NEURAL PROPHET

A PREPRINT

Alexey Voskoboinikov
Skolkovo Institute of Science and Technology
Moscow, Russia
dblokv@gmail.com

Polina Pilyugina
Skolkovo Institute of Science and Technology
Moscow, Russia
polina.pilyugina@skoltech.ru

May 11, 2021

ABSTRACT

This project aims to contribute to the open-source library NeuralProphet by adding state-of-the-art models and refactoring the code to adopt the best machine learning engineering practices.

***K*eywords** Time series forecasting · Neural Prophet · Neural forecasting

1 Problem Statement

NeuralProphet is a new library for time series forecasting built on PyTorch. It is inspired by widely known Facebook library for time series forecasting called Prophet ([7]) and DeepAR model ([6]). However, while Prophet is an additive model focused mostly on seasonal components and holiday effects, NeuralProphet additionally includes AutoRegression components. Moreover, NeuralProphet is built on PyTorch, which allows to configure the model more precisely.

In this project we aim to improve existing NeuralProphet library to allow even more possibilities for its users. Currently, forecasting model is written on pure PyTorch, it has complicated structure, code is hardly reusable, and this makes experiments with implementation difficult for outside users. We aim to use PyTorch Lightning framework in order to structure the code in more concise way for future research. Moreover, current implementation of NeuralProphet does not support distributed training, while PyTorch Lightning allows to introduce distributed running of the models. Another problem of current NeuralProphet implementation is that it has a rather specific API in terms of initial data format and outputs. NeuralProphet has specific requirements to the inputs used in the model and its own preprocessing procedure, which is not the same to other models. Moreover, it has several specific modules, which are not implemented in other models out-of-the-box, as explained in 5.1. This makes comparison with other models difficult, as users are required to write additional code, in order to produce comparable results. We aim to introduce state-of-the-art models in accordance with the existing API and model output structure. In particular, we aim to add wrappers for inputs, that will allow all the models to use the same initial inputs, as NeuralProphet. Further, we will rewrite Pytorch Lightning model steps to support NeuralProphet metrics calculation procedure, to ensure compatability.

2 Description of the project

2.1 Main goals

Taking into consideration existing drawbacks, we outlined the main goals of our project as follows:

- Refactor the main model class from NeuralProphet with PyTorch Lightning
- Refactor the rest of the code to support PyTorch Lightning in accordance with existing API
- Adapt and include existing implementations of state-of-the-art models for time series forecasting under the NeuralProphet API
- Add hyperparameter tuning with Ray Tune as additional module to NeuralProphet

- Recreate LIBRA framework for benchmarking in Python and run it on NeuralProphet and our additionally included models
- Add neccessary tests and documentation for introduced functional

2.2 Existing solutions

First part of the project is to structure existing code in accordance with existing PyTorch Lightning framework ([2]). PyTorch lightning is a lightweight PyTorch wrapper that allows to organise the code into separate PyTorch Lighting modules. This framework provides multiple advantages compared to usual PyTorch: models become hardware agnostic and structured, it provides integration with popular machine learning tools, while keeping flexibility of original PyTorch. Among such tools is Ray Tune for hyperparameter tuning.

PyTorch lightning also provides a robust architecture that will help us to implement four state-of-the art models for time series forecasting: N-Beats ([5]), LSTM ([3]), Temporal Fusion Transformers ([4]) and DeepAR ([6]). This will allow users to compare NeuralProphet with these reference models. We rely on existing implementations, available in PyTorch Forecasting library[1].

Another big part of the project is implementation of additional functional to the NeuralProphet model. It will include hyperparameter optimization and modules for evaluation. For hyperparameter optimization, we will use Ray Tune library, as it has hooks to support PyTorch Lightning. Ray Tune has functional for fast distributed hyperparameter tuning, which allows for additional parallelization and scalability. Moreover, PyTorch Lightning has hooks to Ray Tune, which allow for seamless connection between the model and Ray Tune. For benchmarking we will rely on the work on LIBRA framework, described in [1]. LIBRA is an evaluation framework that evaluates and ranks forecasting methods based on their performance.

All these changes to the NeuralProphet should be accompanied by extensive testing and documentation. For tests and integration, NeuralProphet already has modules with unit and integration tests, and we will extend them to cover all of the refactored and new code. As for documentation, we will add necessary documentation for all new functional provided. This will additionally require to add corresponding pages into the doc of NeuralProphet, and add minimal examples as notebooks to the GitHub repository.

