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

Introduction

Motivation

Many countries in the global south are affected by high and especially volatile prices of staple foods and other commodities. These circumstances most heavily impact low-income consumers, small producers and food traders. These people in general have no direct access to real-time price information, let alone price predictions, that would allow them to plan ahead and thereby at least mitigate the impacts of price volatility. Furthermore many governments donâĂŹt have sophisticated models to coordinate their interventions on the commodity markets and optimally distribute their commodities.

Project goal

The main idea of this project was to find indicators present in social media that help in monitoring and predicting prices of basic commodities. In this context we wanted to conceive a commodity price and supply prediction framework for developing countries tough combining commodity price data from official sources (governments, international organizations such as the World Bank and the IMF) and indicators derived from social media data. We set out piloting this approach for a specific country with freely available price data, India, and hoped to achieve the prediction of commodity prices on a daily basis at state level. Due to the limited amount of penetration of twitter in India and the nature of discussions on twitter we expected filtering out a significant amount of tweets relevant to our needs a hard task. For further details we refer to our project proposal.

Time Series data

'In consequence we will start building the system using data made available by the Ministry of Agriculture of the government of India. It provides monthly and partly even weekly price recordings dating back to 2002 through the website.'

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

We used Pandas, a powerful and fast Python data analysis library, to load the csv files of datasets, to clean up, and to organize them into usable formats. The prices of all products in these datasets are collected through a human reporting mechanism from local institutions to a central ministry. Due to the nature of this data collecting process, the qualities of these 3 datasets suffer from human neglects and mistakes. We thus need to conduct 2 phases of work prior to time series analysis and price prediction: data clean-up and dataset usability analysis.

Price Data Clean-up

In data clean-up, we fixed several defects in the original datasets of the data sources. Each series originally has different portion of missing dates, and some have multiple data points of the same date with different values. The first stage of clean-up was to make each series align to a unique sequence of dates with constant date frequency by inserting NaNs to the missing and duplicated dates.

Next, we discovered that there are many outliers with extreme values in the series. For example, a price of 10 rupee today jumps to 100 tomorrow, and then goes back to 10 the day after. Such cases are more likely caused by human mistakes in reporting prices to the Indian ministries, so we used a heuristic to remove these suspicious spikes in the series. By taking the daily differences in percentage of a series, we were able to remove data points of the dates which perform more than 100

Finally, we patched the missing parts of the series with common methods such as linear or cubic spline interpolation.

Price Dataset Usability Analysis

The dataset usability analysis phase was intended to explore the data to find usable materials for analysis and prediction in later stages. We first filtered out all the series with less than 60

The data availably in the daily wholesale dataset varies tremendously among different products and regions. Although there are about 15 regions, 7 candidate products, and more than 10 sub-categories for each product, most of the combinations of region, product, sub-category do not have enough available data for analysis and prediction. Fortunately, we did find out a few very good series which have more than 80

On the other hand, the daily retail dataset appears to have consistent data availability among products and regions, but in fact all products have only about 60

Considering the limitations of the current price datasets, we have concluded the following potential usage of them: (1) Select a few representative series of each product with very good data quality. By using these individual series as a starting point of analysis and prediction, we can find out general characteristics and also the feasibility of predicting the prices of these highly volatile commodities. (2) Merge the series of the same product in each region to construct an extended dataset containing a uniform profile for each product in each region. In this way, we can gain a national overview of trends and price variations of different products.

Other sources

distribution & production

exchange rate crude oil

Additional types of data were "scraped" from two online sources using HTML parsing library, 'BeautifulSoup' for Python. The first of these sources was meteorological web site Tu Tiempo (http://www.tutiempo.net). Daily climate data (comprising temperature, humidity, perspiration etc.) for the last 16 years (1999-2014) was collected for over 100 locations in India. The script for this is get_climate_data.py. The second source was monthly inflation (CPI) data for India for the past 16 years from inflation.eu. The script for this purpose is get_inflation_data.py.

