Bayesian Estimation and Forecasting of Time Series in statsmodels

Chad Fulton[‡*]

Abstract—`Statsmodels`, a Python library for statistical and econometric analysis, has traditionally focused on frequentist inference, including in its models for time series data. This paper introduces the powerful features for Bayesian inference of time series models that exist in `statsmodels`, with applications to model fitting, forecasting, time series decomposition, data simulation, and impulse response functions.

Index Terms—time series, forecasting, bayesian inference, Markov chain Monte Carlo, statsmodels

Introduction

`Statsmodels` [SP10] is a well-established Python library for statistical and econometric analysis, with support for a wide range of important model classes, including linear regression, ANOVA, generalized linear models (GLM), generalized additive models (GAM), mixed effects models, and time series models, among many others. In most cases, model fitting proceeds by using frequentist inference, such as maximum likelihood estimation (MLE). In this paper, we focus on the class of time series models [MPS11], support for which has grown substantially in `statsmodels` over the last decade. After introducing several of the most important new model classes – which are by default fitted using MLE – and their features – which include forecasting, time series decomposition and seasonal adjustment, data simulation, and impulse response analysis – we describe the powerful functions that enable users to apply Bayesian methods to a wide range of time series models.

Support for Bayesian inference in Python outside of `statsmodels` has also grown tremendously, particularly in the realm of probabilistic programming, and includes powerful libraries such as `PyMC3` [SWF16], `PyStan` [CGH+17], and `TensorFlow Probability` [DLT+17]. Meanwhile, `ArviZ` [KCHM19] provides many excellent tools for associated diagnostics and vizualisations. The aim of these libraries is to provide support for Bayesian analysis of a large class of models, and they make available both advanced techniques, including autotuning algorithms, and flexible model specification. By contrast, here we focus on simpler techniques. However, while the libraries above do include some support for time series models, this has not been their primary focus. As a result, introducing Bayesian inference for the well-developed stable of time series models in `statsmodels`, and providing access to the rich associated feature set already mentioned, presents a complementary option to these more general-purpose libraries.[1]

* *Corresponding author: chad.t.fulton@frb.gov*
‡ *Federal Reserve Board of Governors*

Time series analysis in `statsmodels`

A time series is a sequence of observations ordered in time, and time series data appear commonly in statistics, economics, finance, climate science, control systems, and signal processing, among many other fields. One distinguishing characteristic of many time series is that observations that are close in time tend to be more correlated, a feature known as autocorrelation. While successful analyses of time series data must account for this, statistical models can harness it to decompose a time series into trend, seasonal, and cyclical components, produce forecasts of future data, and study the propagation of shocks over time.

We now briefly review the models for time series data that are available in `statsmodels` and describe their features.[2]

Exponential smoothing models

Exponential smoothing models are constructed by combining one or more simple equations that each describe some aspect of the evolution of univariate time series data. While originally somewhat *ad hoc*, these models can be defined in terms of a proper statistical model (for example, see [HKOS08]). They have enjoyed considerable popularity in forecasting (for example, see the implementation in R described by [HA18]). A prototypical example that allows for trending data and a seasonal component – often known as the additive "Holt-Winters' method" – can be written as

$$
\begin{aligned}
l_t &= \alpha(y_t - s_{t-m}) + (1-\alpha)(l_{t-1} + b_{t-1}) \\
b_t &= \beta(l_t - l_{t-1}) + (1-\beta)b_{t-1} \\
s_t &= \gamma(y_t - l_{t-1} - b_{t-1}) + (1-\gamma)s_{t-m}
\end{aligned}
$$

where l_t is the level of the series, b_t is the trend, s_t is the seasonal component of period m, and α, β, γ are parameters of the model. When augmented with an error term with some given probability distribution (usually Gaussian), likelihood-based inference can be used to estimate the parameters. In `statsmodels`,

1. In addition, it is possible to combine the sampling algorithms of PyMC3 with the time series models of `statsmodels`, although we will not discuss this approach in detail here. See, for example, https://www.statsmodels.org/v0.13.0/examples/notebooks/generated/statespace_sarimax_pymc3.html.

2. In addition to statistical models, `statsmodels` also provides a number of tools for exploratory data analysis, diagnostics, and hypothesis testing related to time series data; see https://www.statsmodels.org/stable/tsa.html.

additive exponential smoothing models can be constructed using the `statespace.ExponentialSmoothing` class.[3] The following code shows how to apply the additive Holt-Winters model above to model quarterly data on consumer prices:

```
import statsmodels.api as sm
# Load data
mdata = sm.datasets.macrodata.load().data
# Compute annualized consumer price inflation
y = np.log(mdata['cpi']).diff().iloc[1:] * 400

# Construct the Holt-Winters model
model_hw = sm.tsa.statespace.ExponentialSmoothing(
    y, trend=True, seasonal=12)
```