3 Main Challenges

3.1 API, architecture and style

One of the main challenges in this project is to maintain NeuralProphet style and API. We should structure the code in a reusable way, so it would be easy for ourselves and other people to contribute to the project in the future. To be more precise, it has inherited from Prophet structure of inputs and outputs. It makes NeuralProphet easily comparable with Prophet, but not with other models. Refactoring in accordance to PyTorch Lightning should not affect existing library consumers. Therefore we will need to introduce additional modules to the TimeNet model used in NeuralProphet. It possesses some challenges, connected to our goal to maintain all existing functionality in place.

And we aim to introduce new models in NeuralProphet in accordance with its existing API to preserve usability. This will require us to understand in detail the explicit structure of inputs and outputs to each model and reformat them accordingly. Additionally, we aim to create models in the same structure, to allow seamless introduction of Ray Tune hyperparameter optimization to all the models.

Existing PyTorch implementations of state-of-the-art models are also available in PyTorch Lightning framework, in PyTorch Forecasting library. However, they do not support NeuralProphet API out of the box, so we will need to refactor them in accordance. In particular, we aim to add metrics calculation and reporting the same way as it is done in NeuralProphet. Also we need to make similar methods and output reporting, in order to allow easy comparison of the results. Additionally, we will refactor existing modules of NeuralProphet to be used in new models.

3.2 Adapting LIBRA

The other challenge - implement LIBRA framework [1] for our purposes. It's implementation is available[2] only in R, so we will have to adapt it in Python as a part of the NeuralProphet. Further, we will run the benchmarking framework on

[1] `https://pytorch-forecasting.readthedocs.io/en/latest/models.html`

[2] `https://github.com/DescartesResearch/ForecastBenchmark`

400 time series from [1], which will require a lot of training time. In order to perform such benchmarking, we will need to test our pipeline and its scalability.

4 The roles for the participants

We distributed our main tasks and goals evenly, as described on the 1. Both of us will work on refactoring into PyTorch Lightning. Alexey will focus on the main TimeNet model class, while Polina will work on the forecaster code. We also distributed addition of models, such that Polina will work on N-Beats and LSTM, while Alexey will work on Temporal Fusion Transformers and DeepAR. We will write corresponding tests and documentation of implemented modules. Further, Polina will focus on hyperparameter tuning addition, while Alexey will implement LIBRA framework in python. Afterwards, we will both work on the benchmarking using LIBRA framework and finalization of the project.

At a current stage, we are finished with code refactoring into Pytorch Lightning. We also added three models: LSTM, NBeats and DeepAR, and created a structure for TemporalFusionTransformer, which will be added further. We also implemented hyperparameter optimization module and will further work on adding support for other models. Currently, it supports LSTM and NeuralProphet.

