Josue Casco-Rodriguez
Prof. Caleb Kemere
Neural Signal Processing

Discriminative Kalman Filter Python Implementation

Abstract

In 2016, Burkhart *et al*. published a new Bayesian decoding algorithm for neural signals: the Discriminative Kalman Filter (DKF). Unlike the original Kalman Filter (which requires a linear relationship between latent states and a linear relationship between observations and latent states) and its popular nonlinear extensions (the Extended and Unscented Kalman Filters, which attempt to linearize nonlinear models by using a model for p(observation | state)), the DKF approaches nonlinear Bayesian filtering by using a model for p(state | observation). The advantage of an algorithm incorporating a model for p(state | observation) becomes especially clear when considering that neural states are often much lower in dimensionality than neural observations. In 2020, Burkhart *et al.* introduced the DKF framework into real-time decoding applications for clinical trials of human neuroprosthetics. This project is an effort to reproduce results from the authors' 2020 paper in Python – specifically, Tables 1 and 2. The former characterizes the normalized root-mean square error (nRMSE) of various decoding methods (including the DKF), while the latter characterizes their mean absolute angular errors. Both the implementation here and the authors' implementation found that the DKF using Nadaraya-Watson kernel regression is the most robust. All code is available through GitHub at https://github.com/Josuelmet/Discriminative-Kalman-Filter-4.5-Python.

1. Introduction

Kalman filters are a family of algorithms whose purpose is to estimate a set of states $\{Z_1, Z_2, Z_3, \ldots, Z_T\}$ given a set of observations $\{X_1, X_2, X_3, \ldots, X_T\}$. Kalman filters operate under the Markov assumption that the observation X_i depends only on its corresponding state Z_i, and that Z_i depends only on its immediately preceding latent state Z_{i-1}. However, unlike Hidden Markov models, the latent states $\{Z_1, Z_2, \ldots, Z_T\}$ are not discrete, instead having continuous values. For real-time decoding, the primary task of interest is to predict the latest latent state Z_T when given the previous latent state Z_{T-1} and the latest observation X_T. Other applications of Kalman filters include smoothing (predicting Z_i given $\{X_1, \ldots, X_T\}$, where $1 \leq i \leq T$) and projection into the future (predicting $\{Z_T, Z_{T+1}, Z_{T+2}, \ldots\}$ given $\{X_1, \ldots, X_T\}$.)

The original Kalman Filter (KF) is a linear, Gaussian model, and thus assumes the following:

- The observation transformation from Z_i to X_i, also known as $p(X_i \mid Z_i)$, is linear with zero-mean Gaussian noise.
- The process transformation from Z_{i-1} to Z_i, also known as $p(Z_i \mid Z_{i-1})$, is linear with zero-mean Gaussian noise.
- The covariance of latent state estimation undergoes the same linear transformation as the latent state itself, with a constant noise covariance matrix being added every timestep.

The KF is the optimal estimator for linear dynamic systems with Gaussian observation and process noise, but neural systems are often nonlinear. To this end, the simplest method for handling nonlinear observation models is the Extended Kalman Filter (EKF). The EKF makes the following modification to the original KF model:

- The observation transformation, h(), from Z_i to X_i is differentiable and nonlinear with zero-mean Gaussian noise.
- The process transformation, f(), from Z_{i-1} to Z_i is differentiable and nonlinear with zero-mean Gaussian noise.
- The transformations f() and h() cannot be applied directly to the latent state covariance. Instead, the Jacobians of f() and h(), evaluated at the current value of Z_t, are applied at timestep t.

While the EKF can perform well with sufficient knowledge of the system, it can also perform poorly without such knowledge, or when strong nonlinearities are involved in the system. Another nonlinear Kalman filter algorithm is the Unscented Kalman Filter (UKF). The UKF differs from the EKF by using a deterministic sampling technique known as the unscented transform to pick a minimal set of sample points (sigma points) around the mean. By

incorporating sampling techniques, the UKF allows usage of transformations whose Jacobians are difficult or impossible to calculate (i.e., large or non-differentiable functions).

