Robust application load forecasting project Technical Report

Xiaoming Zhang

Feb 13, 2019

1 Model basics

DeepAR is one of the built-in algorithms available on AWS Sagemaker machine learning platform. It is built on a recurrent neural network architecture with LSTM cells. What DeepAR does is probabilistic forecasting: It predicts the conditional probability distribution, given the joint probability of past observations, including previous predictions. For example, if we specify a Gaussian likelihood, the network will predict the parameters μ and σ for the Gaussian distribution; or μ and shape parameter α for negative binomial; degree of freedom ν for student's t, and so on. Given the probability distribution parameters, we can obtain the prediction median, confidence intervals or various quantiles, and compare with the target values.

1.1 Automatically generated features

Given N input time series $\{\mathbf{z}_{i,1:T}\}_{i=1,\cdots N}$ at hourly frequency, DeepAR will generate feature time series (covariates $\{\mathbf{x}_{i,1,1:T}\}_{i=1,\cdots N}$, $\{\mathbf{x}_{i,2,1:T}\}_{i=1,\cdots N}$, $\cdots$) such as hour-of-day, day-of-week, day-of-month, day-of-year. DeepAR will also use lagged values from target time series as well. These strategies help well capture the seasonality of the time series.

1.2 Handle scaling

Time series can have very different magnitudes. Take the Wiki pageview data for example, the mean amplitude of different time series could vary from ~ 10 to $\sim 10^6$. And the distribution of the mean amplitude is skewed, dominated by smaller scale time series. DeepAR handles different scales of input time series with two strategies: normalization and weighted sampling. For the former, the input time series are normalized by its mean, that is, taking a simple scaling by dividing the input by the sample average value $\nu_i = 1 + 1/t_0 \sum z_{i,t}$. For the latter, DeepAR takes weighted sampling scheme when sample mini batches during stochastic gradient descent process. The weight is proportional to ν_i.

2 Error metrics and interoperation

DeepAR provided error metrics internally, including the final loss (for training data), root mean square error (RMSE), weighted quantile loss. If we specify the test channel, RMSE and quantile loss of test data will be provided in the training log. During the hyperparameter tuning, final loss, RMSE and the mean weighted quantile loss can be chosen as target metric.

I also provided error metrics externally, which include different flavours of Symmetric mean absolute percentage error (SMAPE) and Mean absolute percentage error (MAPE).

SMAPE is defined as follows,

$$\text{SMAPE} = \frac{1}{n}\sum_{t=1}^{n}\frac{|F_t - A_t|}{|A_t| + |F_t|} \cdot 100\%,$$

where F_t and A_t are the forecasted value and actual value at time t, respectively. When $|A_t| + |F_t| = 0$, SMAPE $= 0$. Note that the SMAPE could also have a slightly different form $\frac{1}{n}\sum_{t=1}^{n}\frac{|F_t - A_t|}{(|A_t| + |F_t|)/2} \cdot 100\%$.

I used the former definition because: 1) the former version is bounded between 0 to 100%, which is more intuitive and easier to interpret. 2) The Kaggle web traffic time series forecast competition used the former formula to evaluate competition submissions, it would be interesting and useful to directly compare our error score with the Kaggle competition winners.

The original SMAPE has a sudden drop at $|A_t| + |F_t| = 0$, we can also define a variant form to smooth this drop:

$$\text{SMAPE} = \frac{1}{n}\sum_{t=1}^{n}\frac{|F_t - A_t|}{\max(|A_t| + |F_t| + \epsilon, 0.5 + \epsilon)} \cdot 100\%$$

where ϵ is a very small positive number.

The advantage of using SMAPE as error metric is that the SMAPE is bounded, it takes values in [0,1] For both extreme over-forecasting $F_t \gg A_t$ and extreme under-forecasting $F_t \ll A_t$, SMAPE $\to 1$. In the cases of perfect prediction in which the $F_t = A_t$, SMAPE $= 0$. SMAPE have nearly **equal penalty** for over-forecasting and under-forecasting.

