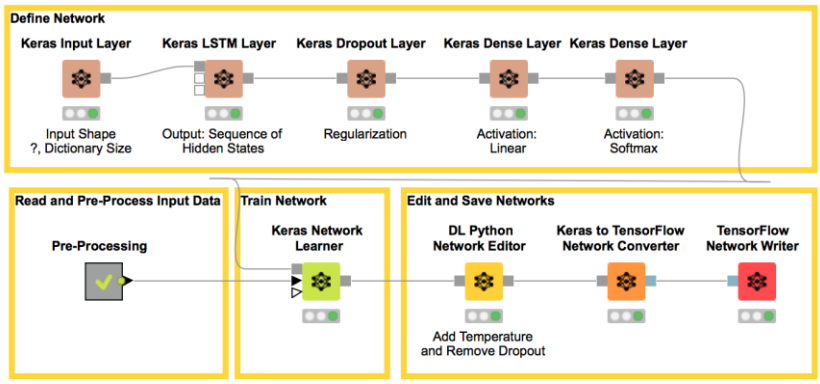


Figure 104. This workflow builds, trains, and saves an RNN with an LSTM layer to generate new fictive mountain names. The brown nodes define the network structure. The Keras Network Learner node trains the network using index-encoded original mountain names from the node named "Pre-Processing". Finally, the trained network is prepared for deployment, transformed into TensorFlow format, and saved to a file.

Defining the Network Structure

The Basic Structure

The network we use has:

- an input layer to feed the one-hot encoded character into the network.
- an LSTM layer for the sequence analysis.
- two dense layers to extract the probabilities for the output characters from the hidden states $h(t)$

The number of different characters in the training set is *95*. Therefore, the size of the input layer is "?, 95". The "?" stands for a variable input length. The possibility of having a variable length input is indeed one of the biggest advantages of RNNs. The output layer also has *95* units.

Introducing Temperature

We could use only one dense layer with SoftMax activation function. However, to change the confidence of the network after training, we decided to introduce temperature τ.

Note: SoftMax is defined as:

$$P(y = j|x) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \text{ with } z_i = x^T w_i.$$

Temperature τ is introduced after the linear part z_i of SoftMax as a dividing factor:

$$P(y = j|x) = \frac{e^{z_j/\tau}}{\sum_{k=1}^{K} e^{z_k/\tau}} \text{ with } z_i = x^T w_i.$$

$\tau < 1$ makes the network more confident, but also more conservative. This often leads to generating the same fictive names. $\tau > 1$ leads to softer probability distributions over the different characters. This leads to more diversity, but at the same time also to more mistakes, e.g. character combinations that are uncommon in English [4].

Temperature $\tau = 1{,}2$ is introduced through a Lambda Layer applied in between the linear and the SoftMax layer in the trained network via a Python snippet code before deployment. Thus, during training, the linear component of the SoftMax must be separated into an additional dense layer. This is obtained with two subsequent dense layers in the network, one with Linear and one with SoftMax activation function.

The Dropout Layer

The network was trained with an additional drop-out layer to prevent overfitting and to allow for better generalization. Dropout consists in randomly setting a fraction of input units to 0 at each iteration during training time, which helps prevent overfitting. This layer is then removed when the network is deployed.

Preprocessing and Encoding

We now need to prepare the data to feed into the network. This is done in the pre-processing and encoding part of the workflow. Here, mountain names are read and transformed into sequences of characters, characters are encoded as indexes and assembled into collection cells to form the input and target sequences.

The longest mountain name in the training set has 58 characters. Therefore, the input sequence contains the start token plus the mountain name as a sequence of 58 indexes. The target sequence contains the same 58 indexes of the characters forming the mountain name plus a zero in the end. The zero in the end serves as the end token on one hand; on the other hand it ensures that the input sequence and the output sequence have the same length. For the purpose of training, the sequence length in one batch has to be the same, therefore we zero pad all those mountain names with less than 58 characters.