Structural time series models

Structural time series models, introduced by [Har90] and also sometimes known as unobserved components models, similarly decompose a univariate time series into trend, seasonal, cyclical, and irregular components:

$$y_t = \mu_t + \gamma_t + c_t + \varepsilon_t$$

where μ_t is the trend, γ_t is the seasonal component, c_t is the cyclical component, and $\varepsilon_t \sim N(0, \sigma^2)$ is the error term. However, this equation can be augmented in many ways, for example to include explanatory variables or an autoregressive component. In addition, there are many possible specifications for the trend, seasonal, and cyclical components, so that a wide variety of time series characteristics can be accommodated. In `statsmodels`, these models can be constructed from the `UnobservedComponents` class; a few examples are given in the following code:

```
# "Local level" model
model_ll = sm.tsa.UnobservedComponents(y, 'llevel')
# "Local linear trend", with seasonal component
model_arma11 = sm.tsa.UnobservedComponents(
    y, 'lltrend', seasonal=4)
```

These models have become popular for time series analysis and forecasting, as they are flexible and the estimated components are intuitive. Indeed, Google's Causal Impact library [BGK+15] uses a Bayesian structural time series approach directly, and Facebook's Prophet library [TL17] uses a conceptually similar framework and is estimated using PyStan.

Autoregressive moving-average models

Autoregressive moving-average (ARMA) models, ubiquitous in time series applications, are well-supported in `statsmodels`, including their generalizations, abbreviated as "SARIMAX", that allow for integrated time series data, explanatory variables, and seasonal effects.[4] A general version of this model, excluding integration, can be written as

$$\begin{aligned} y_t &= x_t\beta + \xi_t \\ \xi_t &= \phi_1\xi_{t-1} + \cdots + \phi_p\xi_{t-p} + \varepsilon_t + \theta_1\varepsilon_{t-1} + \cdots + \theta_q\varepsilon_{t-q} \end{aligned}$$

where $\varepsilon_t \sim N(0, \sigma^2)$. These are constructed in `statsmodels` with the `ARIMA` class; the following code shows how to construct a variety of autoregressive moving-average models for consumer price data:

```
# AR(2) model
model_ar2 = sm.tsa.ARIMA(y, order=(2, 0, 0))

# ARMA(1, 1) model with explanatory variable
X = mdata['realint']
model_arma11 = sm.tsa.ARIMA(
    y, order=(1, 0, 1), exog=X)
# SARIMAX(p, d, q)x(P, D, Q, s) model
model_sarimax = sm.tsa.ARIMA(
    y, order=(p, d, q), seasonal_order=(P, D, Q, s))
```

While this class of models often produces highly competitive forecasts, it does not produce a decomposition of a time series into, for example, trend and seasonal components.

Vector autoregressive models

While the SARIMAX models above handle univariate series, `statsmodels` also has support for the multivariate generalization to vector autoregressive (VAR) models.[5] These models are written

$$y_t = \nu + \Phi_1 y_{t-1} + \cdots + \Phi_p y_{t-p} + \varepsilon_t$$

where y_t is now considered as an $m \times 1$ vector. As a result, the intercept ν is also an $m \times 1$ vector, the coefficients Φ_i are each $m \times m$ matrices, and the error term is $\varepsilon_t \sim N(0_m, \Omega)$, with Ω an $m \times m$ matrix. These models can be constructed in `statsmodels` using the `VARMAX` class, as follows[6]

```
# Multivariate dataset
z = (np.log(mdata['realgdp', 'realcons', 'cpi'])
       .diff().iloc[1:])

# VAR(1) model
model_var = sm.tsa.VARMAX(z, order=(1, 0))
```

Dynamic factor models

`statsmodels` also supports a second model for multivariate time series: the dynamic factor model (DFM). These models, often used for dimension reduction, posit a few unobserved factors, with autoregressive dynamics, that are used to explain the variation in the observed dataset. In `statsmodels`, there are two model classes, `DynamicFactor`` and `DynamicFactorMQ`, that can fit versions of the DFM. Here we focus on the `DynamicFactor` class, for which the model can be written

$$\begin{aligned} y_t &= \Lambda f_t + \varepsilon_t \\ f_t &= \Phi_1 f_{t-1} + \cdots + \Phi_p f_{t-p} + \eta_t \end{aligned}$$

Here again, the observation is assumed to be $m \times 1$, but the factors are $k \times 1$, where it is possible that $k << m$. As before, we assume conformable coefficient matrices and Gaussian errors.

The following code shows how to construct a DFM in `statsmodels`

```
# DFM with 2 factors that evolve as a VAR(3)
model_dfm = sm.tsa.DynamicFactor(
    z, k_factors=2, factor_order=3)
```

Linear Gaussian state space models

In `statsmodels`, each of the model classes introduced above (`statespace.ExponentialSmoothing`, `UnobservedComponents`, `ARIMA`, `VARMAX`,

3. A second class, `ETSModel`, can also be used for both additive and multiplicative models, and can exhibit superior performance with maximum likelihood estimation. However, it lacks some of the features relevant for Bayesian inference discussed in this paper.