Another Bayesian filtering approach is the particle filter. Particle filters allow for incorporation of nonlinear, non-stationary transformations by estimating arbitrary distributions through sampling and resampling techniques. Such filters can work quite well when using enough particles (number of samples), but struggle with very high-dimensional systems and can be too computationally expensive for real-time applications. While Burkhart *et al.* did not use a particle filter in producing Tables 1 and 2 (likely due to particle filters' high computational costs), they did use particle filters in other figures throughout their 2020 publication, specifically to display how the DKF can perform as well as a particle filter while operating much more quickly.

2. Discriminative Kalman Filter

Like the Kalman filter, the Discriminative Kalman Filter (DKF) assumes that the latent state Z undergoes a zero-mean Gaussian random walk with state transition matrix A and process noise covariance Q (also referred to as G). In other words, $p(z_t \mid z_{t-1}) \sim N(Az_{t-1}, G)$, where N() is a multivariate Gaussian distribution. Also like the Kalman filter, the DKF assumes that observation model $p(x_t \mid z_t)$ is stationary.

A key innovation of the DKF is that it approximates $p(z_t \mid x_t)$ as $p(x_t \mid z_t) / p(x_t)$, using Bayes's Rule. Such a substitution can prove useful if $p(z_t \mid x_t)$ is strongly nonlinear or non-Gaussian, as is the case in neural signals. The DKF further models $p(z_t \mid x_t)$ as Gaussian. Specifically, $p(z_t \mid x_t) \sim N(f(x_t), Q(x_t))$, where f(x) and Q(x) are nonlinear functions that map the observation to its corresponding vectors in the state space R^d and the covariance space S_d, respectively.

The authors formulate that the observation-to-state transformation f() and the observation-to-covariance transformation Q() are the conditional mean and covariance of Z given X. In other words:

- $f(x) = E(Z_t \mid X_t = x)$
- $Q(x) = V(Z_t \mid X_t = x)$

The DKF also makes the stationary assumption $p(z_t) \sim N(0, VZ)$, where VZ (also referred to as S or T) is the covariance of z_t when not conditioned on any z_{t-1}. Defining the initial hidden state estimate $\mu_0 = 0$ and the initial hidden covariance estimate $\sum_0 = S$ (where S, also known as VZ or T, is the covariance of $p(z_t)$, and the covariance $\sum$ is also referred to as P in traditional filtering literature), the DKF algorithm proceeds iteratively as such:

1) $v_t = A\mu_{t-1}$
2) $M_t = A\sum_{t-1}A^T + G$
3) $\sum_t = (M_t^{-1} + Q(x_t)^{-1} - S^{-1})^{-1}$
4) $\mu_t = \sum_t(M_t^{-1}v_t + Q(x_t)^{-1}f(x_t))$

In practice, the following additional changes are made to the DKF algorithm:

- $Q(x_t)^{-1} - S^{-1}$ must be positive definite. If it is not, set $Q(x_t)^{-1} = (Q(x_t)^{-1} + S^{-1})^{-1}$.
- $\sum_t = (M_t^{+} + Q(x_t)^{+} - S^{+})^{+}$
- $\mu_t = \sum_t(M_t^{+}v_t + Q(x_t)^{+}f(x_t))$
- Pseudo-inverses are denoted by $^{+}$.

Another formulation of the DKF more suitable for real-time applications is the Robust DKF, which makes the assumption that $p(z_t) \sim N(0, S)$, but where the eigenvalues of S^{-1} are so small that the $-S^{-1}$ term in Step 3 is negligible. Thus, the iterative equations for the Robust DKF remain unchanged, with the exception of the $-S^{-1}$ term being removed. Additionally, an improper prior is placed on Z_0, and modeling starts from $t = 1$. Specifically, $\mu_1 = f(x_1)$ and $\sum_1 = Q(x_1)$, and iterative calculations start $t = 2$ instead of $t = 1$.

3. Data

The data used is from Flint *et al.* (2012). Specifically, the data is from a 96-channel microelectrode array implanted in the primary motor cortex of a rhesus monkey. The monkey was taught to earn juice rewards by moving a manipulandum. A 128-channel acquisition system recorded the resulting signals, which were sampled at 30 kHz, highpass-filtered at 300 Hz, and then thresholded and sorted into spikes offline. The data is made publicly available by Walker and Kording (2013) under the Database for Reaching Experiments and Models (DREAM): https://portal.nersc.gov/project/crcns/download/dream/data_sets/Flint_2012.