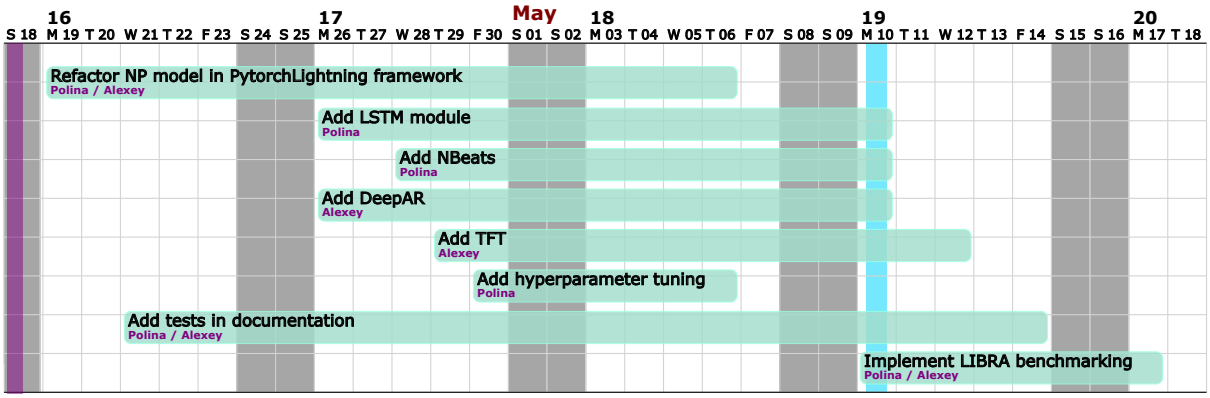

Figure 1: Preliminary Roadmap with Role Distribution

5 Project structure

5.1 Pytorch Lightning introduction to NeuralProphet

Currently, we are finished with refactoring the NeuralProphet code in Pytorch Lightning framework. In particular, we implemented all required modules in TimeNet, which is the base model of NeuralProphet. Additionally, we added necessary hooks to connect the model with all the parts of NeuralProphet. Specifically, we preserved the metrics reporting structure, as well as usage of specific optimizers and schedulers, provided by existing NeuralProphet model. NeuralProphet has its own specific class for metric calculation called Metrics. It consists of several metrics such as MSE and MAE. This class is supplied with predictions and actual values for each batch on each epoch. On epoch end, the metrics is computed. In order to introduce this to models, we changed several modules in Pytorch Lightning implementations: training step, validation step, training epoch end, validation epoch end. Additionally, we implemented module to connect with hyperparameter optimization functionality we added further. It is for internal use in hyperparameter optimization module. It initializes the model and all necessary loaders with provided parameters and outputs it without fitting.

The existing library consists of several modules, from which TimeNet module and forecaster module are the main ones. TimeNet module contains the model itself. We rewrote the model in Pytorch Lightning: we introduced additional required modules, which are requred by Pytorch Lightning. As well as we introduced necessary hooks for the model to preserve the structure of forecaster and its modules. Forecaster module is the main module used by users, with additional preprocessing and initialization functional. It contains the NeuralProphet class which is used by users.

The main methods of NeuralProphet class are:

- init — initializes the class and all the parameters. In case of NeuralProphet, it also initializes metrics structure, which is used further.
- fit — fits the model on the dataset provided. This step consists of preprocessing the data, initializing data loaders, training the model and outputing the training and validation metrics, defined in init module.
- make_future_dataframe — creates a dataframe on which the prediction is done. It contains all the necessary inputs, which the model will require. It support supporting data for prediction on unseen future data, as well as historic prediction.
- predict — predicts the outputs, based on the data, provided in the make_future_dataframe
- plot — plots forecasts in the unified style

On figure 2 is an example of using a NeuralProphet model code structure

```
m = NeuralProphet(
    n_lags=10,
    n_forecasts=3,
    changepoints_range=0.95,
    n_changepoints=30,
    weekly_seasonality=False,
    batch_size=64,
    epochs=10,
    learning_rate=1.0,
)
metrics = m.fit(df, freq='5min')
future = m.make_future_dataframe(df, n_historic_predictions=True)
forecast = m.predict(future)
fig = m.plot(forecast)
```

Figure 2: Example code structure with NeuralProphet model

There are additional methods, but the main changes we made were concerned with these modules. Originally, NeuralProphet included two functions for training_epoch and evaluation_epoch, which we removed and added through Pytorch Lightning into TimeNet. We also moved the main training loop into the Pytorch Lightning Trainer function. Further, it will allow to add new callbacks like EarlyStopping to NeuralProphet. Moreover, we added possibility to train on GPUs through Pytorch Lightning Trainer, which was previously not supported in NeuralProphet at all.

5.2 State-of-the-art models

We have added LSTM, NBeats and DeepAR models in accordance with NeuralProphet API. For LSTM we refactored existing PyTorch implementation in Pytorch lightning framework. We changed training and validation steps to use NeuralProphet Metrics class, as in case of NeuralProphet refactoring. This ensures that LSTM model will provide metrics calculated on each epoch the same way, as NeuralProphet does. For the model class, we used existing NeuralProphet forecaster module and adjusted it to be used with this particular LSTM model. In this case, we made the connection as seamless as possible, with changing all NeuralProphet modules to support new model. Additionally, we added hooks for hyperparameter optimization of LSTM as well. On figure 3 is an example of using a LSTM model code structure