4. Note that in `statsmodels`, models with explanatory variables are in the form of "regression with SARIMA errors".

5. `statsmodels` also supports vector moving-average (VMA) models using the same model class as described here for the VAR case, but, for brevity, we do not explicitly discuss them here.

6. A second class, `VAR`, can also be used to fit VAR models, using least squares. However, it lacks some of the features relevant for Bayesian inference discussed in this paper.

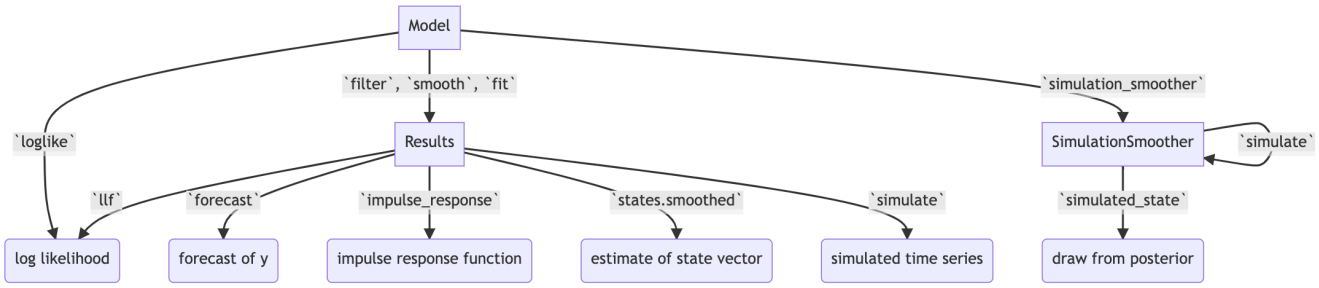

Fig. 1: *Selected functionality of state space models in statsmodels.*

`DynamicFactor`, and `DynamicFactorMQ`) are implemented as part of a broader class of models, referred to as linear Gaussian state space models (hereafter for brevity, simply "state space models" or SSM). This class of models can be written as

$$y_t = d_t + Z_t \alpha_t + \varepsilon_t \qquad \varepsilon_t \sim N(0, H_t)$$
$$\alpha_{t+1} = c_t + T_t \alpha_t + R_t \eta_t \qquad \eta_t \sim N(0, Q_t)$$

where α_t represents an unobserved vector containing the "state" of the dynamic system. In general, the model is multivariate, with y_t and ε_t $m \times 1$ vector, α_t $k \times 1$, and η_t *r times 1*.

Powerful tools exist for state space models to estimate the values of the unobserved state vector, compute the value of the likelihood function for frequentist inference, and perform posterior sampling for Bayesian inference. These tools include the celebrated Kalman filter and smoother and a simulation smoother, all of which are important for conducting Bayesian inference for these models.[7] The implementation in `statsmodels` largely follows the treatment in [DK12], and is described in more detail in [Ful15].

In addition to these key tools, state space models also admit general implementations of useful features such as forecasting, data simulation, time series decomposition, and impulse response analysis. As a consequence, each of these features extends to each of the time series models described above. Figure 1 presents a diagram showing how to produce these features, and the code below briefly introduces a subset of them.

```
# Construct the Model
model_ll = sm.tsa.UnobservedComponents(y, 'llevel')

# Construct a simulation smoother
sim_ll = model_ll.simulation_smoother()

# Parameter values (variance of error and
# variance of level innovation, respectively)
params = [4, 0.75]

# Compute the log-likelihood of these parameters
llf = model_ll.loglike(params)

# `smooth` applies the Kalman filter and smoother
# with a given set of parameters and returns a
# Results object
results_ll = model_ll.smooth(params)

# Produce forecasts for the next 4 periods
fcast = results_ll.forecast(4)

# Produce a draw from the posterior distribution
# of the state vector
sim_ll.simulate()
draw = sim_ll.simulated_state
```

Nearly identical code could be used for any of the model classes introduced above, since they are all implemented as part of the same state space model framework. In the next section, we show how these features can be used to perform Bayesian inference with these models.

