Humanitas

A monitoring & prediction toolset for volatile commodity prices in developing countries

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

Teaching Assistant: *Aleksandar Vitorovic*

Professor: *Dr. Christoph Koch*

a project for the "Big Data" course 2014

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Introduction *Fabian Brix*

Motivation

Many countries in the global south are affected by high and especially volatile prices of staple foods and other commodities. This circumstance most heavily impacts 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 don't have sophisticated models to coordinate their interventions on the commodity markets and optimally distribute their commodity stocks.

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 through 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 penetration of twitter in India and the nature of content on twitter we expected the filtering of a significant amount of relevent tweets to be a hard task. To obtain furhter information on our initial plans please consult our project proposal.

Time Series data *Fabian Brix, Anton Ovchinnikov*

The basis for the prediction part of this project are time series of price data. We acquired these for different stages of pricing in the supply chain and at different time granularities from different Indian government ministries. The data had to be downloaded and processed into usable formats from the websites of these ministries. The different sources consist of the Ministry of Agriculture and the Ministry of Trade and their various departments. In terms of different types of prices we differentiate between wholesale market prices and retail prices. We subsequently structure our documentation of data sources and the data processing according to these two types.

Wholesale prices

Wholesale price index

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.

Data sources

We came across a website maintained by the Indian Ministry of Agriculture that tracked *daily* wholesale prices for units of 100 kilograms for a huge number of market places across the country. The data is recorded back to 2005 and is supplemented by stock arrival numbers.

Due to large amount of data available, as well as time and resources needed to pull it, we constrained ourselves to choosing several major products (Onion, Rice, Wheat, Apple, Banana, Coriander, Potato, Paddy(Dhan), Tomato), where a 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. 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

   The Retail Price Information System of the 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 names were missing in the downloaded .xls files, so a 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 to 2005, but, unfortunately, the data was far from complete, especially for 2005-2007 time frame.

Price sequences *Ching-Chia Wang*

We used Pandas, a powerful and fast Python data analysis library, to load the csv files of datasets, to clean them 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

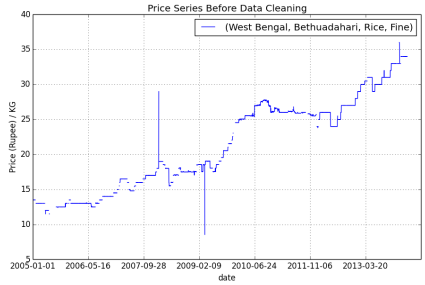
In data clean-up, we fixed several defects in the original datasets of the data sources. Each series originally has different portion of missing dates, and some have multiple data points of the same date with different values. The first stage of clean-up was to make each series align to a unique sequence 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% price change in one time step (daily or weekly).

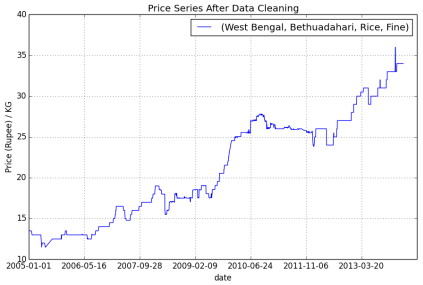
Finally, we patched the missing parts of the series with common methods such as linear or cubic spline interpolation.

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% of non-missing data, then computed some indicators of



(a) Sample price series after date alignment during data cleaning



(b) Sample price series after spikes-removing and interpolation

Figure 0.1: Regression with ARMA error shock

the data availability of each product in each region. Unfortunately, we have found out that all the datasets are far from satisfying.

The data availability 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 after filtering, 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% 90% of valid data and also perform the desired traits of high volatility and rising trend across time. These become good candidates for later analysis and prediction.

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% of valid data in each region. Moreover, after examining each series via plots, we discovered that most of the daily retail series have weird behavior. For instance, the price of a series may have identical values for a long period and suddenly jump to another value, which makes the series look like levels of ladders. These series are not useful since such pattern is probably caused by dysfunctional price reporting process. Last, the weekly retail datatset has too many incorrect data points due to the heuristic of parsing excel files mentioned in the previous section. We thus concluded that the weekly retail dataset can be discarded with no loss.

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,

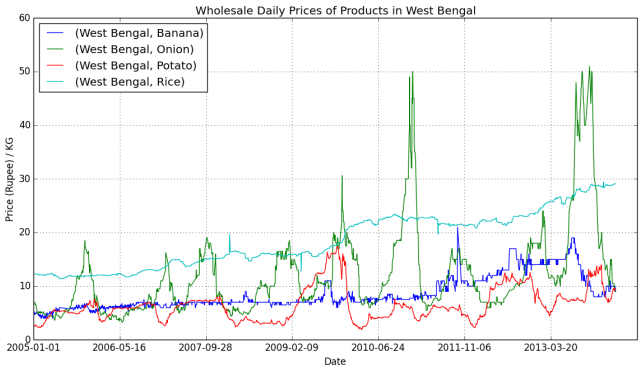


Figure 0.2: Merged series of foods in West Bengal from the wholesale daily dataset

we can gain a national overview of trends and price variations of different products.

Other sources *Joseph Boyd*

After some research we decided to include additional data that affects especially agricultural commodity prices such as the price of crude oil, climate data and the inflation in form of the Consumer Price Index (CPI). The monthly international price of crude oil was downloaded from a US government website. All other 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 *Joseph Boyd*

For the other significant part of the project we decided to limit ourselves to a basic analysis of conversations on twitter. Originally we planned to also look into other platforms such as Facebook but quickly came to realize that the data publicly available through the API was of little use to us. Obtaining data from the twitter APIs turned out to be harder than expected and involved a number of heuristics. Twitter provides different APIs for different purposes. The two that are free are the Streaming API which allows real-time streaming of only 1% of newly posted tweets. Since we wanted 'historical' tweets we had to find a workaround by using the REST API and getting tweets by user.