The biggest advantages of RNNs is that they can handle sequences of different length. However, for the training the sequence length in one batch has to be the same. Therefore, we zero pad all mountain names with less than 58 characters.

In the process, a dictionary is also created to map indexes with characters.

The training node of the KNIME Keras integration automatically converts these index-encoded sequences into one-hot encoded sequences as expected by the network.

Training the Network

The Keras Network Learner node then trains the network. In the node configuration window we set the conversion from a list of integers (the indexes) to a list of one-hot encoded vectors and all the training parameters.

In the last step we save the trained network. Before we do so, we modify the network structure to remove the dropout layer and to introduce the temperature $\tau = 1{,}2$ via a Lambda Layer. For this modification, we use some Python code in the DL Python Network Editor node.

For faster prediction, we convert the network from a Keras model into a TensorFlow model.

The newly converted network structure, TensorFlow formatted, is then saved to a file for further usage in deployment.

The final training workflow is available free of charge on the KNIME EXAMPLES Server under 04_Analytics/14_Deep_Learning/02_Keras/10_Generate_Product_Names_With_LSTM/01_Training.

The Deployment Workflow

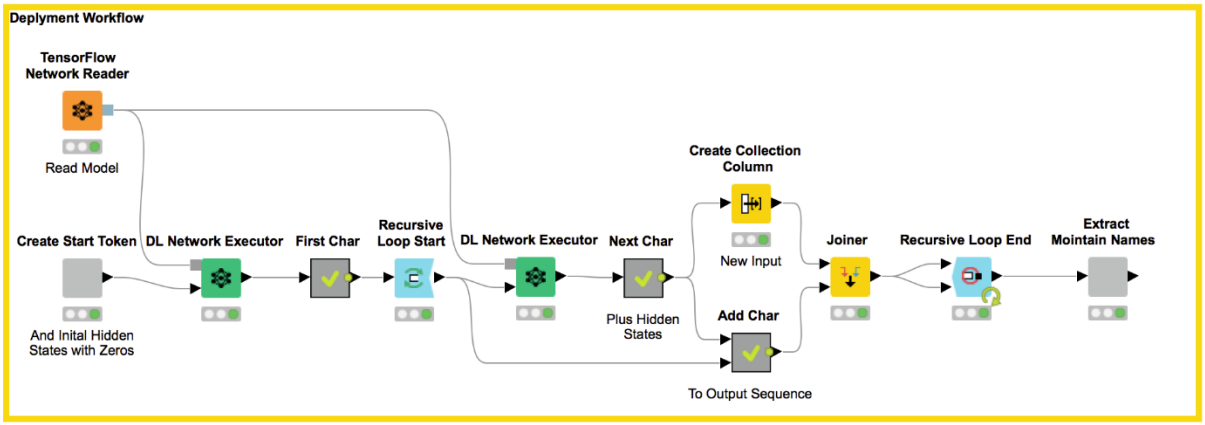


Figure 105. The deployment workflow generates 200 new, fictive mountain names. It reads the previously trained TensorFlow network and predicts 200 sequences of index-encoded characters within a loop. The last node, named Extract Mountain Names translates the sequence of indexes into characters and visualizes the new fictive mountain names.

In Figure 105. The deployment workflow generates 200 new, fictive mountain names. It reads the previously trained TensorFlow network and predicts 200 sequences of index-encoded characters within a loop. The last node, named Extract Mountain Names translates the sequence of indexes into characters and visualizes the new fictive mountain names.you see the deployment workflow. It first creates a list of 200 Start Tokens and the initial hidden states (all 0s). Then it reads the trained network and executes it multiple times to generate new mountain names via the DL Network Executor node, character by character.

For the first character, the network is executed on the start token and the initialized hidden states. The output of the network is a probability distribution for the different index-encoded characters. We can now either choose the index with the highest probability or we pick the output character according to this probability distribution. The deployment workflow uses the second approach, as the first approach would always predict the same name.

