Humanitas

A prediction tool for volatile commodity prices in developing countries

Students: Alexander John Busser, Anton Ovchinnikov, Ching-Chia Wang, Duy Nguyen, Fabian Brix, Gabriel Grill, Julien Graisse, Joseph Boyd, Stefan Mihaila

Time Series data

Different sources Price categories: Retail prices, wholesale prices

Wholesale prices

Wholesale price index

(taken from investopedia.com)
An index that measures and tracks the changes in price of goods in the stages before the retail level. Wholesale price indexes (WPIs) report monthly to show the average price changes of goods sold in bulk, and they are a group of the indicators that follow growth in the economy.

Although some countries still use the WPIs as a measure of inflation, many countries, including the United States, use the producer price index (PPI) instead.

Data sources

Website of Ministry of Agriculture of India (http://agmarknet.nic.in/) was used for our purposes, which allows querying daily range of wholesale prices and stock arrival dating back to 2005 at market places throughout India.

Due to large amount of data available, as well as time and resources needed to pull it, several major products (Onion, Rice, Wheat, Apple, Banana, Coriander, Potato, Paddy(Dhan), Tomato) were chosen, whereat the Python script was used to download all available data for the selected products directly from the website, using simple HTTP GET requests. Raw HTML data was then converted to CSV format with predefined structure and saved for later processing.

Retail prices

Data sources

1. Daily retail prices

   Daily retail prices were found at the website, created by Department of Consumer Affairs of India (http://fcainfoweb.nic.in/). One can choose the type of report (price report, variation report, average/month end report) and the desired date, and then receive the requested data in HTML format. This website was used to query price reports from 2009 to 2014, for 59 cities and 22 products. Similar to the wholesale daily prices, raw HTML data was

converted to CSV files in normalized format, which was intended to facilitate further processing.

2. Weekly retail prices

   Retail Price Information System (http://rpms.dacnet.nic.in/) from Indian Ministry of Agriculture was queried to obtain weekly retail prices.

   Unlike daily prices, the only available data format that was capable of being parsed, was Microsoft Excel .xls format. So all the required .xls files were downloaded using a Python script, and then 'xls2csv' was used to convert .xls documents to CSV data format. For some reason, the majority of product name were missing in downloaded .xls files, so the basic heuristic, which involved the order of product names, was used to reconstruct the data. The data obtained described information about prices for a large number of markets, around 60 products and hundreds of subproducts, dating back from 2005, but, unfortunately, the data was far from complete, especially for 2005-2007 time frame.

Price sequences

Preprocessing

load data from csv into pandas dataframe
    removing NAs

Other sources

distribution & production
    exchange rate crude oil

Social Media data

Twitter

Historical tweets

Approach 1: Fetching "historical" tweets through Twitter API

Using the Twython package for python we are able to interface with the Twitter API. Our methodology (figure 0.1) is to select the twitter accounts of a number of regional celebrities as starting points. These are likely to 'followed' by large numbers of local users. In a first phase (TWEET_COLLECTION.py.get_followers()),

from each of these sources we may extract a list of followers and filter by various characteristics. Once a substantial list has been constructed it must be merged (merge.py and remove_intersection.py), we may proceed to download the tweet activity (up to the 3200 most recent tweets) of each of these users in a second phase (TWEET_COLLECTION.py.get_tweets()).

Despite recent updates allowing developers greater access, Twitter still imposes troublesome constraints on the number of requests per unit time window (15 minutes) and, consequently, the data collection rate. It is therefore necessary to: 1) optimise the use of each request; and 2) parallelise the data collection effort.

As far as optimisation is concerned, the **GET statuses/user_timeline** call may be called 300 times per 15 minute time window with up to 200 tweets returned per request. This sets a hard upper bound of 60000 tweets per time window. This is why the filtering stage of the first phase is so crucial. Using the **GET followers/list** call (30 calls/time window), we may discard in advance the majority of twitter users with low numbers of tweets (often zero), so as to avoid burning the limited user timeline requests on fruitless users, thus increasing the data collection rate. With this approach we may approach optimality and achieve 4-5 million tweets daily per process. However, it may be prudent to strike a balance between tweets per day and tweets per user. Therefore a nominal filter is currently set to 50 tweets minimum rather than 200. It is furthermore necessary to install dynamic time-tracking mechanisms within the source code so as to monitor the request rates and to impose a process 'sleep' when required.