Bayesian inference via Markov chain Monte Carlo

We begin by giving a cursory overview of the key elements of Bayesian inference required for our purposes here.[8] In brief, the Bayesian approach stems from Bayes' theorem, in which the posterior distribution for an object of interest is derived as proportional to the combination of a prior distribution and the likelihood function

$$\underbrace{p(A|B)}_{\text{posterior}} \propto \underbrace{p(B|A)}_{\text{likelihood}} \times \underbrace{p(A)}_{\text{prior}}$$

Here, we will be interested in the posterior distribution of the parameters of our model and of the unobserved states, conditional on the chosen model specification and the observed time series data. While in most cases the form of the posterior cannot be derived analytically, simulation-based methods such as Markov chain Monte Carlo (MCMC) can be used to draw samples that approximate the posterior distribution nonetheless. While PyMC3, PyStan, and TensorFlow Probability emphasize Hamiltonian Monte Carlo (HMC) and no-U-turn sampling (NUTS) MCMC methods, we focus on the simpler random walk Metropolis-Hastings (MH) and Gibbs sampling (GS) methods. These are standard MCMC methods that have enjoyed great success in time series applications and which are simple to implement, given the state space framework already available in `statsmodels`. In addition, the ArviZ library is designed to work with MCMC output from any source, and we can easily adapt it to our use.

With either Metropolis-Hastings or Gibbs sampling, our procedure will produce a sequence of sample values (of parameters and / or the unobserved state vector) that approximate draws from the posterior distribution arbitrarily well, as the number of length

7. `Statsmodels` currently contains two implementations of simulation smoothers for the linear Gaussian state space model. The default is the "mean correction" simulation smoother of [DK02]. The precision-based simulation smoother of [CJ09] can alternatively be used by specifying `method='cfa'` when creating the simulation smoother object.

of the chain of samples becomes very large.

Random walk Metropolis-Hastings

In random walk Metropolis-Hastings (MH), we begin with an arbitrary point as the initial sample, and then iteratively construct new samples in the chain as follows. At each iteration, (a) construct a proposal by perturbing the previous sample by a Gaussian random variable, and then (b) accept the proposal with some probability. If a proposal is accepted, it becomes the next sample in the chain, while if it is rejected then the previous sample value is carried over. Here, we show how to implement Metropolis-Hastings estimation of the variance parameter in a simple model, which only requires the use of the log-likelihood computation introduced above.

```
import arviz as az
from scipy import stats

# Construct the model
model_rw = sm.tsa.UnobservedComponents(y, 'rwalk')

# Specify the prior distribution. With MH, this
# can be freely chosen by the user
prior = stats.uniform(0.0001, 100)

# Specify the Gaussian perturbation distribution
perturb = stats.norm(scale=0.1)

# Storage
niter = 100000
samples_rw = np.zeros(niter + 1)

# Initialization
samples_rw[0] = y.diff().var()
llf = model_rw.loglike(samples_rw[0])
prior_llf = prior.logpdf(samples_rw[0])

# Iterations
for i in range(1, niter + 1):
    # Compute the proposal value
    proposal = samples_rw[i - 1] + perturb.rvs()

    # Compute the acceptance probability
    proposal_llf = model_rw.loglike(proposal)
    proposal_prior_llf = prior.logpdf(proposal)
    accept_prob = np.exp(
        proposal_llf - llf
        + prior_llf - proposal_prior_llf)

    # Accept or reject the value
    if accept_prob > stats.uniform.rvs():
        samples_rw[i] = proposal
        llf = proposal_llf
        prior_llf = proposal_prior_llf
    else:
        samples_rw[i] = samples_rw[i - 1]

# Convert for use with ArviZ and plot posterior
samples_rw = az.convert_to_inference_data(
    samples_rw)
# Eliminate the first 10000 samples as burn-in;
# thin by factor of 10 to reduce autocorrelation
az.plot_posterior(samples_rw.posterior.sel(
    {'draw': np.s_[10000::10]}), kind='bin',
    point_estimate='median')
```

The approximate posterior distribution, constructed from the sample chain, is shown in Figure 2.

8. While a detailed description of these issues is out of the scope of this paper, there are many superb references on this topic. We refer the interested reader to [WH99], which provides a book-length treatment of Bayesian inference for state space models, and [KN99], which provides many examples and applications.

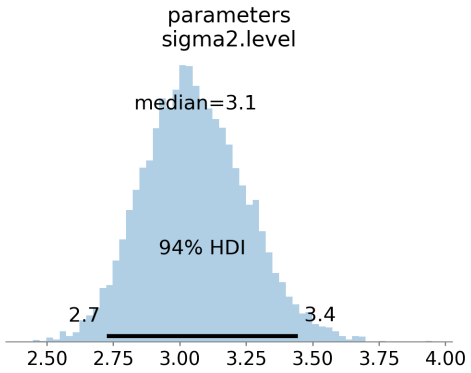

Fig. 2: *Approximate posterior distribution of variance parameter, random walk model, Metropolis-Hastings; U.S. Industrial Production.*