To predict the next index the network is executed again. This time the input is the last predicted index and the hidden states of the last execution. With regards to output, we get the probability distribution for the index-encoded character. And we continue in the same way.

The deployment workflow is available free of charge from the KNIME Community Workflow Hub and from the KNIME EXAMPLES Server under 04_Analytics/14_Deep_Learning/02_Keras/10_Generate_Product_Names_With_LSTM/02_Deployment

We set the deployment workflow to create N=200 new mountain names. Below, you can see the list of our personal favorites, like Morring Peak or Rich Windplace.

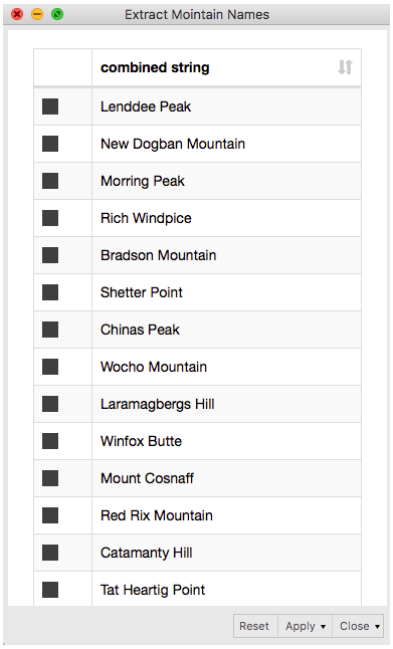


Figure 106. My personal favorites from the output list of fictive mountain names

Summary

We asked AI to help produce a list of fictive mountain names to label the new line of outdoor clothing ready for the market. We produced 200 mountain name candidates to help marketing in the brainstorming phase. These 200 names are just candidates to provide inspiration for the final names of the new products. Human marketers can then select the newest and most adventure-evocative, yet familiar, names to label the new line.

A valid help in putting together a number of name candidates came from an RNN, and specifically an LSTM network many-to-many network. We trained the network on 33,016 names of existing mountains, all represented as sequences of index-encoded characters.

Next, we deployed the network to generate 200 new fictive mountain names. One of the good aspects of this project is that it can automate the repetitive part of the creative process for as many fictive mountain names as we want. Once trained, the network could generate any number from 10 to even 5000 candidate names for our outdoor clothing line.

You can download the workflow for free from the KNIME EXAMPLES Server under 04_Analytics/14_Deep_Learning/02_Keras/10_Generate_Product_Names_With_LSTM. By changing the real names in the dataset you can train the network to get new inspirational names for whatever you need!

References

[1] Kathrin Melcher, "Once upon a Time … by LSTM Network", KNIME Blog, 2018

[2] Jason Brownlee, "Crash Course in Recurrent Neural Networks for Deep Learning", Machine Learning Mastery Blog

[3] Zachary C. Lipton, John Berkowitz, Charles Elkan, "A Critical Review of Recurrent Neural Networks for Sequence Learning", arXiv:1506.00019v4 [cs.LG], 2015

[4] Andrej Karpathy, "The Unreasonable Effectiveness of Recurrent Neural Networks", Andrej Karpathy blog, 2015

[5] Ian Goodfellow, Yoshua Bengio and Aaron Courville, "Deep Learning", The MIT Press, 2016

M

N

P

R

S

T

U

W

X

Practicing Data Science

A Collection of Use Cases

About the Editor

Dr Rosaria Silipo has been mining data since her master degree in 1992. She kept mining data throughout all her doctoral program, her postdoctoral program, and most of her following job positions. She has many years of experience in data analysis, reporting, business intelligence, training, and writing. In the last few years she has been using KNIME for all her data science consulting work, becoming a KNIME trainer and an expert in the KNIME Reporting tool.