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 *Joseph Boyd*

Using the Twython package for python we are able to interface with the Twitter API. Our methodology (figure 0.3) is to select the twitter accounts of a number of regional celebrities as starting points. These are likely to be '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

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

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) was 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

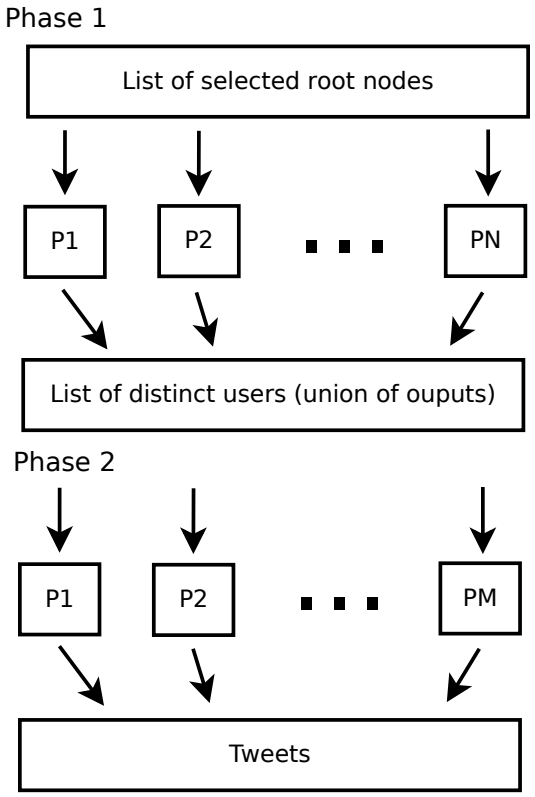


Figure 0.3: Tweet collection methodology.

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 to collecting users (phase 1), for example, by constructing a Twitter user graph.

Approach 2: Filtering tweets provided by webarchive.org *Gabriel Grill*

Before we were sure to get tweets from the twitter APIs we explored a set of archive files made available on archive.org. These archives were recorded via the twitter Streaming API and aggregated by month. They contain contain 1% of all tweets from 2011 to 2013 (not only India!). The collection of tweets 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 for disposal. About 550 GB of compressed tweets were processed and filtered in about 36 hours per node.
The applied filter got rid of tweets not containing at least one food commodity specific 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 were filtered out with this simple 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 states the exact location coordinates 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 this field is tokenized and compared with a list of known regions.

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

Daily tweet aggregator

Our first idea was to build 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, since the 'Twitter Streaming API' supplies only 1% of all available tweets. Since the amount of twitter users in India is rapidly growing, this could be a promising approach for the future.

Clustering according to keywords

Since relevant data was really sparse, we didn't expect much gain from any unsupervised learning techniques and decided to omit clustering. Instead we decided to manually explore a sample of the tweet data and create a list of indicator words used for detecting e.g. poverty, price in/decrease, sentiment and so on. Doing occurrence checks for all crafted words and storing the result, yields feature vectors for every tweet usable for prediction.

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 several times. This was due to configuration mistakes by us and Azure's 'healing'. When Azure detects anomalies, internal errors or just that it has to move images around, affected nodes are restarted, which results in a loss of tweets for us. Because of that, all filtered tweets were later stored on the main disk to avoid future losses.

Issue of localization

It's was very important for us to detect the location of tweets, because we wanted to predict volatile food prices at regional granularity. Since Twitter is not widespread in India and localized tweets are rare, we had to come up with heuristics to deal with that.

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. The 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 tweets were collected, we had to process and reformat their content for the neural networks (prediction) and the web visualisation.

Crafting indicators from tweets *Gabriel Grill*

To use the collected tweets for prediction in the neural network, aggregated indicators had to be extracted from the collection. The final result of this processing was then stored in *csv* files.
Every word occurrence check (either for filtering or feature extraction) in tweet texts was done by iterating over the list of tokens generated from the text. Several NLP techniques were used to improve the word comparison. For every tweet a category/indicator counter was kept to keep track of the number of occurrence of certain predictive words. The processed tweets and their extracted features (indicator counts) were stored in a Cassandra cluster and later queried with Shark. The result of the shark queries was then converted to *csv* files.

Sentiment analysis *Anton Ovchinnikov*

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 ***Anton Ovchinnikov***

First, several "predictor categories" were selected. These categories represent different aspects of price variation, and each category includes 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:
'available', 'full', 'enough', 'sustain', 'access', 'convenient',
and "low":
'run-out', 'empty', 'depleted', 'rotting'.

The dictionary with a 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 encounter 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 *Anton Ovchinnikov*

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 a 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.

Generation of time series *Gabriel Grill*

All these indicators are crafted into a single time series of the form *'Product', 'date', 'region', 'indicator1', ... , 'indicatorN'* via Shark queries. These indicators represent the amount of tweets matching a certain predictive category (as mentioned previously) normalized by the total amount of tweets mentioning the product that day. If a tweet is part of multiple categories, it's not clear what the intention behind the tweet was. That's why, as heuristic, we divide the sum of mentions of each predictive category by the total amount of identified mentioned categories for that tweet. We have a script to generate queries for all products and a script to parse the output into *csv* format. The figure 0.4 illustrates the whole

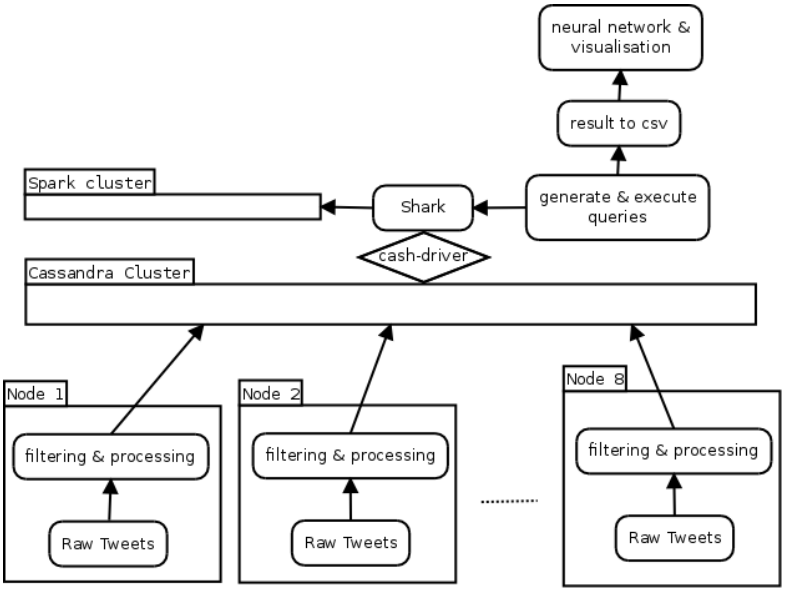


Figure 0.4: Tweet processing pipeline.

processing pipeline and gives a good overview of the architecture at the same time as well.

Infrastructure & Architecture *Gabriel Grill*

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 because it was not essential for the result that the Spark cluster had to be made up of all nodes. Luckily Shark on top of Spark with only one node was still fully functional and executed queries at reasonable 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 Data Analysis

Time Series Analysis *Ching-Chia Wang*

Food price 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, and seasonality of our price datasets in R.

Test of Stationarity of Price Series

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 have conducted the KPSS test for stationarity on some representative series from the price dataset. A test result of a sample series is presented in the table below. In general, the price series of foods in India have p-values less than 0.05, which indicates that we have to reject the null hypothesis that the series is stationary. On the other hand, the first differences of the price series have p-values larger than 0.05, which means that they are potentially stationary time series. Based on this test result, we will use the first difference of the series in later analysis.