Parallelisation begins with obtaining N ($\approx$ 10) sets of developer credentials from Twitter (https://dev.twitter.com/). These N credentials may then be used to launch N processes (get_users.sh) collecting user data in parallel. Given the decision to divide the follower collection and tweet collection into separate phases (this may alternatively be done simultaneously), there is no need for distributed interaction between the processes to control overlap, as each process will simply take $1/N$ th of the follower list produced in phase 1 and process it accordingly. It should be relatively simple to initiate this parallel computation given the design of the scripts.

A benchmarking test (table 0.1) performed in order to support configuration choices for the parallelisation. The test involved collecting the tweets from all good users within the first 20000 followers of @KareenaOnline, the account of a local celebrity. The following observations can be made:

- only 1.5-2% of users are considered "good" under the current choice of filters (location, min. 50 tweets etc.);
- Despite different levels of sleeping, phase 2 reads from users at roughly the same rate that phase 1 collects them (approximately 100 per time window

Phase 1				
Users	Duration (s)	Sleep (s)	User Rate	Type
334	2795	2047	-	Total
299	2700	2047	99.7	Normalised (3 windows)
Phase 2				
Tweets (Users)	Duration (s)	Sleep (s)	Tweet Rate	Type
171990 (334)	3108	922	-	Total
150008 (309)	2700	922	50002.7	Normalised (3 windows)

Table 0.1: A picture of the same gull looking the other way!

in both cases);

- Phase 2 produces around 50000 tweets per time window.

It is important to note however, that the rate of "good" users increases varies depending on the notoriety of the source account outside of India. To ensure good coverage for user collection, a wide variety of source users was chosen including rival politicians, musicians, sportspersons, film stars, journalists and entrepreneurs.

Tweet collection for Humanitas occurred in two main waves. In the first wave 180 000 users identifiers were collected. This amounted to 110 million tweets, collected over about three days, totalling 288GB of information (note a tweet response comprises the textual content as well as a substantial amount of meta data). In second wave of collection we encountered the effect of diminishing returns as many of the newly harvested users had already featured in the first wave. Despite a lengthier collection effort, only 110 000 new users were collected, leading to 70 million additional tweets and a grand total for the two waves of about 500GB of data. Future collection work for Humanitas would benefit from a more sophisticated approach, for example, by constructing a Twitter user graph.

Approach 2: Filtering tweets provided by webarchive.org

https://archive.org/details/twitterstream

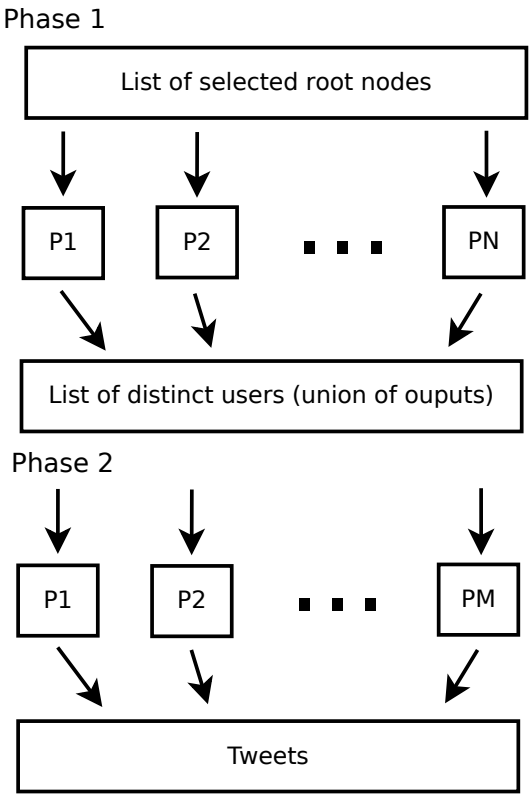


Figure 0.1: Tweet collection methodology.

Daily? tweet aggregator

Clustering according to keywords

Issue of localization

Geolocalized tweets

Filtering the available archives of tweets taken from the API yielded near to no geolocalized tweets from India matching our set of keywords. This reason is evident, because the twitter API only allows extraction of 1% of tweets and only 2% of tweets are actually geolocalized. In effect, getting tweets that match our keywords specific to food commodities is very unlikely. We had more luck with tweets from Indonesia, however as already explained we were unable to attain enough price sequences from Indonesia to actually train a model. Furthermore, the time constraints didn't allow us to get tweets from India and Indonesia in parallel in order to do some "stand-alone" clustering analysis.