Social Media data

Twitter is a rich resource for studying cutting-edge trends on a global scale. Given a sufficiently large data collection effort, Twitter user discourse indicating changes in commodity prices may be obtained. This discourse supplies us with predictors

The Humanitas project harvests huge amounts of user activity from this social media platform in order to capture the sparsely distributed activity pertaining to food prices. It is this aspect of the project which promotes it to the domain of 'big data'.

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

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: Trial run results

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 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

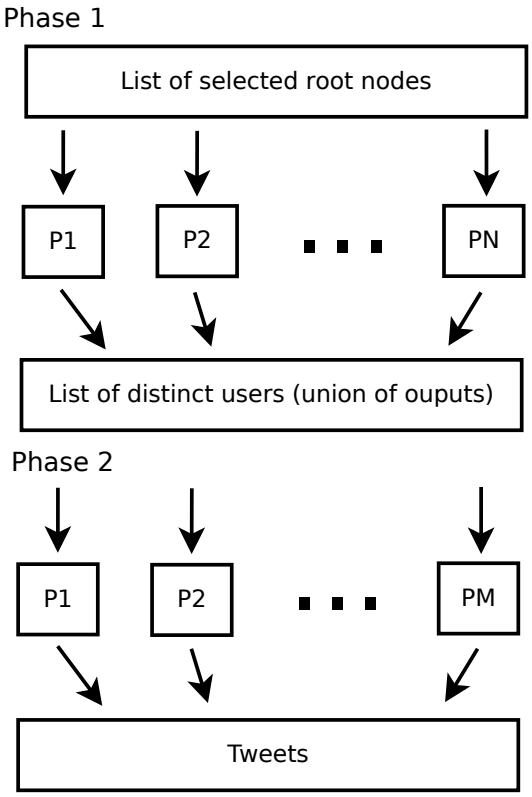


Figure 0.1: Tweet collection methodology.

500GB of data. Future collection work for Humanitas would benefit from a more sophisticated approach to collecting users (phase 1), for example, by constructing a Twitter user graph.

Approach 2: Filtering tweets provided by webarchive.org

Since previous approaches failed because of Twitters download limitations, we decided to collect tweets also from the freely available tweet archives from *https://archive.org/details*
The collection was done via a python script which was executed on all 8 azure nodes

in parallel. Beforehand the respective archives were downloaded to the nodes with a download speed of approx. 20 MB/s. The storage was to our surprise no problem, because every azure node gets 133 GB of temporal storage to for disposal. About 550 GB of compressed tweets were processed and filtered in about 36 hours per node.

The applied filter removed tweets not containing at least one food word (e.g. rice, curry, food, ...) and one predictive word (e.g. increase, decrease, price, ...). Also retweets were filtered out, because sophisticated duplicate detection across 8 nodes can be costly and some exploration of the data showed that almost all duplicates are filtered out through this check. Since we want to predict food prices for certain regions, the location of the tweets is very important. We came up with a simple scheme to detect locations:

- **Geo-location:** Some tweet objects contain a field 'coords' which indicates n the exact location the tweet was sent from.
- **Mentioned regions:** In the associated tweet text regions can be mentioned, which can also give clues on the regions affected.
- **Places:** When submitting a tweet an user can also specify a place associated with the tweet. This information can be extracted from the tweet objects.
- **User location:** Most tweets objects also have an associated user object, which contains a user location sometimes. The textual information of the filed is tokenized and compared with a list of known regions.

According the categories mentioned above, the tweets were split up in several files. Since the *Approach 1.* conveyed much more tweets, the yielded sample was only used for experimental data exploration and testing of the more refined tweet processing.

Daily? tweet aggregator

Our first idea was to build up a continuously running process, which fetches the newest data from the twitter stream from india. But after applying a simple filter, we came to the conclusion that the data is to sparse for this approach.

Clustering according to keywords

Since the relevant data was really sparse, we didn't expect much gain from any unsupervised learning techniques and decided to omit clustering.