	Price Series	First Difference of Price Series
p-value of null hypothesis "Trend"	0.01	0.1

Table 0.2: Table of results of KPSS-test of stationarity on the sample series: Price of "Jyoti" Potato in West Bengal. A p-value less than 0.05 indicates a rejection of the null hypothesis that the series is stationary under the 95% confidence level. [3]

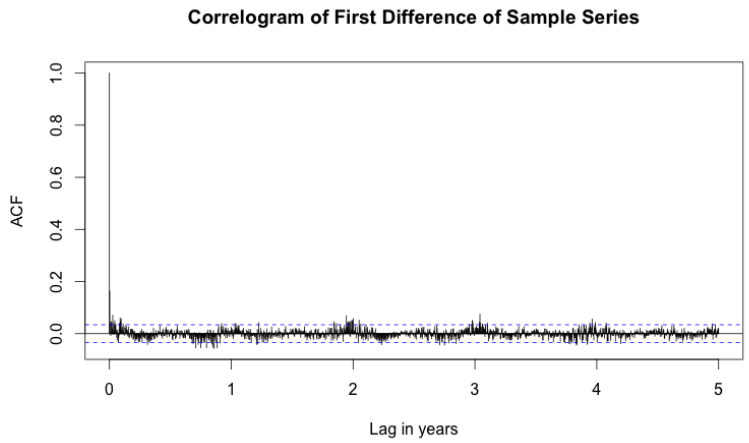


Figure 0.5: Correlogram of the first difference of the sample series: Price of "Jyoti" Potato in West Bengal.

Autocorrelation of Price Series

Autocorrelation(ACF) is another important trait in time series data. It suggests the degree of correlation between current value and the value with a period before. 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.

The figure below is the correlogram of a sample series. The dotted line indicates the region of white noises. Any signal located inside that region is considered statistically insignificant. The ACF value at lag 0 is fixed at 1 because current value is 100% correlated with itself. Observe that we have some short positive signals near lag 0, and there appears to be some annual patterns in price changes. The current change in price of the series seems to be positively correlated with those a few months before, and a few years before, up to 4 years.

Seasonality

Seasonality is reasonably expected in our agricultural related time series. We have conducted a standard seasonality decomposition on our price dataset. One sample result is shown in Figure 2. The price series in the first subplot is decomposed into 3 series: seasonal, trend, and remainder. Slight annual patterns are presented, but note that they are in small magnitude comparing to the price series. One important observation is that the remainder series is highly volatile and inconsistent over time.

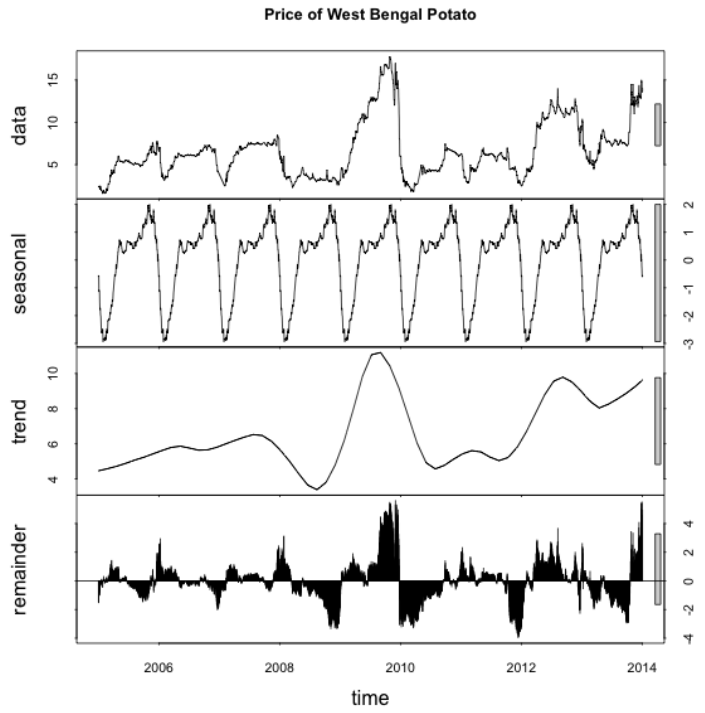


Figure 0.6: Correlogram of the first difference of the sample series: Price of "Jyoti" Potato in West Bengal.

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 sufficiently cleaned and interpolated.

Time Series Forecasting *Ching-Chia Wang*

Regression with ARMA error

In our regression model, the target vector is the sum of linear combinations of multiple regressors and an ARMA (Auto-Regressive Moving Average) error shock. Our regression model is formulated as such:

$Y = \beta * X + \epsilon, \epsilon : ARMA(p, q)$,

where Y is the target fitted variables (the price series), X contains the regressors, β is a linear coefficient matrix, and ϵ is the error shock. The ARMA modeling of error shock consists of the auto-regressive part (lag variables) and the moving average part (effects from recent random shocks). The fitting of the ARMA model and the historical error is accomplished by maximum likelihood estimation. On the other hand, through OLS (Ordinary Least Squares) or GLS (General Least Squares) processes, we can obtain an optimal β which minimized the mean square error between the predicted Y and the actual Y.

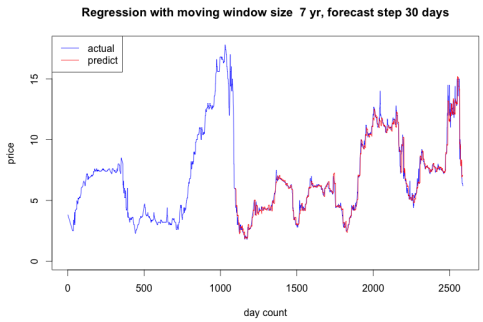
The prediction dataset consists of a food price series, an oil price series, an inflation index series, 2 weather series (temperature and rainfall), and some extra series generated from the previous 3 series with pre-defined time lag in quarters or years. By introducing some extended series, we try to capture the annual cyclic pattern observed in the time series analysis. The food price series is used as the target prediction variable, while the others are regressors.

We use the regression model to predict the price series in a rolling fashion using R. First, a window of some fixed size, such as 3 years, defines the range of the training data. After fitting the model, the algorithm will then forecast a fixed length of future days after the training window. Then, the end of the window moves to the end of the previous forecasted dates so that the process can predict for the next days. This process runs recursively, and our price prediction is in continuous days up to the end of the dataset.