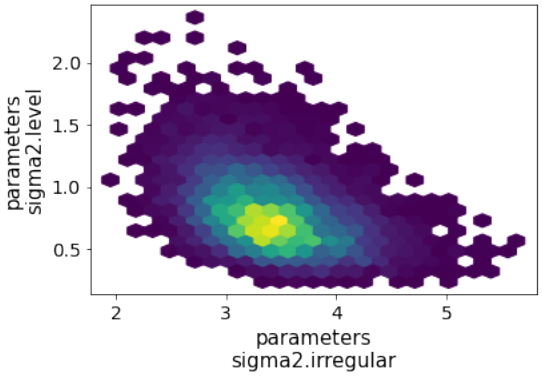

Fig. 3: *Approximate posterior joint distribution of variance parameters, local level model, Gibbs sampling; CPI inflation.*

Gibbs sampling

Gibbs sampling (GS) is a special case of Metropolis-Hastings (MH) that is applicable when it is possible to produce draws directly from the conditional distributions of every variable, even though it is still not possible to derive the general form of the joint posterior. While this approach can be superior to random walk MH when it is applicable, the ability to derive the conditional distributions typically requires the use of a "conjugate" prior – i.e., a prior from some specific family of distributions. For example, above we specified a uniform distribution as the prior when sampling via MH, but that is not possible with Gibbs sampling. Here, we show how to implement Gibbs sampling estimation of the variance parameter, now making use of an inverse Gamma prior, and the simulation smoother introduced above.

```
# Construct the model and simulation smoother
model_ll = sm.tsa.UnobservedComponents(y, 'llevel')
sim_ll = model_ll.simulation_smoother()

# Specify the prior distributions. With GS, we must
# choose an inverse Gamma prior for each variance
priors = [stats.invgamma(0.01, scale=0.01)] * 2

# Storage
niter = 100000
samples_ll = np.zeros((niter + 1, 2))

# Initialization
samples_ll[0] = [y.diff().var(), 1e-5]

# Iterations
```

```
for i in range(1, niter + 1):
    # (a) Update the model parameters
    model_ll.update(samples_ll[i - 1])

    # (b) Draw from the conditional posterior of
    # the state vector
    sim_ll.simulate()
    sample_state = sim_ll.simulated_state.T

    # (c) Compute / draw from conditional posterior
    # of the parameters:
    # ...observation error variance
    resid = y - sample_state[:, 0]
    post_shape = len(resid) / 2 + 0.01
    post_scale = np.sum(resid**2) / 2 + 0.01
    samples_ll[i, 0] = stats.invgamma(
        post_shape, scale=post_scale).rvs()

    # ...level error variance
    resid = sample_state[1:] - sample_state[:-1]
    post_shape = len(resid) / 2 + 0.01
    post_scale = np.sum(resid**2) / 2 + 0.01
    samples_ll[i, 1] = stats.invgamma(
        post_shape, scale=post_scale).rvs()

# Convert for use with ArviZ and plot posterior
samples_ll = az.convert_to_inference_data(
    {'parameters': samples_ll[None, ...]},
    coords={'parameter': model_ll.param_names},
    dims={'parameters': ['parameter']})
az.plot_pair(samples_ll.posterior.sel(
    {'draw': np.s_[10000::10]}), kind='hexbin');
```

The approximate posterior distribution, constructed from the sample chain, is shown in Figure 3.

Illustrative examples

For clarity and brevity, the examples in the previous section gave results for simple cases. However, these basic methods carry through to each of the models introduced earlier, including in cases with multivariate data and hundreds of parameters. Moreover, the Metropolis-Hastings approach can be combined with the Gibbs sampling approach, so that if the end user wishes to use Gibbs sampling for some parameters, they are not restricted to choose only conjugate priors for all parameters.

In addition to sampling the posterior distributions of the parameters, this method allows sampling other objects of interest, including forecasts of observed variables, impulse response functions, and the unobserved state vector. This last possibility is especially useful in cases such as the structural time series model, in which the unobserved states correspond to interpretable elements such as the trend and seasonal components. We provide several illustrative examples of the various types of analysis that are possible.

Forecasting and Time Series Decomposition

In our first example, we apply the Gibbs sampling approach to a structural time series model in order to forecast U.S. Industrial Production and to produce a decomposition of the series into level, trend, and seasonal components. The model is

$$
\begin{aligned}
y_t &= \mu_t + \gamma_t + \varepsilon_t && \text{observation equation} \\
\mu_t &= \beta_t + \mu_{t-1} + \zeta_t && \text{level} \\
\beta_t &= \beta_{t-1} + \xi_t && \text{trend} \\
\gamma_t &= \gamma_{t-s} + \eta_t && \text{seasonal}
\end{aligned}
$$

Here, we set the seasonal periodicity to *s=12*, since Industrial Production is a monthly variable. We can construct this model in Statsmodels as[9]