Approximation: Mapping tweets to user location

Processing

Merging Series

Crafting indicators from tweets

Sentiment analysis

Sentiment analysis, or opinion mining, is the concept of using different computer science techniques (mainly machine learning and natural language processing) in order to extract sentiment information and subjective opinions from the data. In our context this may help us to find out how circumstances in relation to commodity prices affect the overall mood in the population.

From the start we decided that we did not want to build our own sentiment analysis system since the proper implementation, testing and evaluation would require a considerable effort compared with the total project workload. Instead, we are planning to use some of already developed solutions and tune them to our needs.

Several sentiment analysis frameworks were tested, including:

- SentiStrength
  (http://sentistrength.wlv.ac.uk/)

- Stanford CoreNLP
  (http://nlp.stanford.edu/sentiment/code.html)
- 'Pattern' library from the University of Antwerp
  (http://www.clips.ua.ac.be/pages/pattern-en)

All of these software packages produced reasonable results on some short and simple sentences, but sentiment grades looked almost random on a set of real collected tweets. Most likely, factors such as misspelled words, acronyms, usage of slang and irony contributed to the overall ambiguity of sentiment grades assignment.

Therefore, we decided to build and use our own simple system, which incorporated basic opinion mining, but was mainly focused on extracting relevant keywords, which could help to estimate tweets from the specific point of view. This approach, which also takes into account issues originating from word misspelling, is described in next two paragraphs.

Extracting predictor categories

First, several "predictor categories" were selected. These categories represent different aspects of price variation, and each category include several sets of words of different polarities. For example, the category "price" has two polarities: "high" and "low". The following word list belongs to "high" polarity: 'high', 'expensive', 'costly', 'pricey', 'overpriced', and these are words from "low" list: 'low', 'low-cost', 'cheap', 'low-budget', 'dirt-cheap', 'bargain'. Likewise, a category "supply" has "high" polarity (with words 'available', 'full', 'enough', 'sustain', 'access', 'convenient') and "low" ('run-out', 'empty', 'depleted', 'rotting'). The dictionary with total of 6 categories (each having at least two polarity word lists) was built ("predict", "price", "sentiment", "poverty", "needs", "supply"), let's call it D.

Then, for each tweet a feature vector is built, representing the amount of words from each category and polarity. Several cases have to be taken into account. First of all, a word may be not in its base form ("price" -> "prices", "increase" -> "increasing"), which will prevent an incoming word from matching one from D. Therefore, we use stemming technique to reduce each word to its stem (or root) form. Another problem is misspelled words ("increase" -> "incrase", "incraese"), and for tweets it happens more than usual due to widespread use of mobile devices with tiny keyboards. Our solution to this problem is covered in the next section.

Here is the overview of predictor category extraction algorithms we implemented:

Preprocessing: For each relevant word in D a stem is computed using the Lancaster stemming method, and the stem is added to reverse index RI, which maps a stem to a tuple: (category, polarity).

function get_category(w):
    Compute a stem s from w
    Check if s is present in RI.
    if *yes* **then**
        return the corresponding tuple.
    else
        ask spell checker for a suggestion
        is suggestion stem returned?
        if *yes* **then**
            return the corresponding tuple from RI
        else
            return None;
        end
    end

Tweets spell checking

People often do not pay much attention about the proper word spelling while communicating over the Internet and using social networks, but misspelled words may introduce mistakes in processing pipeline and significantly reduce the amount of filtered tweets, since the relevant, but incorrectly written word might not be recognized by the algorithm.

Several spell checking libraries were checked (Aspell and Enchant to name a few), but their 'suggest' method lacked both performance (several seconds to generate a suggestion for thousands words, which is very slow) and flexibility (it's not possible to specify the number of generated suggestions, as well as a threshold, such as maximal edit distance between words). Therefore, we decided to use simple approach which involved computing edit distances between a given word and words from predictor categories dictionary (D).

For each given word w we compute its stem s and then edit distance (also known as Levenshtein distance) to each word (stem) from D. It can be done really fast thanks to C extensions of *python-levenshtein* module.