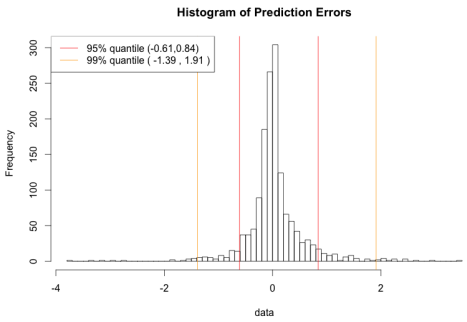
Our experimental result is shown in the figure and table below. The 95% or 99% quantiles of prediction error in the table means that 95% or 99% of the prediction errors fall inside the interval of the quantiles. As forecasting days increase, the root mean square error and the quantiles rise too. The best result is using a moving window size of 3 years and running 1 week price forecasting as 95% of the prediction errors are less than 1 rupee.

Moving Training Window Size	3 years	3 years	5 years	5 years
Rolling Forecast Step	1 week	1 month	1 week	1 month
RMSE	1.0023	1.3397	1.8486	3.0762
95% Quantiles of Prediction Error	(-0.61, 0.84)	(-1.63, 2.18)	(-0.67, 1.12)	(-2.24, 2.07)
99% Quantiles of Prediction Error	(-1.39, 1.91)	(-2.68, 3.7)	(-2.61, 1.94)	(-4.06, 2.68)

Table 0.3: Prediction result of regression with ARMA error shock with different training moving window size and forecasting steps.



(a) Prediction result of the sample series: Price of "Jyoti" Potato in West Bengal.



(b) Histogram of prediction errors of the sample series: Price of "Jyoti" Potato in West Bengal.

Figure 0.7: Example of time series cleaning process

Neural networks - introduction *Stefan Mihaila*

Artificial neural networks are computational models with very varied applications in pattern recognition. Their capability of learning impressively complex patterns and finding deep meaning in data have caused such models to gain widespread attention and become a very popular choice for a large number of prediction tasks. For the purpose of price predictions, we have mainly focused on two different types of neural networks: *feed-forward neural networks* and *echo-state neural networks*.

Feed Forward Neural Networks - Multilayer Perceptrons

Stefan Mihaila

A feed-forward neural network is generally described by the number of neurons and the connections between them. Each connection is assigned a weight (real number), and the value (the activation in neuro-slang) of a neuron is computed as the weighted sum of its input neurons. This value is then passed through a non-linear function that "squeezes" the real number into $[-1, 1]$. It is this non-linear mapping that allows neural networks to learn non-linear patterns from the data.

In the case of feed-forward neural networks, the neurons and the connections form a directed acyclic graph. In other words, a neuron cannot have a connection to itself (neither directly nor through any path). In this sense, the information is only sent "forward" (in one direction). Such a network can be arranged in layers, based on the existing connections. Therefore, a more practical way of describing a FFNN model is by the number of layers and by the connections between them (most often full connection between neurons is used).

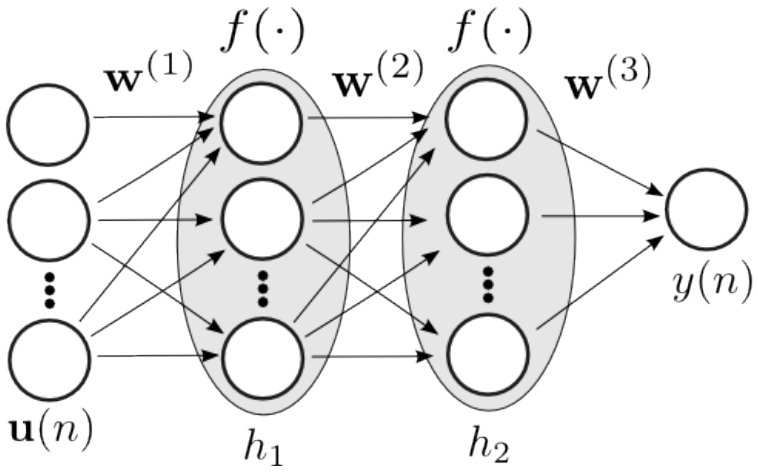


Figure 0.8: A feed-forward neural network with two hidden layers

Two layers, namely the first and last, have special meaning: they are called the *input layer* and *output layer*, respectively. The other layers (the ones in the middle) are called *hidden layers*.

The *input layer* is used as the "source" (the place from where the data is fed into the network), and the *output layer* is viewed as the "sink" (the place where a prediction is made on the input data). If our data-set has 5 input columns and one target column (to be predicted) then we will place 5 neurons in the input layer, each being responsible for taking values in one of the columns and one output neuron in the output layer.

The process of training a neural network is the process of assigning weights (real values) to all the connections in such a way that when we are feeding input from our training set into the network, we get an output very close to the correct output. This closeness is measured in terms of an error function (such as the mean squared error). In order to optimize the weights of such a network, we have to find a minimum of this error function, process which is achieved most commonly through *gradient descent*. The process of adapting the weights after every gradient descent step is called *backpropagation* (because we are going backwards from output to input).

Note that unlike echo-state neural networks, FFNN do not have an internal state (i.e. do not have a memory). As such, they are most commonly used to train non-sequential data (data for which the order of the data-points does not matter). This is obviously not the case for a time series. However, a FFNN can be abused into learning sequential data by feeding a sliding window of input over a given interval.

Feed Forward Neural Networks - models used *Stefan Mihaila*

During our project, I have thought of numerous FFNN models that would make sense to be trained, but most of my ideas in the beginning were overly optimistic: I assumed we can train a single large neural network to simultaneously predict all prices for all products in all regions; feeding all of those inputs at once into the network would have been very useful as it would have allowed the training process to find correlations across different regions and different products.

However, it turned out that this was impossible to do, as the available time series are way too sparse and gapped to allow such models to be trained. As such, I resorted to individually training a network for each of a few wholesale daily time series which had good data (downloaded by Anton, selected, preprocessed and interpolated by Ching-Chia).

As some time series are more complex than others, I had to try different model hyperparameters for each time series (to have good generalization and prevent overfitting). The hyperparameters are the number of hidden layers and the number of neurons in each hidden layer. I have mainly worked with the following models of 2 and 3 hidden layers: $(10, 10)$, $(10, 10, 5)$, $(15, 15)$, $(15, 20)$, $(20, 20)$. For all the models, I have used fully connected layers (i.e. each pair of consecutive layers were fully connected).