MAPE is another choice of metric to evaluate forecasting errors:

$$\text{MAPE} = \frac{1}{n}\sum_{t=1}^{n} |\frac{F_t - A_t}{A_t}| \cdot 100\%$$

The MAPE metric is not bounded, it takes values in $[0, \infty)$. In the extreme over-forecasting case, MAPE $\to \infty$, and in the extreme under-forecasting case, MAPE $\to 1$. An important property of MAPE is that it has **heavier penalty** on over-forecasting than under-forecasting. One possible way to reduce this imbalanced penalty is to use the log variant MAPE $= \frac{1}{n}\sum_{t=1}^{n} |\log(|\frac{F_t}{A_t}|)| \cdot 100\%$, which is still unbounded ($\in [0, \infty)$) but have nearly equal penalty for under- and over-forecasting.

3 Source of data

To train the model, we turn to public available data set, WIKI hourly pageview data. They are enormous in amount and rich in different patterns. We combine two sources of hourly pageview data:

1. Pageview at wiki-project level, for example, pageviews for species.wikimedia.org, de.wikipedia.org. I access this part of data from wikimedia REST API. Data are available from 2016 to present. To access the data, we need to specify the project/access method (desktop, mobile) / agent (user, spider) /start and end time in the api url. After sending GET request to wiki api, we will get an response of json objects , it is an hourly pageview data from the specified project within the given period.

2. Pageview at per-article level, such as page-view for `White House`. I access this part of data from Google Bigquery public tables (e.g. fh-bigquery:wikipedia_v3). Data are available from 2015 to present. I queried the table with SQL, saved the query result to my own project, downloaded the data as JSON lines, each line contains a JSON object with "title" and "time stamp".

After obtaining the raw data, I reformatted them to JSON lines that DeepAR can consume.

4 Time series characterization

In this section we will go through how to categorize the time series and then segment the data into different buckets.

- Please refer to the JupyterNotebook `notebooks/visualization/characterize-and-segment-time-series.ipynb` for detailed demonstration of different time series patterns and data segmentation.

The current characterization tool is able to identify half-day / day / week / month seasonality, day-impulse (a special day-seasonality), trend, and spiky (random noise) patterns and quantify them by giving a score that indicates how strong this characteristic is.

The input is a time series in `Pandas.series format`, `window`. The output is a list containing scores and type, arranged as follows `[trend score, half day score, day score, week score, month score, 'strongest characteristic', mean in time domain, standard deviation in time domain]`. The `'strongest characteristic'` is a str, takes one of the values in Day, hDay, week, month, trend, spike. The math behind the scene is a signature peak scan on frequency spectrum. This is done by applying a discrete Fourier transform on the time series then scanning for peaks.

We apply discrete Fourier transform to an input time series through FFT. For signal X of length n, its Fourier transform $Y = \text{fft}(X)$ is defined as follows,

$$Y(k) = \sum_{j=1}^{n} X(j) W_n^{(j-1)(k-1)}$$

where

$$W_n = e^{(-2\pi i)/n}$$

is one of n roots of unity.

The time step between successive points in the time series is Δt, so the sampling frequency is $F_s = \frac{1}{\Delta t}$, and the Nyquist frequency will be $F_N = 1/2F_s$. The time series have n points, so the frequency interval is $\Delta F = \frac{1}{n\Delta t}$. The frequency term of FFT will be complex numbers ranges from $-F_N$ to $+F_N$. We keep the positive half, then plot the amplitude (as absolute value of the complex component). After that, we do a simple conversion, convert the frequency to cycles (how often the signal repeats itself in a given time window) that are easy to interpret: $cycle = frequency \cdot window$. Here are two examples of how to determine the window parameter:

- Given the time series at hourly frequency, then window = length of a day / sample interval = 24 H / 1 H = 24.
- Given the time series at 30 min frequency, then window = length of a day / sample interval = 24 H / 30 min = 48.

The location of the frequency peak indicates whether it's periodic / trend / random noise, a few examples:

Time: seasonality with period T $\rightarrow$ Freq: peak at $1/T$ (T should not be too large so that $1/T > \Delta F \cdot window$ to separate it from the zero frequency component). Half day seasonality peaks at 2, day peaks at 1, week peaks at ~ 0.143, month peaks at ~ 0.033, etc.