After that, we choose the stem with minimal edit distance (using heap to store the correspondence between distances and words and to speed up selection of the minimal one), and check if the resulting number of "errors" (which is equal to distance) is excusable for the length of word w. For example, we don't allow errors for words of length 5 or less, only one error is allowed for lengths from 6 to 8,

etc. If everything is alright, then the suggestion is returned, otherwise the word is discarded.

The approach proved to be fast and tweakable, and was successfully used for tweets processing.

Price Transmission Analysis

Interpretation

automate interpretation to a certain extent by learning about circumstances through online data.

Time Series Analysis

Time series data has a natural temporal relation between different data points. It is important in the analysis to extract significant temporal statistics out of data. We will focus on analyze stationarity, autocorrelation, trend, volatility change, and seasonality of our price datasets in R.

Stationarity of a series guarantees that the mean and variance of the data do not change over time. This is crucial for a meaningful analysis, since if the data is not stationary, we can not be sure that anything we derive from the present will be consistent in the future. We can transform our data into a stationary one by taking k-th difference to remove the underlying trend, and then apply standard test procedures such as KPSS test [1] to see if the differenced series is stationary.

Autocorrelation is another important trait in time series data. It suggests the degree of correlation between different time periods. By plotting correlograms (autocorrelation plots) of our data, we will be able to identify if the fluctuation of prices may be due to white noise or other hidden structures.

Seasonality is reasonably expected in our agricultural related time series. Several methods might help us to detect seasonality, such as common run charts, seasonal subseries plots, periodograms, and the correeologrames we mentioned before.

(trend and volatility change is straightforward and can be concluded once we have the datasets)

[1] Kwiatkowski, D.; Phillips, P. C. B.; Schmidt, P.; Shin, Y. (1992). "Testing the null hypothesis of stationarity against the alternative of a unit root". Journal of Econometrics 54 (1âĂŞ3): 159âĂŞ178.

Prediction Models

DATA: regularly sampled time series: wholesale daily, retail daily and weekly

Time Series Forecasting

ARMA Model

The classical Time series forecasting approach is to use the ARMA (Auto-Regressive Moving Average) model to predict the target variable as a linear function which consists of the auto-regressive part (lag variables) and the moving average part (effects from recent random shocks).

The ARMA(p,q) model: (will refine math representations later)

$Phi(B) * Y_t = Theta(B) * eps_t$

The fitting of the model and the historical data can be accomplished by maximum likelihood estimation.

Regression

We can also apply ARMA to the linear regression model. It is formulated as such:

$Y = Beta * X + eps, eps\ ARMA(p, q)$

Through OLS (Ordinary Leasat Square) or GLS (General Least Square) processes, we can obtain an optimal Beta.

Multilayer Perceptrons

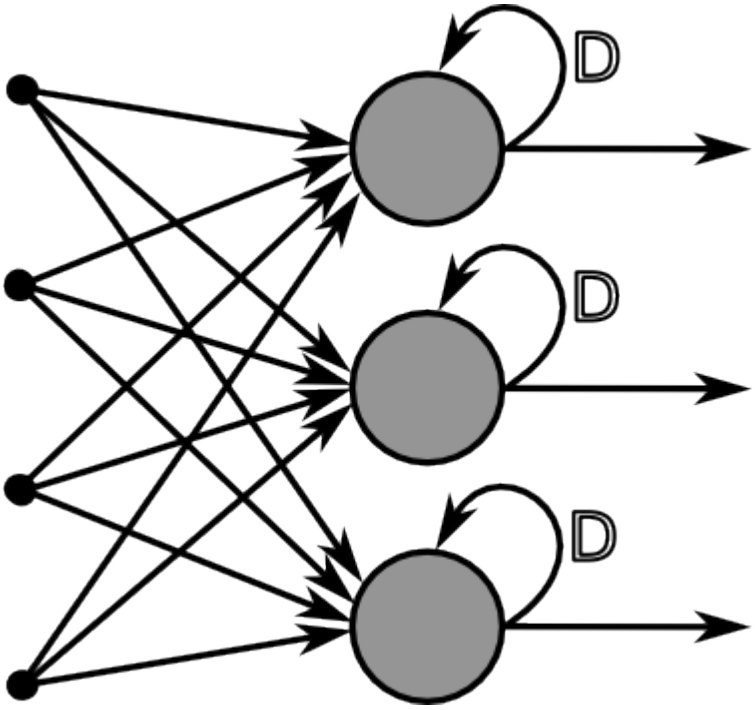
taken from M. Seegers course on Pattern Recognition and ML

Recurrent Neural Networks (RNN)

source: scholarpedia A recurrent neural network (RNN) is a neural network in with feedback connections, enabling signals to be fed back from a layer l in the network to a previous layer.