To maximize the prediction power of the trained network, I have used 40 inputs to the network:

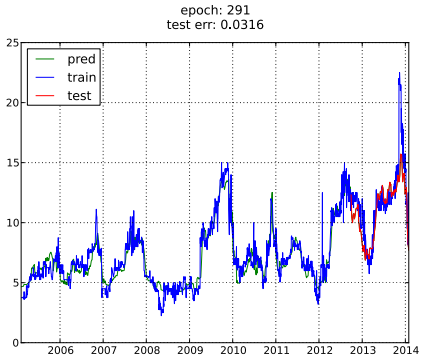
1. A sliding window of the normalized per-region monthly *precipitation* amounts over the last 9 months (9 input neurons)

2. A sliding window of the normalized per-region monthly mean *temperatures* over the last 9 months (9 input neurons)

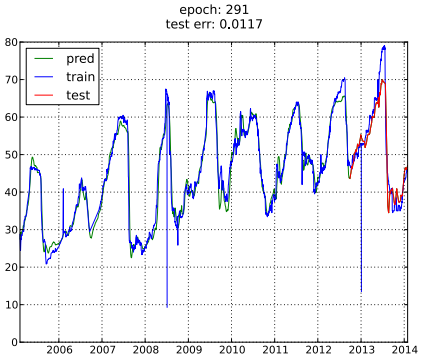
3. Sliding window for last 9 months of normalized *international oil prices* (9 input neurons)

4. Sliding window for last 4 months of the *consumer price index* (4 input neurons)

5. Sliding window for the last 36 days of prices, in 4 day jumps (9 input neurons)



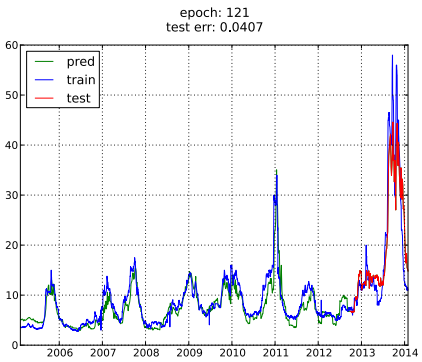
(a) Wholesale daily prediction for potato in Vadhvan (Gujarat)



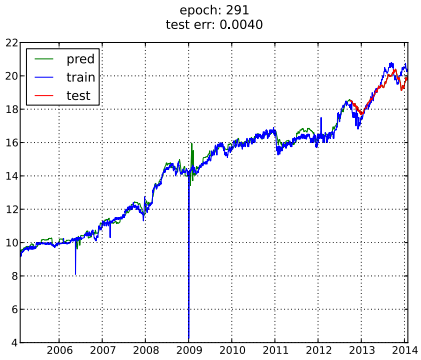
(b) Wholesale daily prediction for apple in Bareilly (Uttar Pradesh)

Figure 0.9: Four-day-ahead predictions with the FFNN on various time series

Because the prices are sometimes stable for a few days in a row, I am trying to predict prices in 4 day jumps (i.e. given input at $t-36, t-32, t-28, \ldots, t$, the target is the price at $t+4$). I am then filling in the predictions for $t+1$, $t+2$ and $t+3$ using linear interpolation between t and $t+4$.



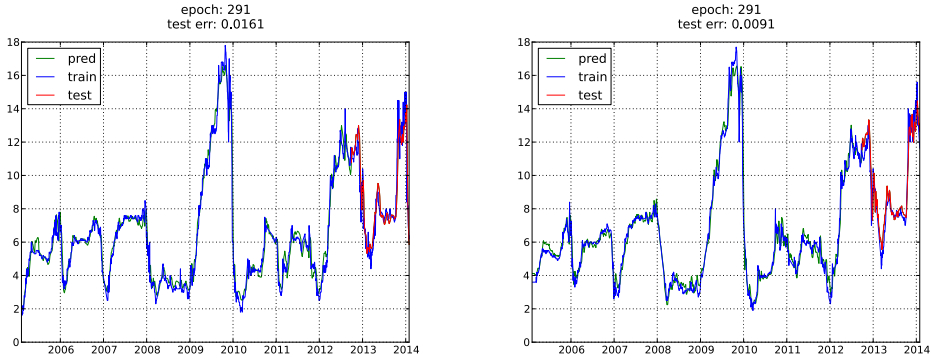
(a) Wholesale daily prediction for onion in Bareilly (Uttar Pradesh)



(b) Wholesale daily prediction for rice in Bareilly (Uttar Pradesh)

Figure 0.10: Four-day-ahead predictions with the FFNN on various time series

Such a network should be able to predict 4 days in advance without receiving any feedback. However, after 4 days, the network should receive feedback. I believe this can be stretched for longer periods, but the limited time and the long durations of training the models has not allowed me to test this hypothesis.



(a) Wholesale daily prediction for potato in Burdwan (West Bengal)

(b) Wholesale daily prediction for potato in Champadanga (West Bengal)

Figure 0.11: Four-day-ahead predictions with the FFNN on various time series

On the time series we have selected, the FFNN seems to be performing quite well. The testing of the neural network was performed as follows: the first 85% of the time series was used for training the network, the last 15% of the time series was used for testing. The blue plot highlights the true series, the green plot shows the network prediction for the train data (naturally very good, as it has already seen the data) and the red plot shows the prediction over test data (data it has never seen). The performance can be estimated by comparing the red plot to the blue plot. Remember that the network still gets feedback on how it's doing, but only every 4 days.

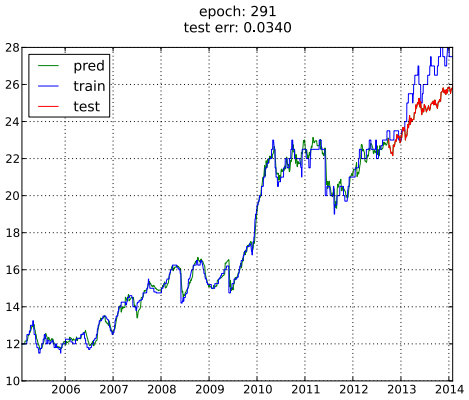


Figure 0.12: Wholesale daily prediction for rice in Champadanga (West Bengal)