For NBeats we used Pytorch Forecasting NBeats model as a base. We refactored training and validation steps to support the same metrics reporting class as in NeuralProphet. For this we introduced wrappers of predictions, in order for them to be supported by all the metrics inside NeuralProphet metrics class. We implemented additional data preprocessing in order for the model to have the same input as the NeuralProphet. Models from Pytorch Forecasting library use specific TimeSeriesDataset class, so we needed to add wrappers to process NeuralProphet inputs into this class. We used similar initial preprocessing tools for imputing data as are used in NeuralProphet, to induce compatability. Additionally, we reformated the input dataframe in accordance with TimeSeriesDataset class requirements. For this we reformulated the TimeSeriesDataset parameters in accordance with NeuralProphet parameters. Moreover, the specificity of output processed with Pytorch Forecasting models and its prediction functions required us to completely rewrite main existing

```
m = LSTM(
    n_lags=10,
    n_forecasts=3,
    num_hidden_layers=1,
    d_hidden=64,
    learning_rate=0.1,
    epochs=10,
    batch_size=None,
    loss_func="Huber",
    optimizer="AdamW",
    train_speed=None,
    normalize="auto",
    impute_missing=True,
    lstm_bias=True,
    lstm_bidirectional=False,
)
metrics = m.fit(df, freq="5min")
future = m.make_future_dataframe(df, n_historic_predictions=True)
forecast = m.predict(future)
fig = m.plot(forecast)
```

Figure 3: Example code structure with LSTM model

modules of NeuralProphet forecaster class, in order to support new model. In particular, by default predictions are stored as dictionaries of specific sort. Therefore in order to calculate metrics for each epoch, we preprocessed the network outputs to be compatible with NeuralProphet Metrics class. Moreover, the prediction function by itself outputs raw results, which we needed to further process and wrap into the same dataset structure, as in case of NeuralProphet. Therefore, we basically introduced a new forecaster holder for the NBeats model, which has the same structure of modules and produces the same outputs as NeuralProphet. It also follows the same logic, as in NeuralProphet: it contains initialization of the model, creation of dataset modules. We also refactored existing modules for data preprocessing and parameter selection, in order to work with new models and their parameters. Additionally, Pytorch Forecasting allows to define hyperparameters from the dataset, and we added this functional as well, by introducing a hyperparameter from_dataset. If this hyperparameter is set to True, the model parameters will be set automatically, which is particulary useful for new users of NBeats. However, we reserved the possibility to change main NBeats parameters manually, based on the users' desire. On figure 4 is an example of using a NBeats model code structure

```
m = NBeatsNP(
        max_encoder_length = 150,
        batch_size = None,
        epochs = 10,
        num_gpus = 0,
        patience_early_stopping = 10,
        early_stop = True,
        weight_decay=1e-2,
        learning_rate=3e-2,
        auto_lr_find=False,
        num_workers=3,
        from_dataset=True,
    )
metrics = m.fit(df, freq='5min')
future = m.make_future_dataframe(df, n_historic_predictions=True)
forecast = m.predict(future)
fig = m.plot(forecast)
```

Figure 4: Example code structure with NBeats model

DeepAR is a probabilistic forecasting model, and Pytorch Forecasting provides a good base implementation. This model uses Normal Distribution Loss (NDL) as a default loss function, and data has to be normalised accordingly

(NDL does not work with all-positive target values). As probabilistic model, for NDL it predicts 2 values - "loc" and "scale", and training network with this loss function also decreases other metrics, as shown in the example notebook. We override basic Pytorch Lightning methods (test/validation steps and methods for epoch end), and then wrapped this model to make it work with NeuralProphet datasets and produce metrics and predictions in the same way.

```
deepar = DeepAR(
        context_length=60,
        prediction_length=20,
        batch_size = 32,
        epochs = 10,
        num_gpus = 0,
        patience_early_stopping = 10,
        early_stop = True,
        learning_rate=3e-4,
        auto_lr_find=True,
        num_workers=3,
        loss_func = 'normaldistributionloss',
        hidden_size=10,
        rnn_layers=2,
        dropout=0.1,
)
metrics_df = deepar.fit(df, freq = freq)
future = deepar.make_future_dataframe(df, freq,
                                      periods=10,
                                      n_historic_predictions=10)
forecast = deepar.predict(future)
f = deepar.plot(forecast)
```

Figure 5: Example code structure with DeepAR model

In general, all the models we introduced have the same main modules, as NeuralProphet, and output the results in the same format. This allows for a fast comparison of training and validation metrics, as well as plotting of the results with existing NeuralProphet functional. In the future, it is possible to refactor it even further and make implementation easier to read and maintain. We created example notebooks for each of the model, in order to provide users with an introduction to how these models work.