Simple Recurrent Networks

The simplest form of an RNN consists of an input, an output and one hidden layer as depicted in fig.[].



[source: wikipedia]

General description of a discrete time RNN

A discrete time RNN is a graph with K input units $\mathbf{u}$, N internal network units $\mathbf{x}$ and L output units $\mathbf{y}$. The activation (per layer) vectors at point n in time are denoted by $\mathbf{u}(n) = (u_1(n), ..., u_n(n))$, $\mathbf{x}(n) = (x_1(n), ..., x_n(n))$, $\mathbf{y}(n) = (y_1(n), ..., y_n(n))$. Edges between the units in these sets are represented by weights $\omega_{ij} \neq 0$ which are gathered in adjacency matrices. There are four types of matrices:

- $\mathbf{W}^{in}_{N\times K}$ contains inputs weigths for an internal unit in each row respectively
- $\mathbf{W}_{N\times N}$ contains the internal weights. This matrix is usually sparse with densities $5\% - 20\%$
- $\mathbf{W}^{out}_{L\times(K+N+L)}$ contains the weights for edges, which can stem from the input, the internal units and the outputs themselves, leading to the output units.
- $\mathbf{W}^{back}_{N\times L}$ contain weights for the edges that project back from the output units to the N internal units

In a *fully recurrent network* every unit receives input from all other units neurons and therefore input units can have direct impact on output units. Output units can further be interconnected.

Evaluation The calculation of the new state of the internal neurons in time-step $n+1$ is called evalution.

$$\mathbf{x}(n+1) = \mathbf{f}(\mathbf{W}^{in}\mathbf{u}(n+1) + \mathbf{W}\mathbf{x}(n) + \mathbf{W}^{back}\mathbf{y}(n))$$

where $f = (f_1, ..., f_N)$

Exploitation The output activations are then computed from the internal state of the network in the exploitation step.

$$\mathbf{y}(n+1) = f^{out}(\mathbf{W}^{out}(\mathbf{u}(n+1), \mathbf{x}(n+1), \mathbf{y}(n)))$$

where $f^{out} = (f_1^{out}, ..., f_L^{out})$ are the output activation functions and the matrix of output weights is multiplied by the concatenation of input, internal and previous output activation vectors.

RNNs can in theory approximate any dynamical system with chosen precision, however training them is very difficult in practice.

Echo State Networks

Echo State Networks (ESN) are a type of discrete time RNNs for which training is straightforward with linear regression methods. The temporal inputs to the network are transformed to a high-dimensional *echo state*, described by the neurons of a sparsely connected *random* hidden layer which is also called a reservoir. The output weights are the only weights in the network that can change and are trained in a way to match the desired output. ESNs and the related liquid state machines (LSMs) form the field of *reservoir computing*.

Echo State Property

The intuitive meaning of the *echo state property* (ESP) is that the internal state is **uniquely** determined by the history of the input signal and the teacher forced output, given that the network has been running long enough. Teacher forcing essentially means that the output $\mathbf{y}(n-1)$ is forced to be equal to the next time series value $\mathbf{u}(n)$ and thus to the next input.

Definition 1 *For every left infinite sequence* $(\mathbf{u}(n), \mathbf{y}(n-1)), n = \ldots, -2, -1, 0$ *and all state sequences* $\mathbf{x}(n), \mathbf{x}'(n)$ *which are generated according*

to

$$\mathbf{x}(n+1) = \mathbf{f}(\mathbf{W}^{in}\mathbf{u}(n+1) + \mathbf{W}\mathbf{x}(n) + \mathbf{W}^{back}\mathbf{y}(n))$$
$$\mathbf{x}'(n+1) = \mathbf{f}(\mathbf{W}^{in}\mathbf{u}(n+1) + \mathbf{W}\mathbf{x}'(n) + \mathbf{W}^{back}\mathbf{y}(n))$$

it holds true that $\mathbf{x}(n) = \mathbf{x}'(n)$ *for all* $n \leq 0$.