Issue of storage

At first we believed that there wouldn't be enough space to store all tweets, but after setting up an azure node, we found that there is about 133 GB of temporal storage associated with it. We used this space to store the huge amount of tweets, but since the storage is temporal we lost tweets a couple of times. Azure 'heals' (restarts) nodes, when it detects anomalies, internal errors or just moves images around, which results in memory loss. We lost big tweet collections several times because of that. All filtered tweets were later stored on the main disk to avoid further loss.

Issue of localization

It's is very important for use to detect the location of tweets, because we want to predict the volatile food prices at regional granularity.

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

To get as many tweets as possible associated with a location we decided to use the locations of user accounts as a simple heuristic. We created a mapping between city and region names and used to to identify valid locations, which were then used during later processing.

Processing

After all data was collected we had to process the data for the neural networks (prediction) and the web visualisation.

Merging Series

Crafting indicators from tweets

To use the collected tweets for prediction in the neural network, aggregated indicators need to be extracted from the collection. The result of this processing is then stored in a csv file which is supplied to the neural network.

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 natural language processing and opinion mining techniques, 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
```

On a high level, every tweet is split into words, and then each word (token) is passed through 'get_category' function. But here's another problem we face using this approach: each relevant word we met may have a negation word (particle) before it, which subverts the meaning: "increases" -> "doesn't increase", "have food" -> "have no food", etc. To deal with this problem, we employed the following method: we added a special 'negation' category with a list of negation words ("not", "haven't", "won't", etc.), and if there is a word with "negative" category before some relative word (to be more precise, within some constant distance from it, say 2), then we change the polarity of relative word's category. For example, if a word is from category "poverty" and has "high" polarity (like "starving"), then negative category word right before it (such as "aren't") will turn the polarity to "low".

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.

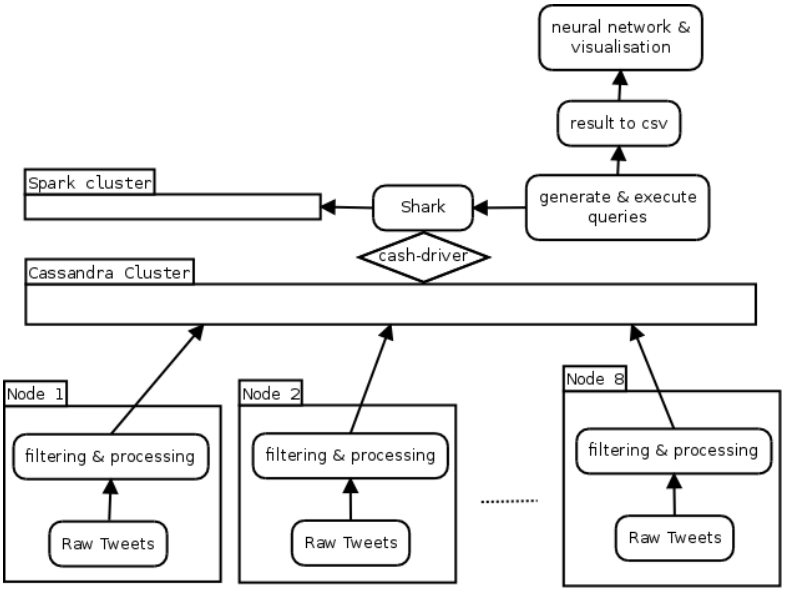


Figure 0.2: Tweet processing pipeline.

Infrastructure

Almost all downloading and processing of tweets was done in parallel on 8 Azure nodes. A script was written to upload public keys to the nodes for seamless access. The same script was used to execute various task on all or specific nodes. To handle the huge amounts of data and do efficient OLAP, we decided to use spark/shark. The data was first stored on each node separately on disk, then processed and afterwards inserted into a Cassandra DB cluster running on the nodes. The Apache Cassandra project is open source implementation of a NoSQL, column-oriented data base. It's known for being fault tolerant, which was very useful during various restarts, and processing many inserts fast. Since we had very limited memory on the non-temporal disk, we experienced 'Out of memory' errors, but because all data has been uploaded to the whole cluster, the scripts could continue running smoothly although some nodes were down. We also experienced node faults whenever the Azure node management 'healed' nodes.