On figure 6 we provide with example predictions plotting for three models: NeuralProphet, LSTM, NBeats and DeepAR, on a sample Yosemite dataset. These baseline results are contained in example notebooks of each model.

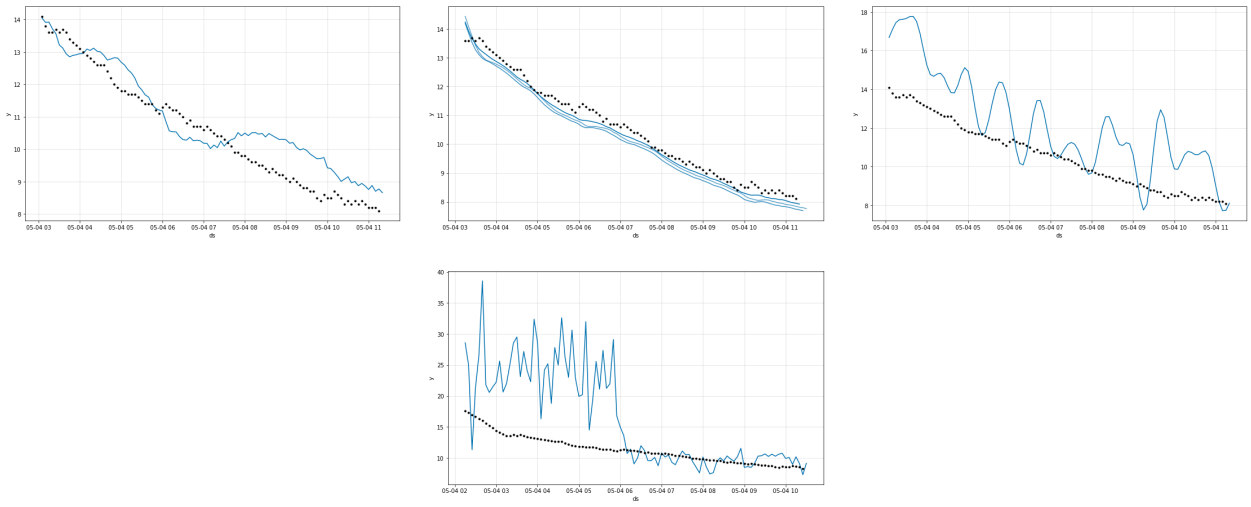

Figure 6: Example of predictions plotting for models (from left to right): NeuralProphet, LSTM, NBeats and DeepAR at the bottom

5.3 Hyperparameter optimization

As for hyperparameter tuning, we created a separate module in NeuralProphet library for this sake. We added hooks to each of the models we implemented and NeuralProphet itself, using the functionality provided by Pytorch Lightning, to fastly connect the optimization module with the main model modules. Moreover, our hyperparameter optimization works not only with NeuralProphet, but for all the models we implemented. Currently, the function works in two modes: auto and manual. The automatic parameter selection uses predefined by us sets of hyperparameters to tune over. The manual mode requires user to provide their own configuration, in accordance with tune API. We have added a notebook example of how this can be done, so the users will have an idea on how to use Ray Tune functionality as well. This function is called tune_hyperparameters and it out-of-the-box requires only three parameters: the model name, the dataframe with data, and frequency of dates in data. Its default set up can be useful for fast basic tuning and also for new users, which have no experience with the model. We evaluated the tuning progress on the Yosemite dataset, included in NeuralProphet. We compared the default one-step-ahead forecasting initialization of NeuralProphet with its tuned version, and it showed drastical improvement. Therefore, the proposed functional can be exceptionally useful for new users, as it includes automatic mode which requires no initial understanding of the model parameters and provides reasonable improvement out-of-the-box. On figure 7 we provide a resulting metrics of two NP models: with default parameters, and with tuned parameters

SmoothL1Loss	0.005514	SmoothL1Loss	0.000469
MAE	3.886362	MAE	1.086094
MSE	29.934882	MSE	3.560099
RegLoss	0.000000	RegLoss	0.000108
SmoothL1Loss_val	0.170988	SmoothL1Loss_val	0.001177
MAE_val	27.786178	MAE_val	1.659725
MSE_val	928.262373	MSE_val	8.928988

Figure 7: Left is metrics of default model, and right is metrics on tuned model

Ray Tune additionally provides a functionality to distribute resources among trials of experiments on both CPUs and GPUs, which allows for fast training. We used AsyncHyperBandScheduler for hyperparameter tuning. This scheduler is an improved version of HyperBand scheduler. The idea behind it, is that it early stops low-performing trials using the HyperBand optimization algorithm. This allows not to waste computational resources on low-performing hyperparameter settings, which also induces faster optimization.