Time: monotonic time dependency or single/composed Gaussian pulse $\rightarrow$ Freq: peak at ~ 0 with vary fast decay, typically diminishes around ~ 0.01 in this case.

Time: constant signal $\rightarrow$ Freq: zero frequency component

Time: random noise with constant mean $\rightarrow$ Freq: random noise

Time: $\delta(x - x_0)$ $\rightarrow$ Freq: $e^{-i2\pi(x-x0)k}$

Time: periodic impulse $\rightarrow$ Freq: periodic impulse

The amplitude of the peak reflects how strong this characteristic is. I quantify this by the characteristic score: if the input time series have certain characteristic, it gets a score (normalized amplitude); if not, it gets a 0 score in this category. The characteristic score is normalized both within the frequency spectrum (by divided over number of time points) and across different time series (normalized to N(0,1)). This means we can compare within one time series, and we can compare across different time series.

Finally, we determine the type of the time series by it's highest characteristic score. Based on the time series characterization, given the data set, we can examine what's in there:

- A break-down of different types of time series (type = the strongest characteristic of given time series. Note that time series can show multiple characteristics, e.g. periodic + trend).
- A detailed look of the score distributions for each characteristic, and whether different characteristics are correlated.

5 Model training

5.1 Handling missing values

Raw Wiki pageview data json objects have two key:value pairs we are going to use: for example, "timestamp": "2018010100", and "views": 12246074. But for our specified period between 2018010100 and 2018020100, some of the json objects could be missing. There are two possibilities: the pageview data for the timestamp might be 0 or unavailable. DeepAR requires that all the points in the "target field must be successive values at fixed time interval. We have two methods to deal with the missing values: 1. Imputation 2. Leave a placeholder to indicate the value at the particular time step is missing.

I first tried to pad the missing values to 0. This will create intermittent the time series. The error based on the same test dataset is obviously larger than results by the method 2. Other imputation schemes include fill mean, forward/backward fill and interpolation.

Imputation values in time series would alters important properties of the time series signals such as the phase or the frequency spectrum. As the model predicts the conditional distribution of the next values, given the joint probability of the previous ones. Any imputation will actually adding more information into the time series, and will thus change the joint probability of the previous ones. Mathematically it is makes more sense to leave the missing values as is.

My final suggestion is to encode missing values as `null` literals, or as "NaN" strings in JSON. And filter out very sparse ones: when a time series is too sparse, it will cause instabilities calculating the conditional probability – a rule of thumb I used is to filter out time series contains more than 20% missing values.

5.2 Train-test split

Schemes for train/test split: I used day-forward chaining. For the test dataset, I reserved a part of data that doesnt have any overlap with the training dataset or the validation dataset (for hyperparameter tuning).

5.3 Improve the model performance

I experimented with the following steps to reduce the models overall errors:

Hyperparameter tuning: It is convenient to use set up a hyperparameter tuning job from the Sagemaker console. To do this, I specified the train data goes to the training channel and validation data goes to the test channel. Then I did a grid search to find the set of hyperparameters that yields the smallest RMSE.

- context_length = 118
- Dropout_rate = 0.052384954005170334
- Learning_rate = 0.0030902721170490166
- Likelihood (only choose from Gaussian, negative-binomial, student's t as they are applicable to the positive valued time series data), student's t yields the best RMSE score.
- Mini_batch_size = 85

- Num_cells = 35
- Num_layers = 2
- epochs = 39

And a side note about the LSTM cells for the built-in algorithm: according to the original paper, the forget bias is set to 1.

Feature extraction: DeepAR supports two types of features: the categorical features that assign time series into different groups; the dynamic features that are covariates time series. The latter does not apply to our case. As the time series characterization provides , it is convenient to directly make use of the characteristics as a way to extract generalizable features, instead of features that are exclusive to specific domain (e.g. the access method for wikipedia pages).

Schemes of adding categorical features

1. Quantiles of scores for the three most predominant characteristics: Day-seasonality, half-day seasonality, trend

Model Cat: this scheme actually increased both overall error and error for each buckets.