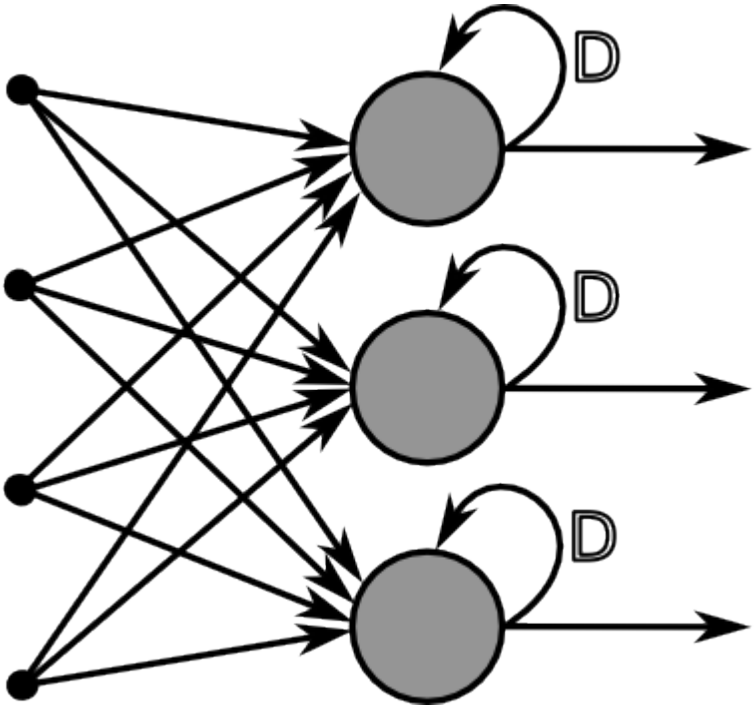


Figure 0.13: source: wikipedia

Recurrent Neural Networks (RNN) *Fabian Brix*

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. As opposed to feedforward neural networks where a layer l can only receive inputs from a previous layer, in an RNN the weight matrix for a layer l can contain input weights from *all* other neurons in the network. The simplest form of an RNN consists of an input, an output and one hidden layer as depicted in 0.13.

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 exactly 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*. This part of the report is based on [1].

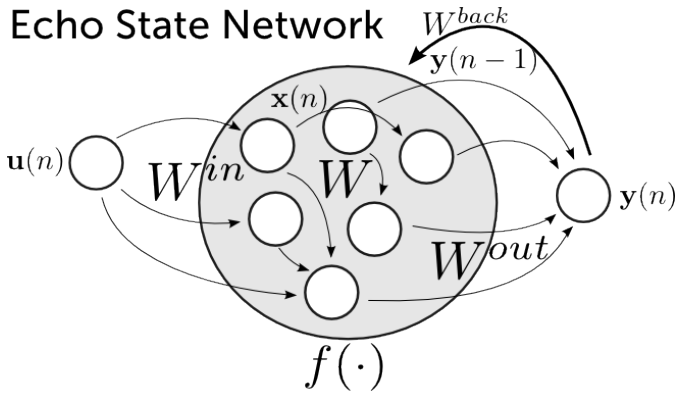


Figure 0.14: Network structure of an ESN with 1-dim. input and output

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 it suffices to make sure the negation of the second point holds.

Training the ESN

The state of the ESN is a function of the inputs it has been presented with. Each state describes an oscillator at a given time n and during training the relation between these oscillators and the output has to be learnt. Before training can be started the reservoir weights have to be normalized by the spectral radius of the weight matrix W. The hyperparameters are the reservoir size (dimensionality of W) and a rescale parameter of the weight matrix for slightly increasing/decreasing the spectral radius. This part of the report is baed on a comprehensive overvie of practical issues of ESNs [2].

Initial state determination Prior to training the Echo State Network has to be run for an initial set of inputs discarding 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. We used an initiation sequence of 740 points which equates to two years of data.

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 Tikhonov 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 Tikhonov 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 possibility 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:

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

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

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 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 which is represented by the output weights W^{out} in the case of

the ESN. Essentially, LMS algorithms are stochastic gradient descent methods that perform updates according to the error present at time n that update the weights according to the a priori error and a predefined constant in every iteration.

Initialization:
$W^{out}(0) = \mathbf{0}$
Update steps:

1. $e(n) = y_{teach} - W^{out}(n-1)x(n)$
2. $W^{out}(n) = W^{out} + \mu e(n)x(n)$

- **Recursive Least Squares (RLS):** RLS algorithms have a faster convergence speed compared to LMS algorithms. RLS optimizes the discounted square a priori error $\sum_{k=1}^{n} \lambda^{n-k}((y_{teach}(k) - y(n))^2$ with forgetting factor λ at every timestep n. All signals generated in the sequence $1 \leq k \leq n$ are taken into account using the inverse correlation matrix of inputs P. The inputs for the adaptive filter in the case of the ESN can be the concatenation of input, generated echo states and last output.

  Initialization:
  $\lambda < 1$
  $P_0 = \delta I$ where $\delta \propto \sigma(input)$
  $W^{out}(0) = \mathbf{0}$
  Update steps:

  1. $\pi(n) = P(n-1)^T \cdot x(n)$
  2. $\gamma(n) = \lambda + \pi(n)x(n)$
  3. $k(n) = \pi(n)/\gamma(n)$
  4. $e(n) = y_{teach} - W^{out}(n-1)x(n)$
  5. $W^{out}(n) = W^{out} + k(n)e(n)$
  6. $P(n) = [P(n-1) - k(n)\pi(n)s]/\lambda$

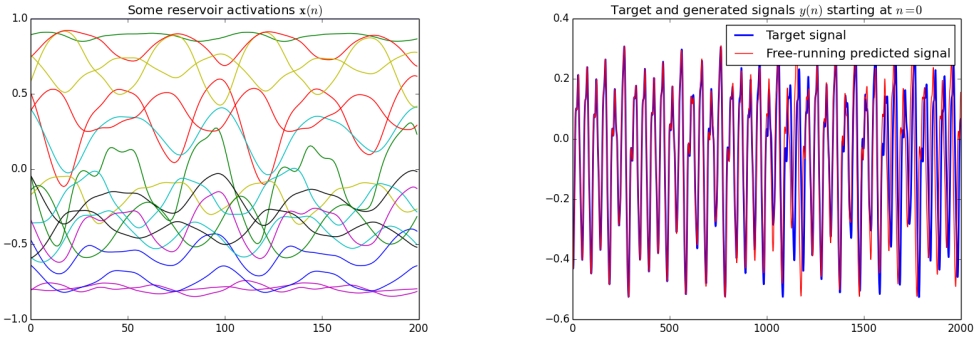
The standard RLS algorithm is unstable during training, diverges when the $P(n)$ loses positive definiteness. Choosing λ very close to 1 was temporarily enough to alleviate the problem. The QR decomposition-based RLS (QR-RLS) algorithm can, however, resolve completely this instability. Instead of

working with the inverse correlation matrix of the input signal, the QR-RLS algorithm performs QR decomposition directly on the correlation matrix of the input signal. Therefore, this algorithm guarantees the property of positive definiteness and is more numerically stable than the standard RLS algorithm.

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 tried to create replicate time series of a selected price sequence dataset. The method employed 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. We did, however, not manage to fully implement the algorithm from how it was described in the paper.

Demonstration

In this section we demonstrate the predictive power of the simplest ESN configuration with a reservoir size of 1000 neurons. Ridge Regression is used to train on and predict a time series generated using the Mackey-Glass differential equation and noise.