The echo state propery is ensured through the matrix of internal weights W

Theorem 1 *Define* σ_{max} *as largest singular value of* $\mathbf{W}$*,* λ_{max} *as largest absolute eigenvalue of* $\mathbf{W}$*.*

1. *If* $\sigma_{max} < 1$ *then the ESP holds for the network*
2. *If* $\|\lambda_{max}\| > 1$ *then the network has no echo states for any input/output interval which contains the zero input/output tuple (0,0)*

In practice is suffices to make sure the negation of the second point holds.

Training the ESN

"The state of the ESN is therefore a function of the finite history of the inputs presented to the network. Now, in order to predict the output from the states of the oscillators the only thing that has to be learned is how to couple the outputs to the oscillators, i.e. the hidden to output connections:" http://stackoverflow.com/questions/21940860/e state-network-learning-mackey-glass-function-but-how

Hyperparameters: dimensionality of W, spectral radius α

Initial state determination The network is run for a first set of inputs and the results are then discarded. If the spectral radius is close to unity, implying slow forgetting of the starting state, the initial set has to be a substantial part of the training dataset.

"Likewise, when the trained network is used to predict a sequence, a long initial run of the sequence must be fed into the network before the actual prediction can start. This is a nuisance because such long initial transients are in principle unneccessary"

"By contrast, a recurrent neural network such as our echo state network, but also such as the networks used in [5] need long initial runs to âĂIJtune inâĂİ to the to-be-predicted sequence. [5] tackle this problem by training a second, auxiliary âĂIJinitiatorâĂİ network." EchoStatesTechRep.pdf, p.32

Teacher forcing

Feedback

Batch learning

Ridge Regression

Leaky integrator neurons *Taken from Echo State Tech Rep. page 26/27*
In order for the Echo State Network to be able to learn slowly and continuously changing dynamics and thereby to capture longterm phenoma in the price sequences we feed in, we need a way to introduce continuous dynamics. This is done via approximation of the differential equation of a continuous-time leaky integrator network

$$\frac{d\mathbf{x}}{d\mathbf{t}} = C(-\alpha\mathbf{x} + \mathbf{f}(\mathbf{W}^{in}\mathbf{u}(n+1) + \mathbf{W}\mathbf{x}(n) + \mathbf{W}^{back}\mathbf{y}(n))) \tag{1}$$

where C is a time constant and α the leaking decay rate. For the approximation we introduce a stepsize δ:

$$\mathbf{x}(n+1) = (1 - \delta C\alpha) + \delta C(\mathbf{f}(\mathbf{W}^{in}\mathbf{u}(n+1) + \mathbf{W}\mathbf{x}(n) + \mathbf{W}^{back}\mathbf{y}(n)))$$

Theorem 2 *Let a network be updated according to*

Online learning with Recursive Least Squares (RLS) update weights in each iteration ...

Algorithm 1 *1.*

2.

Parameter selection with Maximum Entropy Bootstrap (Meboot) In order to find the best parameters for generalization during training of the neural network models with we create replicate time series of a selected price sequence dataset. The method we employ to this end is called 'Maximum Entropy Bootstrap' (meboot) and was introduced by H.D. Vinod in 2006. [reference]. The reason for the use of this specific method is that, due to temporal dependence, time series cannot simply be randomly sampled into a new dataset. The meboot algorithm allows for construction of random replicates of the given time series showing the same statistical properties.

RNN with backpropagation decorrelation algorithm

Data Mining

In this section we give a brief introduction to association rule mining and detail the implementation and evaluation of the system. In order to run the code you will need to install the libraries Scipy and Orange

A brief introduction to Association rule mining

The objective of association mining is the elicitation of useful rules from which new knowledge can be derived. Association mining applications have been applied to many different domains including market basket analysis, risk analysis in commercial environments, clinical medicine and crime prevention. They are all areas in which the relationship between objects can provide useful knowledge.

Itemsets are identified by the use of two metrics support and confidence.

Support is a measure of the statistical significance of the rule. Rules with a very low support are more likely to occur by chance. With respect to the market basket analysis items that are seldom bought together by customers are not profitable to promote together. For this reason support is often used as a filter to eliminate uninteresting rules.