2. Quantiles of mean amplitude

Model Cat-1: this scheme helped to reduce the error for day-seasonality time series by a small amount (SMAPE from 17.769% to 17.391%) but increased the error on the trend bucket.

Larger training sets:

Original data: 810 time series, including 14210640 observations

Ext: 3516 time series, including 61684704 observations

Data added: wiki projects from

Ext-1: 9321 time series, including 112402989 observations

Added Year 2017, 2018 per-article pageview. At this stage I started to add Wiki per-article level pageview data from Google BigQuery public tables. The per-article pageview time series has much lower mean amplitude than the project-level data, which is expected. The missing value problem is much severe here, so I loosened the threshold that filter out sparse time series.

Ext-2: 12848 time series, including 143288516 observations

Added Year 2015, 2016 per-article pageview. The time series for these two years (especially 2015 with 50.4% average missing values) are very sparse. The error is increased compared to the previous models with fewer but higher quality data (Ext 1).

Compared all the previous tests, the model Ext-1 is currently the best one (with smallest overall SMAPE).

6 Model robustness

By applying time series characterization methods, I segmented the test datasets into different buckets, so we can evaluate how the model performs across different types of time series, and how the model performs across different scales (heavy traffic, light traffic).

- Please refer to the JupyterNotebook `notebooks/test/final-model-evaluation.ipynb` for detailed discussion with results plot.

7 Usage notes and further suggestions

7.1 Inference considerations

Important parameters for querying the trained model:

- Prediction length = 48 (points)
- Context length = 118 (points)

The trained model will predict the distribution of the next 48 values conditioned on the previous 118 values. The time series for inference should be at least longer than the context length, and it's better to provide the entire time series whenever possible (please see [1] for the best practice part).

The model is trained on hourly frequency data. Ideally the time series to predict are related and have the same granularity as the training data. In cases when we have 5 second frequency data, and we would like to predict the next prediction_length values, a method we can use is to map the 5 second to 1 hour interval to inference, and revert to original 5 second time scale for the predicted values. The reason it could work is that time series are essentially points arranged in certain order, and the RNN/LSTM forecaster is taking in the context points to predict the conditional probability of the future points.

The model outputs the **quantiles** calculated based on the probability distribution parameters. The quantile can be any float number between 0 and 1. But DeepAR computes quantile through Monte Carlo sampling, instead of calculating the integration. We need to keep in mind of the num_samples (could bespecified in `DEEPAR_INFERENCE_CONFIG` or `DeepARPredictor` class if use my `predictor.ipynb` code) that are used to calculate the quantiles: When evaluating with higher quantiles, for example 0.95, it may be important to increase the number of samples. Please see the table in Appendix for further details.

7.2 Proposed next steps

The following steps could be used to improve the model performance. During my test, methods 1 and 2 have shown potentials to reduce the model errors (overall or per-bucket):

1. Increase high-quality training data. Our tests so far have shown that increasing high-quality training data could reduce the model errors, adding sparse data would do the opposite. Currently we have queried the wiki per-article pageview data with LIMIT 5000 and wiki='en', which means the query will return maximum 5000 English article pageview time series. We still have room to largely increase the Wiki data by increasing the limit and search for other languages. When we have much more data, we can set more stringent threshold for the sparseness of the data, for example, missing values $< 10\%$. This is the most doable option.

2. Refine the category feature. I have tested two schemes of injecting features, based on a crude quantile. Given lots of information provided by the time series characterization tool, a refined grouping method can be tested.

3. Use a time series characterization method as a traffic light system: predict the predictables (seasonal or trend time series with medium to large amplitudes), and take extra steps for the rest (such as the Day Impulse pattern that most time series forecasting schemes will fail at, or the incoming time series is the spike type with large standard deviation).

4. New model architecture: possible considerations include a CNN-RNN hybrid model approach [3, 4], Sequence to sequence with Causal CNN [5].