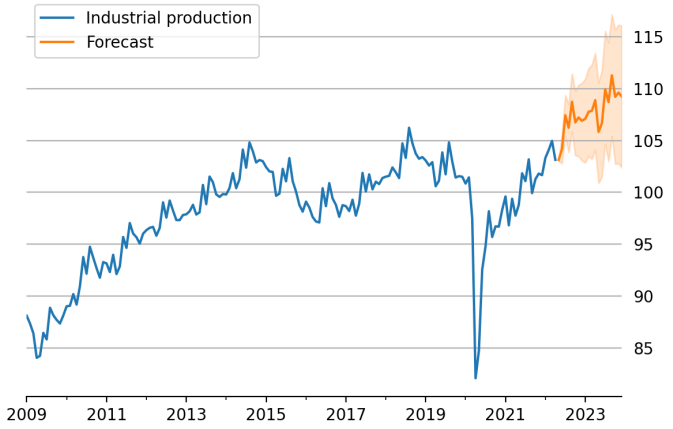

Fig. 4: *Data and forecast with 80% credible interval; U.S. Industrial Production.*

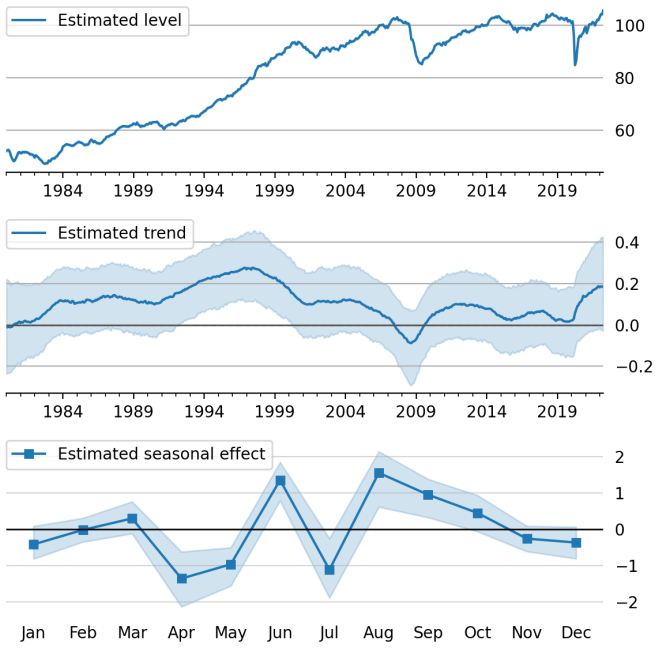

Fig. 5: *Estimated level, trend, and seasonal components, with 80% credible interval; U.S. Industrial Production.*

```
model = sm.tsa.UnobservedComponents(
    y, 'lltrend', seasonal=12)
```

To produce the time-series decomposition into level, trend, and seasonal components, we will use samples from the posterior of the state vector $(\mu_t, \beta_t, \gamma_t)$ for each time period t. These are immediately available when using the Gibbs sampling approach; in the earlier example, the draw at each iteration was assigned to the variable *sample_state*. To produce forecasts, we need to draw from the posterior predictive distribution for horizons $h = 1, 2, \ldots H$. This can be easily accomplished by using the *simulate* method introduced earlier. To be concrete, we can accomplish these tasks by modifying section (b) of our Gibbs sampler iterations as follows:

9. This model is often referred to as a "local linear trend" model (with additionally a seasonal component); *lltrend* is an abbreviation of this name.