6 Next steps

As for the next steps, we aim to finalize addition of TemporalFusionTransformer implementation. Currently we have a holder structure for it, but because of the model complexity we need some additional changes to be made. Further, we will create the LIBRA framework for evaluation and run the benchmarking of all models. As a part of it, we will add datasets from benchmarking to example datasets in the NeuralProphet repo. We will add additionally a modul with processing the datasets and running all the models on them. Also we will introduce the similar metrics structure as was proposed by the paper, and compare the results. Additionally, we will add support of all models in hyperparameter tuning module.

During the whole process of code writing we included documentation to all the models we introduced. We will summarize it in additional pages of doc in GitHub repository. Moreover, we have already added notebooks with examples of usage of additional modules we provide. As a final step we will add this examples to the test section, which will be run at pre-commit stage.

7 A link to the GitHub repository

For this project we forked the original repository and will contribute to it. It will allows us to create a pull request into the master in the future. GitHub repository is available through `https://github.com/adasegroup/neural_prophet/tree/master`. We have preserved the initial GitHub repository structure. On figure 8 we provide an overview of GitHub repository structure. We outlined the files that were added or changed on the course of this project. This includes: refactoring of existing main modules, three additional modules and example notebooks.

```
└── neural_prophet
    ├── LICENSE
    ├── MANIFEST.in
    ├── README.md
    ├── docs
    ├── example_data
    ├── example_notebooks
    │   ├── LSTM_example.ipynb
    │   ├── NBeats_example.ipynb
    │   ├── DeepAR_example.ipynb
    │   ├── hyperparameter_example.ipynb
    ├── mkdocs.yml
    ├── neuralprophet
    │   ├── __init__.py
    │   ├── additional_models.py
    │   ├── configure.py
    │   ├── forecaster.py
    │   ├── forecaster_additional_models.py
    │   ├── hyperparameter_tuner.py
    │   ├── time_net.py
    ├── notes
    ├── peer_reviews
    ├── pyproject.toml
    ├── requirements.txt
    ├── scripts
    ├── setup.py
    └── tests
```

Figure 8: Structure of GitHub repository with outlining the files added or changed during the project, as for the current state

References

[1] André Bauer et al. "Libra : A Benchmark for Time Series Forecasting Methods Libra : A Benchmark for Time Series Forecasting Methods". In: April (2021).

[2] WA Falcon and .al. "PyTorch Lightning". In: *GitHub. Note: https://github.com/PyTorchLightning/pytorch-lightning* 3 (2019).

[3] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (1997), pp. 1735–1780. ISSN: 08997667. DOI: 10.1162/neco.1997.9.8.1735.

[4] Bryan Lim et al. "Temporal fusion transformers for interpretable multi-horizon time series forecasting". In: *arXiv* Bryan Lim (2019), pp. 1–27. arXiv: 1912.09363.

[5] Boris N. Oreshkin et al. "N-BEATS: Neural basis expansion analysis for interpretable time series forecasting". In: *arXiv* (2019), pp. 1–31. ISSN: 23318422. arXiv: 1905.10437.

[6] David Salinas et al. "DeepAR: Probabilistic forecasting with autoregressive recurrent networks". In: *International Journal of Forecasting* 36.3 (2020), pp. 1181–1191. ISSN: 01692070. DOI: 10.1016/j.ijforecast.2019.07.001. arXiv: 1704.04110. URL: https://doi.org/10.1016/j.ijforecast.2019.07.001.

[7] Sean J. Taylor and Benjamin Letham. "Forecasting at Scale". In: *American Statistician* 72.1 (2018), pp. 37–45. ISSN: 15372731. DOI: 10.1080/00031305.2017.1380080.