Project flow chart

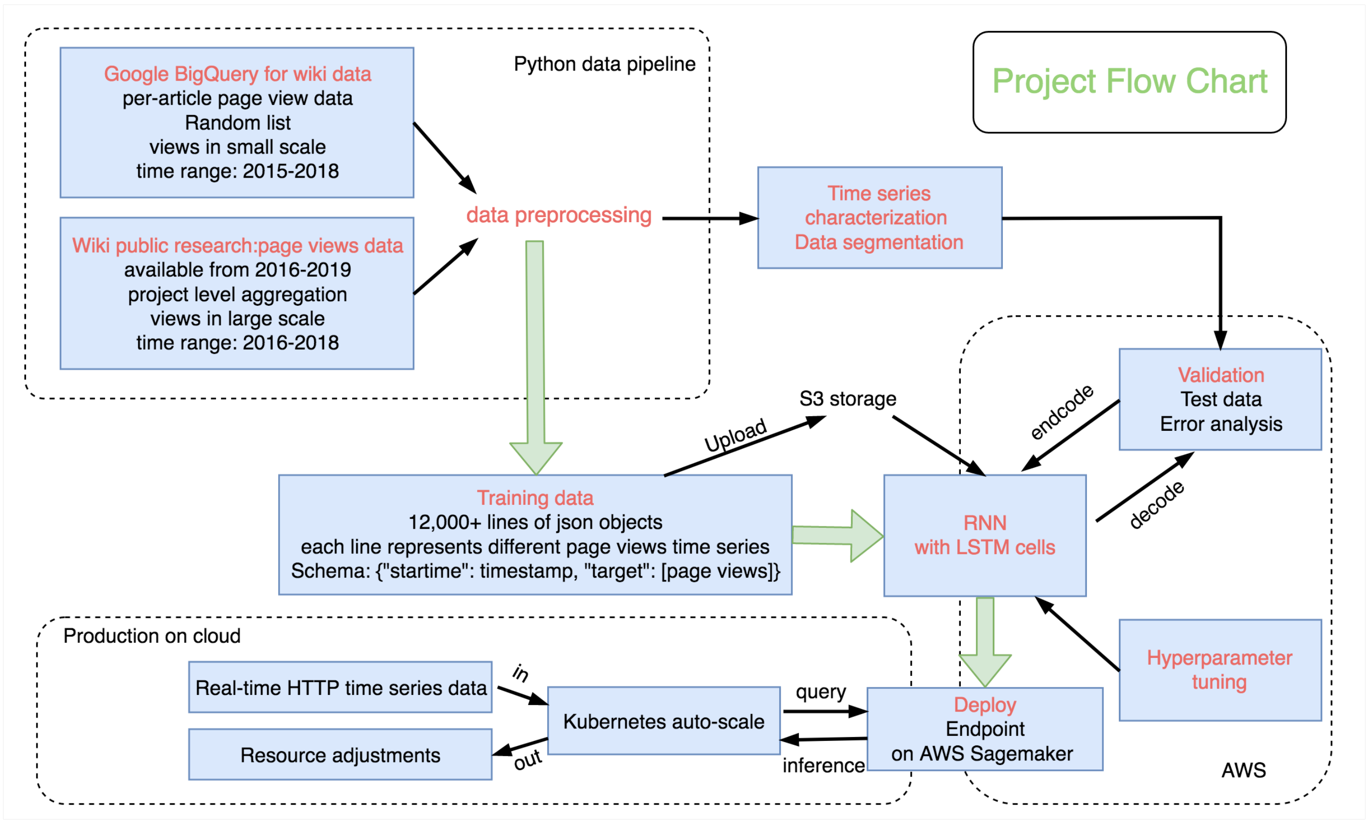


Figure 1: Project flow chart

References

1 AWS DeepAR document, https://docs.aws.amazon.com/sagemaker/latest/dg/deepar_how-it-works.html

2 V. Flunkert et al, DeepAR: Probabilistic Forecasting with Autoregressive Recurrent Networks. arXiv:1704.04110v2, 2017.

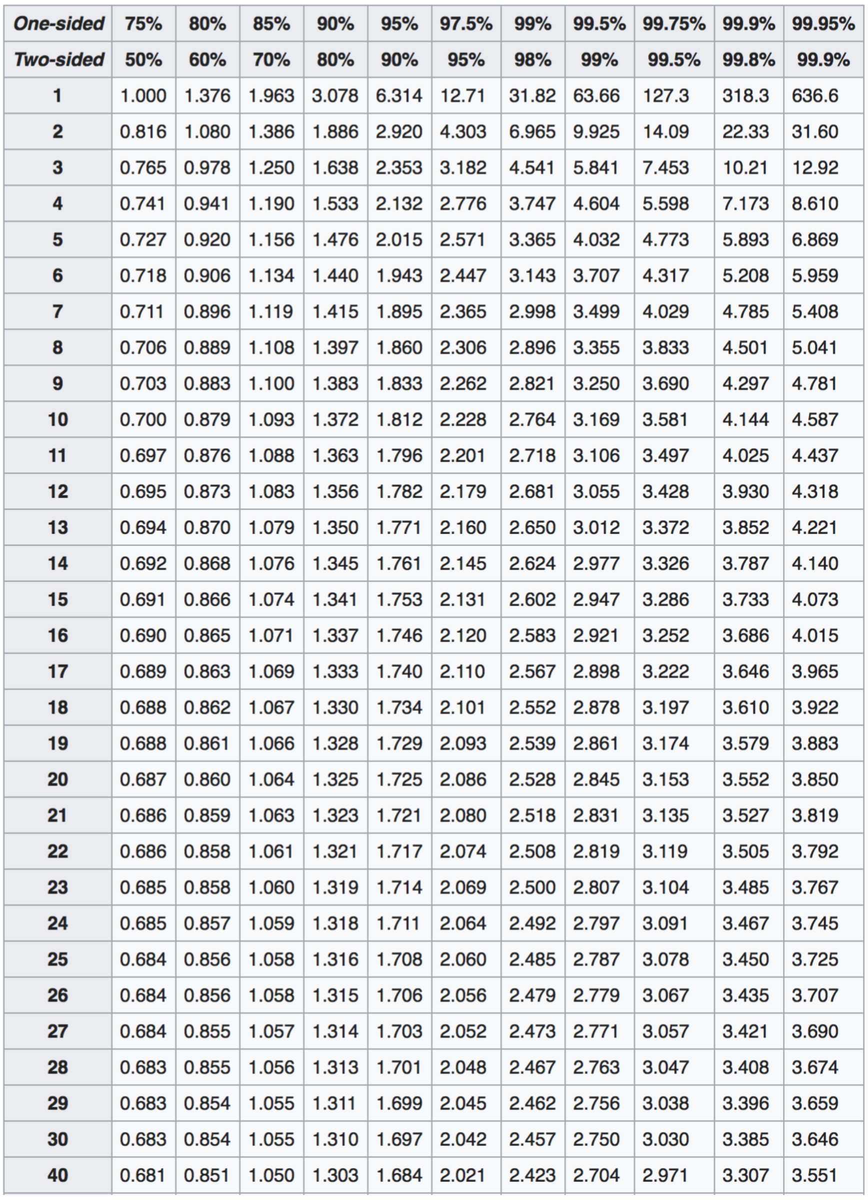
3 W. He, Load Forecasting via Deep Neural Networks. ITQM 2017.

4 R. Cirstea et al, Correlated Time Series Forecasting using Deep Neural Networks: A Summary of Results. arXiv:1808.09794v2, 2018.

5 S. Du et al, Modeling approaches for time series forecasting and anomaly detection. http://cs229.stanford.edu/proj2017/final-reports/5244275.pdf *Note:* This is not a peer-reviewed article but the final SMAPE score for the CNN approach seems an easy win over all other approaches.

Appendix: confidence interval chart for student's t-distribution

Here is a confidence-interval table for student-t distribution. The first column is degrees of freedom ν. Given the same output quantile (one-sided confidence interval), a lower degrees of freedom will make the distribution more "flattened", therefore has higher right-boundary value. We can draw two conclusions here: 1) specify very high output quantile may lead to extremely large values. For example, for $\nu = 1$, quantile = 0.9, right boundary value = 3.078, and quantile = 0.99, right boundary value = 31.82. So if we would like to increase the output quantile to obtain a safer cushion, start with a smaller increasement (such as output quantile $0.9 \rightarrow 0.925$. 2) increase the number of samples as well when we increase the output quantile. As a rule of thumb, if we change the output from 0.9 quantile to 0.95 quantile, triple the num_samples from 100 (default) to 300.