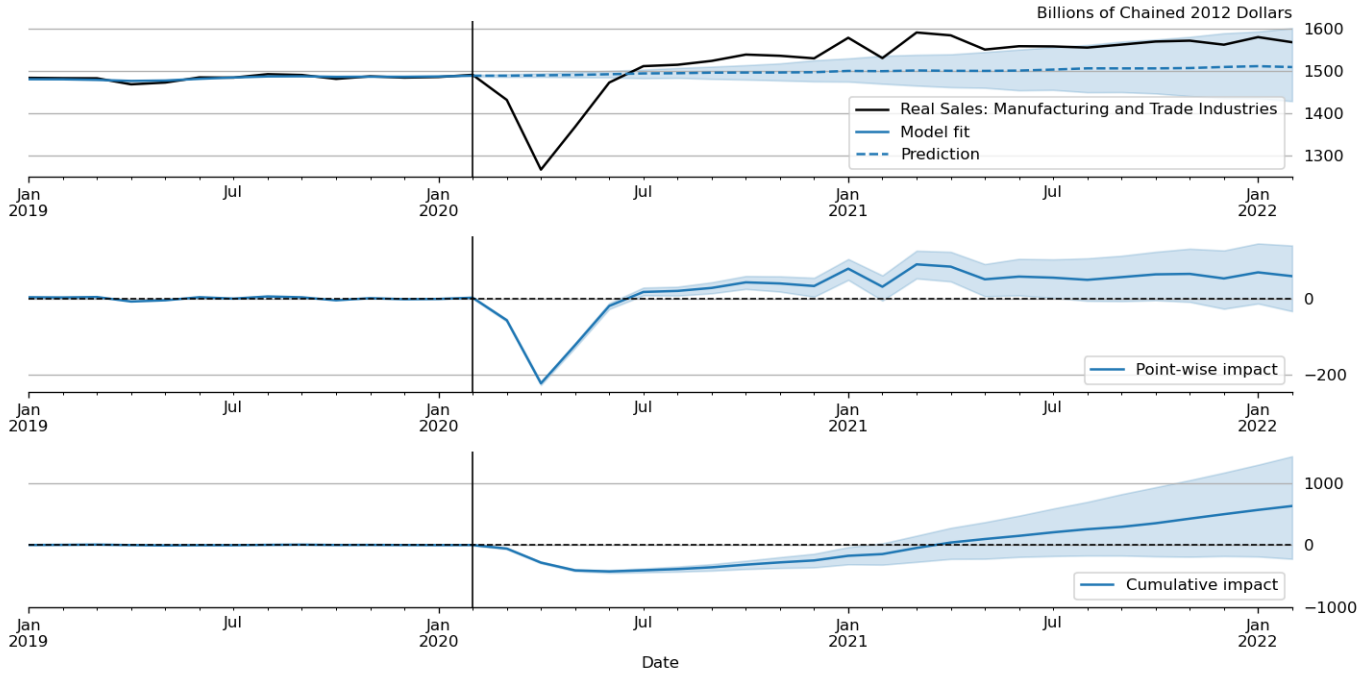

Fig. 6: *"Causal impact" of COVID-19 on U.S. Sales in Manufacturing and Trade Industries.*

```
# (b') Draw from the conditional posterior of
# the state vector
model.update(params[i - 1])
sim.simulate()
# save the draw for use later in time series
# decomposition
states[i] = sim.simulated_state.T

# Draw from the posterior predictive distribution
# using the `simulate` method
n_fcast = 48
fcast[i] = model.simulate(
   params[i - 1], n_fcast,
   initial_state=states[i, -1]).to_frame()
```

These forecasts and the decomposition into level, trend, and seasonal components are summarized in Figures 4 and 5, which show the median values along with 80% credible intervals. Notably, the intervals shown incorporate for both the uncertainty arising from the stochastic terms in the model as well as the need to estimate the models' parameters.[10]

Casual impacts

A closely related procedure described in [BGK+15] uses a Bayesian structural time series model to estimate the "causal impact" of some event on some observed variable. This approach stops estimation of the model just before the date of an event and produces a forecast by drawing from the posterior predictive density, using the procedure described just above. It then uses the difference between the actual path of the data and the forecast to estimate impact of the event.

An example of this approach is shown in Figure 6, in which we use this method to illustrate the effect of the COVID-19 pandemic on U.S. Sales in Manufacturing and Trade Industries.[11]

Extensions

There are many extensions to the time series models presented here that are made possible when using Bayesian inference. First, it is easy to create custom state space models within the `statsmodels` framework. As one example, the `statsmodels` documentation describes how to create a model that extends the typical VAR described above with time-varying parameters.[12] These custom state space models automatically inherit all the functionality described above, so that Bayesian inference can be conducted in exactly the same way.

Second, because the general state space model available in `statsmodels` and introduced above allows for time-varying system matrices, it is possible using Gibbs sampling methods to introduce support for automatic outlier handling, stochastic volatility, and regime switching models, even though these are largely infeasible in `statsmodels` when using frequentist methods such as maximum likelihood estimation.[13]

Conclusion

This paper introduces the suite of time series models available in `statsmodels` and shows how Bayesian inference using Markov chain Monte Carlo methods can be applied to estimate their parameters and produce analyses of interest, including time series decompositions and forecasts.

10. The popular Prophet library, [TL17], similarly uses an additive model combined with Bayesian sampling methods to produce forecasts and decompositions, although its underlying model is a GAM rather than a state space model.

11. In this example, we used a local linear trend model with no seasonal component.

12. For details, see https://www.statsmodels.org/devel/examples/notebooks/generated/statespace_tvpvar_mcmc_cfa.html.

13. See, for example, [SW16] for an application of these techniques that handles outliers, [KSC98] for stochastic volatility, and [KN98] for an application to dynamic factor models with regime switching.

REFERENCES

[BGK+15] Kay H. Brodersen, Fabian Gallusser, Jim Koehler, Nicolas Remy, and Steven L. Scott. Inferring causal impact using Bayesian structural time-series models. *Annals of Applied Statistics*, 9:247–274, 2015. doi:10.1214/14-aoas788.

[CGH+17] Bob Carpenter, Andrew Gelman, Matthew D. Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan : A Probabilistic Programming Language. *Journal of Statistical Software*, 76(1), January 2017. Institution: Columbia Univ., New York, NY (United States); Harvard Univ., Cambridge, MA (United States). URL: https://www.osti.gov/pages/biblio/1430202-stan-probabilistic-programming-language, doi:10.18637/jss.v076.i01.