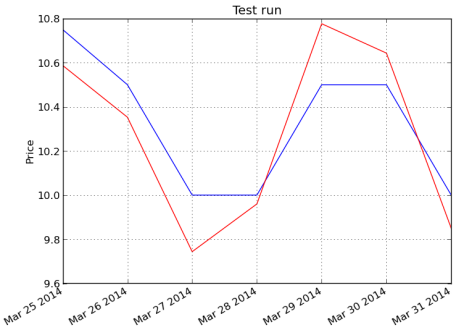
(a) Sample of ESN activations resulting from the training series

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

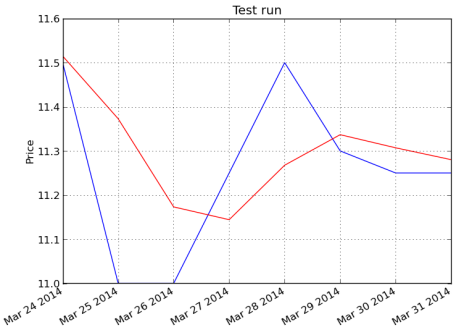
Figure 0.15: Results of simple ESN trained with Ridge Regression on a Mackey Glass time series consisting of 2000 points; parameters: spectral radius 1.25, leaking rate 0.3.

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 basis without any need for retraining. 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 in isolated cases where the dynamics of the training period favoured a prediction and we are not sure if greater accuracy is achievable with these models.



(a) Wholesale price of potato, state Gujarat; test RMSE in INR: 0.199

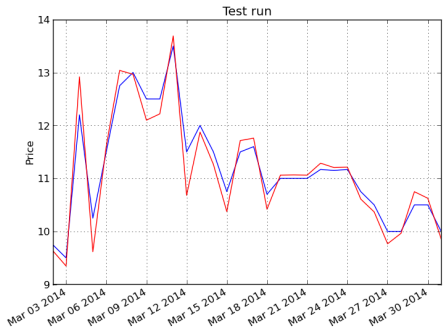


(b) Wholesale price of red onion, state Uttar Pradesh; test RMSE in INR: 0.145

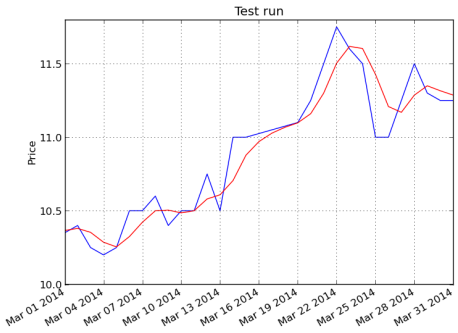
Figure 0.16: 7 day day-to-day prediction for two different commodities in different states

Adjusted for inflation the prices mostly lose their upwards trend and should become easier for the network to predict. This is the case, but good predictions are again obtained more by chance than anything else by selecting subsets of a series to train on. Modeling the dynamics of the whole series seems to remain too complex for the network.

In their current state, due to the 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 data monitoring of the Indian government.

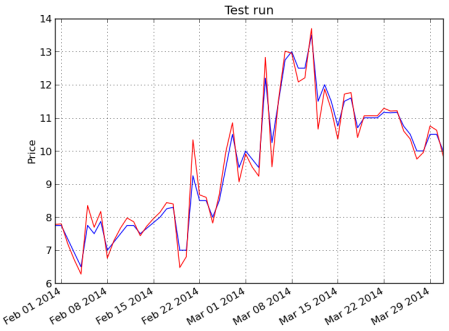


(a) Wholesale price of potato, state Gujarat; test RMSE in INR: 0.293

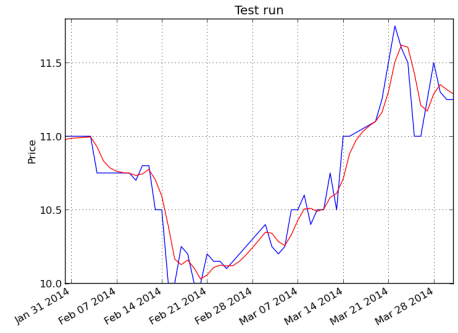


(b) Wholesale price of red onion, state Uttar Pradesh; test RMSE in INR: 0.145

Figure 0.17: 30 day day-to-day prediction for two different commodities in different states



(a) Wholesale price of potato, state Gujarat; test RMSE in INR: 0.310



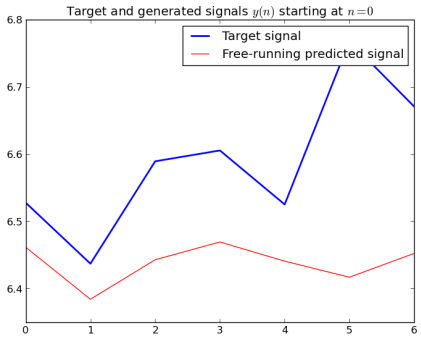
(b) Wholesale price of red onion, state Uttar Pradesh; test RMSE in INR: 0.127

Figure 0.18: 60 day day-to-day prediction

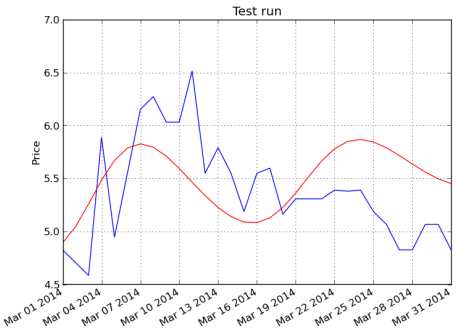
Improving predictions, if possible, would certainly require a lot more focused research in the area of recurrent neural networks for time series prediction. The ESN was implemented from scratch in scientific python.

Further ESN research

Further research concerning the stability of online training of the ESN and its predictability power would encompass the implementation of a stable RLS algorithm



(a) Wholesale price of red onion, accurate prediction produced for 5 days by training with a limited amount of data somewhere in the middle of the series



(b) Wholesale price of red onion, trying to reproduce accurate prediction by training on the last two years, best result of various parameter choices

Figure 0.19: Results from training with ridge regression on a very limited amount of data maximum three years. The prediction can be gotten about right even for one month horizon. However, so far it adjusting the training phase and the spectral radius is more of a game of chance. The influence of the reservoir size seems to be neglectable having tried sizes of 500, 1000, 2000.

such as the Inverse QR-RLS as well as the implementation of the Backpropagation and Decorrelation (BPDC). The latter does not require a specific set up of the network rendering concerns about the initialization run and the spectral radius of the weight matrix void. Using these algorithms a performance analysis of the network has to be done before we can test the influence of additional inputs. Maybe research into Linear System Theory can provide answers as to how to make the network responsive to its own inputs in the generative run. It would lastly be interesting to compare the ESN to its "rival" the Long Short Term Memory Network trained with the Evolino method.