One-sided	75%	80%	85%	90%	95%	97.5%	99%	99.5%	99.75%	99.9%	99.95%
Two-sided	**50%**	**60%**	**70%**	**80%**	**90%**	**95%**	**98%**	**99%**	**99.5%**	**99.8%**	**99.9%**
1	1.000	1.376	1.963	3.078	6.314	12.71	31.82	63.66	127.3	318.3	636.6
2	0.816	1.080	1.386	1.886	2.920	4.303	6.965	9.925	14.09	22.33	31.60
3	0.765	0.978	1.250	1.638	2.353	3.182	4.541	5.841	7.453	10.21	12.92
4	0.741	0.941	1.190	1.533	2.132	2.776	3.747	4.604	5.598	7.173	8.610
5	0.727	0.920	1.156	1.476	2.015	2.571	3.365	4.032	4.773	5.893	6.869
6	0.718	0.906	1.134	1.440	1.943	2.447	3.143	3.707	4.317	5.208	5.959
7	0.711	0.896	1.119	1.415	1.895	2.365	2.998	3.499	4.029	4.785	5.408
8	0.706	0.889	1.108	1.397	1.860	2.306	2.896	3.355	3.833	4.501	5.041
9	0.703	0.883	1.100	1.383	1.833	2.262	2.821	3.250	3.690	4.297	4.781
10	0.700	0.879	1.093	1.372	1.812	2.228	2.764	3.169	3.581	4.144	4.587
11	0.697	0.876	1.088	1.363	1.796	2.201	2.718	3.106	3.497	4.025	4.437
12	0.695	0.873	1.083	1.356	1.782	2.179	2.681	3.055	3.428	3.930	4.318
13	0.694	0.870	1.079	1.350	1.771	2.160	2.650	3.012	3.372	3.852	4.221
14	0.692	0.868	1.076	1.345	1.761	2.145	2.624	2.977	3.326	3.787	4.140
15	0.691	0.866	1.074	1.341	1.753	2.131	2.602	2.947	3.286	3.733	4.073
16	0.690	0.865	1.071	1.337	1.746	2.120	2.583	2.921	3.252	3.686	4.015
17	0.689	0.863	1.069	1.333	1.740	2.110	2.567	2.898	3.222	3.646	3.965
18	0.688	0.862	1.067	1.330	1.734	2.101	2.552	2.878	3.197	3.610	3.922
19	0.688	0.861	1.066	1.328	1.729	2.093	2.539	2.861	3.174	3.579	3.883
20	0.687	0.860	1.064	1.325	1.725	2.086	2.528	2.845	3.153	3.552	3.850
21	0.686	0.859	1.063	1.323	1.721	2.080	2.518	2.831	3.135	3.527	3.819
22	0.686	0.858	1.061	1.321	1.717	2.074	2.508	2.819	3.119	3.505	3.792
23	0.685	0.858	1.060	1.319	1.714	2.069	2.500	2.807	3.104	3.485	3.767
24	0.685	0.857	1.059	1.318	1.711	2.064	2.492	2.797	3.091	3.467	3.745
25	0.684	0.856	1.058	1.316	1.708	2.060	2.485	2.787	3.078	3.450	3.725
26	0.684	0.856	1.058	1.315	1.706	2.056	2.479	2.779	3.067	3.435	3.707
27	0.684	0.855	1.057	1.314	1.703	2.052	2.473	2.771	3.057	3.421	3.690
28	0.683	0.855	1.056	1.313	1.701	2.048	2.467	2.763	3.047	3.408	3.674
29	0.683	0.854	1.055	1.311	1.699	2.045	2.462	2.756	3.038	3.396	3.659
30	0.683	0.854	1.055	1.310	1.697	2.042	2.457	2.750	3.030	3.385	3.646
40	0.681	0.851	1.050	1.303	1.684	2.021	2.423	2.704	2.971	3.307	3.551

Figure 2: Confidence interval chart for student's t-distribution