[CJ09] Joshua C.C. Chan and Ivan Jeliazkov. Efficient simulation and integrated likelihood estimation in state space models. *International Journal of Mathematical Modelling and Numerical Optimisation*, 1(1-2):101–120, January 2009. Publisher: Inderscience Publishers. URL: https://www.inderscienceonline.com/doi/abs/10.1504/IJMMNO.2009.03009.

[DK02] J. Durbin and S. J. Koopman. A simple and efficient simulation smoother for state space time series analysis. *Biometrika*, 89(3):603–616, August 2002. URL: http://biomet.oxfordjournals.org/content/89/3/603, doi:10.1093/biomet/89.3.603.

[DK12] James Durbin and Siem Jan Koopman. *Time Series Analysis by State Space Methods: Second Edition*. Oxford University Press, May 2012.

[DLT+17] Joshua V. Dillon, Ian Langmore, Dustin Tran, Eugene Brevdo, Srinivas Vasudevan, Dave Moore, Brian Patton, Alex Alemi, Matt Hoffman, and Rif A. Saurous. TensorFlow Distributions. Technical Report arXiv:1711.10604, arXiv, November 2017. arXiv:1711.10604 [cs, stat] type: article. URL: http://arxiv.org/abs/1711.10604, doi:10.48550/arXiv.1711.10604.

[Ful15] Chad Fulton. Estimating time series models by state space methods in python: Statsmodels. 2015.

[HA18] Rob J Hyndman and George Athanasopoulos. *Forecasting: principles and practice*. OTexts, 2018.

[Har90] Andrew C. Harvey. *Forecasting, Structural Time Series Models and the Kalman Filter*. Cambridge University Press, 1990.

[HKOS08] Rob Hyndman, Anne B. Koehler, J. Keith Ord, and Ralph D. Snyder. *Forecasting with Exponential Smoothing: The State Space Approach*. Springer Science & Business Media, June 2008. Google-Books-ID: GSyzox8Lu9YC.

[KCHM19] Ravin Kumar, Colin Carroll, Ari Hartikainen, and Osvaldo Martin. ArviZ a unified library for exploratory analysis of Bayesian models in Python. *Journal of Open Source Software*, 4(33):1143, 2019. Publisher: The Open Journal. URL: https://doi.org/10.21105/joss.01143, doi:10.21105/joss.01143.

[KN98] Chang-Jin Kim and Charles R. Nelson. Business Cycle Turning Points, A New Coincident Index, and Tests of Duration Dependence Based on a Dynamic Factor Model With Regime Switching. *The Review of Economics and Statistics*, 80(2):188–201, May 1998. Publisher: MIT Press. URL: https://doi.org/10.1162/003465398557447, doi:10.1162/003465398557447.

[KN99] Chang-Jin Kim and Charles R. Nelson. State-Space Models with Regime Switching: Classical and Gibbs-Sampling Approaches with Applications. MIT Press Books, The MIT Press, 1999. URL: http://ideas.repec.org/b/mtp/titles/0262112388.html.

[KSC98] Sangjoon Kim, Neil Shephard, and Siddhartha Chib. Stochastic Volatility: Likelihood Inference and Comparison with ARCH Models. *The Review of Economic Studies*, 65(3):361–393, July 1998. 01855. URL: http://restud.oxfordjournals.org/content/65/3/361, doi:10.1111/1467-937X.00050.

[MPS11] Wes McKinney, Josef Perktold, and Skipper Seabold. Time Series Analysis in Python with statsmodels. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 10th Python in Science Conference*, pages 107 – 113, 2011. doi:10.25080/Majora-ebaa42b7-012.

[SP10] Skipper Seabold and Josef Perktold. Statsmodels: Econometric and Statistical Modeling with Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 92 – 96, 2010. doi:10.25080/Majora-92bf1922-011.

[SW16] James H. Stock and Mark W. Watson. Core Inflation and Trend Inflation. *Review of Economics and Statistics*, 98(4):770–784, March 2016. 00000. URL: http://dx.doi.org/10.1162/REST_a_00608, doi:10.1162/REST_a_00608.

[SWF16] John Salvatier, Thomas V. Wiecki, and Christopher Fonnesbeck. Probabilistic programming in Python using PyMC3. *PeerJ Computer Science*, 2:e55, April 2016. Publisher: PeerJ Inc. URL: https://peerj.com/articles/cs-55, doi:10.7717/peerj-cs.55.

[TL17] Sean J. Taylor and Benjamin Letham. Forecasting at scale. Technical Report e3190v2, PeerJ Inc., September 2017. ISSN: 2167-9843. URL: https://peerj.com/preprints/3190, doi:10.7287/peerj.preprints.3190v2.

[WH99] Mike West and Jeff Harrison. *Bayesian Forecasting and Dynamic Models*. Springer, New York, 2nd edition edition, March 1999. 00000.