Data Mining

Alexander Buesser

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 can 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 categorical 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 ***Duy Nguyen***

We dynamically represent the results of data collection and prediction on the Indian map. In particular it includes Twitter tweets data (total number and by keywords), historical price data of commodities, and prediction of commodity prices. The live demo version can be accessed through this URL.

Technologies

LeafletJS: A javascript framework that builds the world map on top of OpenStreetMap library. MapboxJS was also used to extend the functionality of different behaviors on the map as well as enhance the UI design.
HighchartsJS: A charting library written in pure JavaScript, offering an easy way of adding interactive charts to websites or web applications. It supports various chart types, from popular ones such as line, area, column, bar, pie, scatter, to more sophisticated types including angular gauges, arearange, areasplinerange, columnrange, bubble, box plot, and so on. Furthermore the functionalities of zooming to a smaller timeframe or printing charts in different formats could facilitate investigating and researching on timeseries data.
And last but not least, our visualization was developed in HTML5 which allows dynamic manipulation of visualization concepts as well as easy access through web browser by various targeted users.

Geo-visualization

The map shows colored states in India by average monthly tweets per inhabitant. Data is monthly from 2007 to 2014 so sliders for year and month can be used to navigate between different times.
The behavior of clicking on a state will zoom in and display prominent cities in that state. Monthly number of tweets for cities are also available on hovering the city markers.

Monthly number of tweets per inhabitant in states

Number of tweets include all the tweets we collected in data collection phase for the specific state. This is divided by the population of state to come up with the final number of tweets per inhabitant. For the sake of user-friendliness those values are displayed in the format i * 10^{-6}.

Unit visualization

This visualization is shown after choosing a particular state or city on the map. Further contents in tabs depend on whether the chosen location is a state or city.

- *State:* Available contents include average monthly tweets per inhabitant in that state from 2007 till now, comparison of daily tweets about several commodities in that state from 2008 till now, merged daily retail prices of commodities for all cities in that state from 2009 to 2013, merged daily wholesale prices of commodities for all cities in that state from 2005 to 2013, and results of predicted prices for commodities.

- *City:* Available contents include total monthly tweets in that state from 2007 till now, association rules for data mining of commodities, daily retail prices of commodities from 2009 to 2013, daily wholesale prices of commodities from 2005 to 2013.

Daily number of tweets per inhabitant in states

This is similar to the data displayed on geolocation map, with a slight difference in interval by daily instead of monthly. Specific periods can be investigated by selecting a timeframe.

Daily number of tweets by keywords in states

In order to facilitate commodities price prediction, tweets are categorized by food-related keywords such as "rice", "wheat", "fish", etc. Their daily occurences are represented by lines so that we can make a comparison about which commodities are usually mentioned on Twitter, which are useful for the prediction (via sentiment analysis phase), and so on. On the graph viewers can turn "on and off" the indicators to focus on the most interesting keywords as they wish.

Historical wholesale/retail price of commodities by state and city

We use timeseries analysis for the historical wholesale/retail price of commodities, and their results are visualized by state and city. The main analysis bases on cities, while the results were also merged for states to produce a more macro view on those data.
Furthermore, like mentioned above, turning indicators "on and off" is also supported to reduce the distraction created by many goods being displayed on the chart.

Results of price prediction in states

As an extreme target of our project being the price prediction for commodities, on this visualization viewers can choose between predictions of different goods having been analyzed. Actual and predicted prices are compared by the two paralleling lines.

Some observations *Duy Nguyen, Fabian Brix*

By investigating the visualizations of different data (historical prices, social media activities, distribution of Tweets by population, etc.), we identified several interesting trends as following:

- Our visualization of tweets per region exemplifies the result of poverty in India's most populous state of Uttar Pradesh (200 million inhabitants) in the use of social media and probably Information & Communication Technologies (ICTs) in general. When moving through time in the visualization one can see that the states Karnataka (Bangalore), Maharashtra (Mumbai), NCT of Delhi are constantly ahead of Uttar Pradesh in the number of tweets they produce although their population is significantly smaller. Another phenomenon that can be observed is that Maharasthra and Karnataka are having a head-to-head race starting in 2007 although the population of Karnataka is only half the size of that of Maharashtra. This can be attributed to the fact

that the state capital Bangalore is the major IT hub of India and therefore boasts more early adopters. In 2010 as more and more people take to twitter Maharashtra raises far ahead in the number of tweets produced.

- We have tried creating the map with both approaches: monthly number of tweets and monthly number of tweets per inhabitant. Interestingly the distribution shown on the visualization seems to stay put, which indicates that the number of tweets in states are proportional to their population. The most significant difference, as mentioned above, is in Uttar Pradesh which has the largest population but not as many tweets compared to other regions.

Conclusion & Future Work

In conclusion the project team can be proud of the results of the project. Although we were held up by many data collection and data quality issues we managed to experimentally construct a pipeline from collection tweets to time series of tweet indicators and price data collected online to cleaned an interpolated time series. The neural networks and incorporation of additional data sources did not deliver the predictive power hoped for yet once trained provide very accurate day-to-day predictions without having to be retrained. Due to the many issues we encountered with collecting, setting up the infrastructure and analyzing tweets we were not able to collect enough to data to evaluate the use of twitter indicators. However, since we have successfully set up the infrastructure and filtering of tweets we are positive that this can be done in a follow-up project.
In order to refine the prediction models used and to complete the pipeline it might be a good idea to reapply this approach to the Indonesian national price table that has recently been published by the Indonesian Ministry of Trade. Twitter statistics very much favor Indonesia with 11.7% of the population using twitter over India where the percentage only amounts to 1.3%.
Taking the idea of twitter indicators further our map visualization shows that the volume of tweets in India has massively increased in the last few years. By continuing the collection of tweets from India the tweet indicators could be bolstered and compared to price time series. If significant relationships are found the tweet indicators could be used to support real-time prediction of prices when prices monitored by the government are not yet available.
Another interesting point would be to take the analysis of the social media content further in order to figure if it is possible to track the patterns of food supply in throughout the country. Such a system could help the highly decentralized Indian government in managing the allocation of resources including food market interventions.

Closing remarks

The whole project team would like to sincerely thank Professor Christoph Koch for giving us the possibility to propose this project and implement it despite the uncertainty of its outcome. We would further like to thank our TA Aleksandar Vitorovic for his valuable criticism, advice and motivation.

Bibliography

[1] Jaeger, Herbert. The “echo state” approach to analysing and training recurrent neural networks. Fraunhofer Institute for Autonomous Intelligent Systems. January 2010.

[2] Lukosevicius, Mantas. A Practical Guide to Applying Echo State Networks. Jacobs University Bremen. Appeared in: Neural Networks: Tricks of the Trade, Reloaded. Springer. 2012.

[3] 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.