Setting up the Cassandra cluster across multiple Azure subscriptions was tiring, since all used ports had to be opened on all machines manually via the Azure management web interface. Opening ports was regretfully not enough to get a fully functioning Spark cluster running, because Spark uses random ports for communication between running jobs. We tried setting up a VPN, but decided to quit after several hours due to time constraints and it not being essential for the result that the Spark cluster is made up of all nodes. Luckily Shark on top of Spark with only one node was still fully functional and executed queries at rea-

sonable speed. To connect Shark with Cassandra we used the Hive support for Cassandra driver 'cash' by tuplejump. We experienced several problems (some undocumented) while installing (e.g. libraries missing, wrong paths, ...), but still managed to get it running. To improve speed between the nodes all VMs were created in the same region.

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

We decided to evaluate different prediction models on the processed price time series. Since the processing of the price data made available by the Indian government yielded very few usable time series we constrained ourselves to try the different prediction model on a set of *daily* wholesale price series that had over 90% support and could be sufficenlty cleaned and interpolated.

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 Least Squares) or GLS (General Least Squares) processes, we can obtain an optimal Beta.

Feed Forward Neural Networks - Multilayer Perceptrons

taken from M. Seeger's course on Pattern Recognition and ML

Recurrent Neural Networks (RNN)

A recurrent neural network (RNN) is a neural network 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.[].

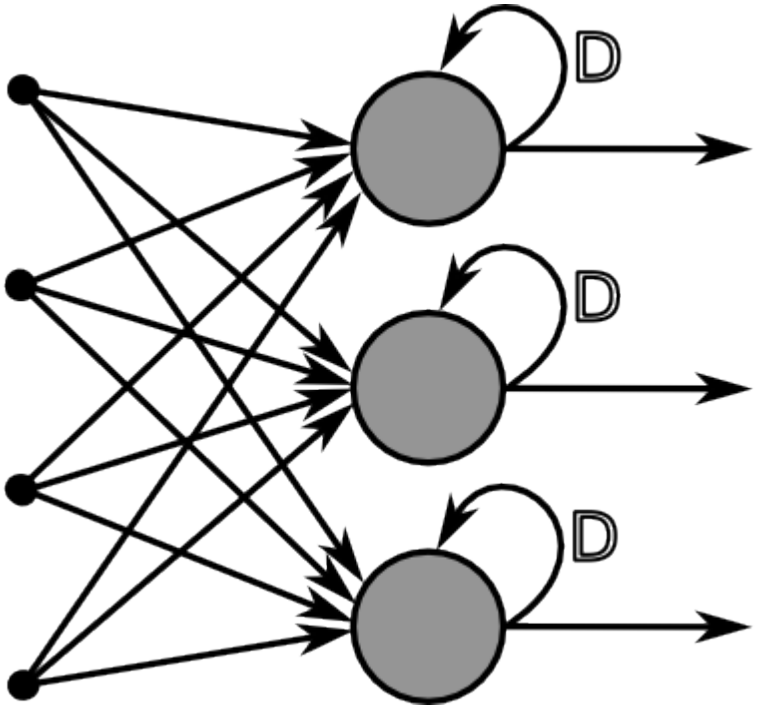


Figure 0.3: source: wikipedia

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 weights 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 evaluation.

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

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

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

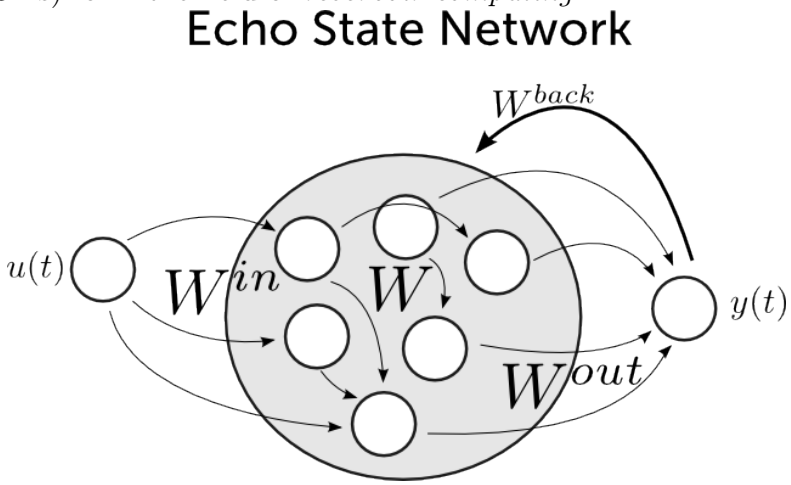
$$\mathbf{y}(n+1) = f^{out}(\mathbf{W}^{out}(\mathbf{u}(n+1), \mathbf{x}(n+1), \mathbf{y}(n)))$$

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. In the following section we are going to describe our use of an RNN that exhibits hexactly these properties yet is easy to train.

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 property 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/echo-state-network-learning-mackey-glass-function-but-how

Hyperparameters: dimensionality of W, spectral radius α

Initial state determination Prior to training the Echo State Network has to be run for substantive set of inputs with discarding of the results. A spectral radius close to unity implies slow forgetting of the starting state and therefore a substantial part of the training set has to be invested. EchoStatesTechRep.pdf, p.32

Batch learning with Ridge Regression The easiest way to train an Echo State Network is through minimizing the residual of an overdetermined linear set of equations. After the reservoir initialization phase, at init_index of the time series data, the internal states are saved together with the network inputs at every time step as rows to a matrix X. This results in a linear set of equations $XW^{out} = Y$, where $Y \in \mathbb{R}^N$ is the vector of outputs starting at init_index+1. Since the system is overdetermined with no unique solution for the weights W^{out} that perfectly fits all equations one applies Thikonov regularization instead of ordinary least squares. Consequently we minimize the residual of the SLE with an added regularization term:

$$\|XW^{out} - Y\|^2 + \|\nu W^{out}\|^2 \tag{1}$$

The thikonov regularizer allows finding a stable solution that will not adversely affect the network output by improving the conditioning of the problem. The corresponding normal equations become:

$$W^{out} = (X^T X + \nu I)^{-1} X^T Y \tag{2}$$

Teacher forcing Teacher forcing is the simplest training method in which the respective value of the target time series $y_{target}(n) = u(n+1)$ is fed in at every consecutive time step with input weights W^{in}.

Feedbacks During training with feedbacks the output $y(n)$ produced by the network is additionally fed back into the reservoir with feedback weights W^{back}. There further exists the possibly to decouple the network from the actual input during *online* training which we will discuss later.

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 long-term phenomena in the price series 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

$$\frac{d\mathbf{x}}{d\mathbf{t}} = C(-\alpha\mathbf{x} + \mathbf{f}(\mathbf{W}^{in}\mathbf{u}(n+1) + \mathbf{W}\mathbf{x}(n) + \mathbf{W}^{back}\mathbf{y}(n))) \tag{3}$$

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

$$\mathbf{x}(n+1) = (1 - \delta C\alpha) + \delta C(\mathbf{f}(\mathbf{W}^{in}\mathbf{u}(n+1) + \mathbf{W}\mathbf{x}(n) + \mathbf{W}^{back}\mathbf{y}(n))) \quad (4)$$

In the implementation we used a simple weighted average of consecutive reservoir states $(1 - \alpha)x(n-1) + \alpha x(n)$.

Theorem 2 *Let a network be updated according to*

Online learning In contrast to learning the weights once in batch-mode after the training run is completed one can also train the output weights in an online fashion after each consecutive presentation of an input. This method allows for a better adaption of the weights to local structures present in the data. There exist several algorithms for online training of the Echo State Networks and related reservoir networks. In the following we will shortly discuss two of the basic algorithms which are mentioned most often mentioned in academic literature.

- **Least Mean Squares (LMS):** LMS algorithms try to find the optimal weights (filtered coefficients) that minimize the mean squared error for an adaptive filter. Essentially LMS are stochastic gradient descent methods that perform updates according to the error present at time n. LMS algorithms are a class of adaptive filter used Very simple: Least Mean Squares, but bad performance update weights in each iteration ...

Algorithm 1 *1.*

2.

Recursive Least Squares unstable during training Therefore use (inverse) QR RLS, for motivation visit this link: http://zone.ni.com/reference/en-XX/help/372357A-01/lvaftconcepts/aft_rls_algorithms/

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.

Demonstration

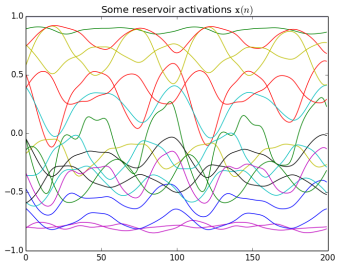
In this section we demonstrate the predictive power of the simplest ESN configuration. Ridge Regression is used to train on and predict a time series generated using the Mackey-Glass differential equation

$$http://www.scholarpedia.org/article/Mackey-Glass_equation$$

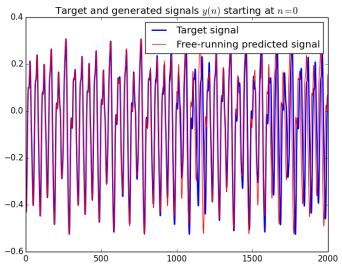
and noise.

Obtained results