Confidence on the other hand is is a measure on how reliable the inference made by a rule is. For a given rule $A \implies B$, the higher the confidence, the more likely it is for the item set B to be present in the transactions that contain A. IN a sense confidence provides an estimate of the conditional probability for B given A.

It is worth noting that the inference made by an association rule does not necessarily imply causality. Instead the implication indicates a strong concurrence relationship between items in the antecedent and consequent of the rule.

So how do we go about implementing this in an algorithm? A rather naive approach would be to check if each itemsset satisfies minimum support. However

this is rather inefficient and unnecessary. We can make use of of the observation that every subset of a frequent item set also has to be a frequent item set. This works the otherway as well. If a set is not frequent then its superset can not be frequent either. This observation allows us to prevent unnecessary computation. The Apriori Algorithm uses this downward closure to identify frequent item sets. Candidates that do not satisfy minimal support are pruned, which automatically reduces the algorithm's search space. Once frequent item sets have been generated the algorithm enters the second face, namely generating derivations for which the metric minimal confidence is used.

Implementation

The implementation is guided through three stage namely data crunching, classification and mining where the later is a straight forward implementation of the mining library orange.

In order to run the Apriori Algorithm we needed to do some preprocessing since the data available was in a uncompatible format. The data readily available to us was in the form of Date Country City Category Commodity 1, Date Country City Category Commodity 2. In the statistics community this kind of format is referred to as the "long format". In order to run Apriori we need to convert the table to a "wide format" meaning that all commodities need to be related to one index. You can imagine a matrix where the y axis is described by city + date and the x axis is labeled with all the available commodities. To do the conversion we used Stata, a statistical software which is mostly used in the field of social science. Before reading the data into state we filtered the document for special characters ("", /) and replaced all unavailable price informations with "NA". For this purpose we used the filter.py script. Once we read in the data into state we removed duplicates and started the conversion. To reproduce the correct table format you fan follow our implementation in the stata script. The data is now in the correct table format what remains to be done is slicing the table according to a city and sorting the data in order of decreasing time stamps. These steps are performed in the dataFrame.py. We now continue to process the files by classifying continues variables into categories. We therefor filtered the maximum and minimum increase/decrease in price to establish a range of values. Depending on whether a price increase fall into the first, second or third third of the price range we classified it as small, medium or big increase/decrease respectively. The categorial data can now be processed by the assoc.py file. The library orange provides an implementation of the a priori gen algorithm. We simply set the min support value and write the 10 rules with the highest support count to a file.

Results

What we were hoping to find were seasonal related price changes such as those we experience when shopping for fruits and vengtables at our local grocery stores. Rules such as $Tomatoes = bigincrease \implies potatoes = smalldecrease$ could serve

Our initial granularity was set to weekly data. From our meta analysis we observed that prices quite frequently stay unchanged over the period of several weeks. This resulted in an overwhelming amount of rules in the form of commodity $A = unchanged \implies B = unchanged$. This made it really hard to filter the set for insightful rules. We therefor decided to compare price changes over the period of 12 weeks. Although the majority of the rules still remained in the above form we managed to extract some relations related to price changes.

We conclude that the strongest correlations exist between commodities with unchanged prices. This observation underlines the nature of food commodities. Agricultural commodities are known to be less volatile then for example energy commodities. We further noticed that depending on the region we would have a totally different rule count. Capital cities tend to have a bigger rule base meaning that there exist a higher correlation between products. Two extreme example are Dassau, which resulted in no patterns at all and Shillong which produced over 260000 rules. In addition our experiments showed the finer the granularity of the data the higher the support and correlation between products. This is only natural given that prices can stay stable over the period of weeks or even months.

Trading advice

As concluded above our algorithm found many associations between products. In order to interpret most of the results specialized domain knowledge in trading with commodities is necessary. Some more obvious rules we managed to interpret were the following:

$BreadLocal = bigincreaseMilkCowBuffalo = unchangedincrease \implies MaidaNA = unchangedincrease$. If prices of bread inflate it is save to trade Maida, as its price will most certainly stay stable.
$BiscuitGlucose = unchangedincrease \implies BesanNA = smallincrease$, similarly means that stable pries in biscuits will imply a small increase in Besan.