The mackey glass time series is generated by a function with added noise. Since the development of commodity prices doesn't follow a similarly easily describable pattern we did not expect to obtain comparable results. Once trained on several years of price series data the ESN is capable of predicting prices very accurately on a day-to-day prices without any retraining needed (show 35/70/140 days). However, due to the limited usefulness of day-to-day predictions, we would like to predict prices farther into the future with relative accuracy. So far we have only achieved predicting the general trend for 6 to 7 days and we are not sure if greater accuracy is achievable with these models. Our experiments with inputing additional data like temperature, precipipitation, consumer price index (inflation) and twitter data show NO? significant improvements. In their current state, due to very limited amounts of available data, the twitter indicators cannot prove their use. Significantly bolstered however they would offer a wide range of possibilities. Matching these indicators to available price data series could show up relationships between conversations and the actual price. These relationships could be used to track price changes in real-time on a regional level through the conversations on twitter and thereby prove an interesting alternative to the inadequate/flawed data monitoring of governments. Improving predictions if possible would certainly acquire a lot more focused research in the area recurrent neural networks for time series prediction. The Echo State Network was implemented from scratch in scientific python.



(a) Sample of resulting activations after running the training series through the ESN



(b) Near perfect prediction 1000 points Mackey Glass testset by ESN

Figure 0.4: Results of simple ESN trained with Ridge Regression on a Mackey Glass time series consisting of 2000 points; parameters: spectral radius 1.25, leaking rate 0.3.

Further research

Implement stable version of RLS - Inverse QR RLS (book), implement Backpropagation Decorrelation algorithm (doesn't require specific set up of the network - spectral radius, initialization run etc., try to obtain better results for further research (make the network responsive to feeding back its own outputs), contrast ESN to its 'rival' the Long Short Term Memory Network trained with the Evolino method

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 itemset 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 other way 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 an incompatible 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 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 vegetables 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.

Visualization of results

We essentially represent the results of data collection and prediction on the Indian map.

Geo-visualization

Technology

LeafletJS: A javascript framework that builds the world map on top of OpenStreetMap library. MapboxJS is also used to extend the functionality of different behaviors on the map as well as enhance the UI design.

Content

There are several layers shown on the map:

- Tweets density of users in different Indian states.
- Price comparison between states

Atomic visualization

Technology

HighchartsJS

Tweets per state

Price fluctuation

Conclusion

Twitter statistics however very much favor Indonesia with 11.7% of the population being users while in India the percentage is only 1.3%. Were we to continue with the project it would make sense to apply the methods we conceived to the newly available national price table published by the Trade Ministry of Indonesia.

An interesting additional feature would be to derive food supply indicators from social media and to figure out if it is possible to detect underlying patterns in food supply in a country that are actually caused by policy decisions or inaction. This could aid the respective government in allocating resources in their substantial intervention in the food market.

Closing remarks