The five .mat files from the Flint *et al.* (2012) experiment must be placed in the same directory as *flint_preprocess_data.ipynb*. The notebook performs both data preprocessing and wrangling. First, the notebook loads data from all five experiments. Since the original data was saved in the DREAM database as MATLAB structs, wrangling in Python can seem more convoluted than wrangling in MATLAB. Each experiment undergoes the following wrangling:

1) Isolate all data from the i-th subject of the experiment. Some experiments only have one subject (i = 0), while other experiments have up to four subjects (i = 0, 1, 2, or 3).
2) For each trial in the experiment:
    a) Stack each timestamp's three-dimensional velocity vertically.
    b) For each neuron in the trial:

        i) Discretize the neuron's spike times by binning them with respect to the timestamps produced by 30 kHz sampling.
    c) Stack the neurons' spike bins vertically.
3) Only keep the first two dimensions of the velocities, since the third dimension is not relevant for operation of the manipulandum.
4) Save the resulting 2D array of spike bins from the i-th subject as a unique entry in a list of 2D spike bin arrays. In equivalent fashion, store the array of velocities in a list of 2D velocity arrays.
5) Repeat steps (1) through (4) for each subject from each experiment. There are a total of 12 subjects across all experiments, treating each subject from each experiment as a different subject, regardless of whether they are the same monkey.

The data preprocessing in *flint_preprocess_data.ipynb* proceeds as such:

1) Isolate the velocity and spike bin data from the i-th subject.
2) Downsample the spike bin data into intervals of 100 ms.
3) Replace the spike bin data with a moving sum (with a window length of 10) of the spike bin data.
4) Downsample the velocity data into intervals of 100 ms – specifically, indices that are halfway between the indices used to downsample the spike bin data.
5) Reduce the spike bin data to 10 dimensions via Principal Component Analysis.
6) Replace the spike bin data with its z-scores, using 1 degree of freedom correction, as per MATLAB's zscore function.
7) As during wrangling, store the 2D array of processed spike bins and the 2D array of processed velocities in a list of 2D spike arrays and a list of 2D velocity arrays, respectively.
8) Repeat steps (1) through (8) for each of the 12 subjects.

The result of *flint_preprocess_data.ipynb* is an array of processed velocities and an array of processed spike bins. Each array is stored as a .npy file, which must be moved into the same directory as *Paper_Script_45.ipynb* before proceeding. Each velocity has 2 dimensions, and each spike bin observation has 10 dimensions. In other words, the latent state (velocity) is 2-dimensional, while the observations (z-scored principal component scores of neural activity) are 10-dimensional.

4. Methods

The results that Burkhart *et al.* published in 2020 were produced using MATLAB. The goal of this project is to reproduce their Experiment 4.5 results in Python. All analysis occurs in *Paper_Script_45.ipynb*, which can be run once the two .npy files representing the preprocessed velocities and neural spike activity are located in the same directory as it.

While there are a total of 12 subjects, analysis was only conducted on the first 6 subjects (which the authors refer to as trials), since the authors seem to have done the same. The following analyses and procedures were performed separately on each of the aforementioned subjects.

4.1 Training, Validation, and Test Data Split

Before performing regression, the data from the current subject is isolated and split into training and test data. The first 5,000 samples are used as training data, while the subsequent 1,000 samples are used as test data. Various methods are used for learning the observation-to-state transformation $f:R^{10}\rightarrow R^2$ or the state-to-observation transformation $f:R^2\rightarrow R^{10}$. However, only Nadaraya-Watson (NW) kernel regression is used for learning the transformation $Q:R^{10}\rightarrow S_2$, where Q(x) is the estimated conditional covariance of Z_t given X_t. Burkhart *et al.* (2020) used 70% of the training data (3,500 samples) purely to train regression models, while the remaining 30% (1,500 samples) are used to learn Q(x) using NW regression; thus, the same approach is done in Python for the current subject. The aforementioned 3,5000 samples will be referred to as training data, the next 1,500 samples will be referred to as validation data, and the next 1,000 samples will be referred to as testing data, for clarity. While the indices of the 5,000 (training + validation) and 1,000 samples are not randomized, whe indices of the 3,500 (training) and the 1,500 (validation) samples are randomly drawn (without replacement) from the 5,000 samples.

4.2 Linear Kalman Filter

The first, and most important, regression method is the simple linear KF. It acts as a fundamental baseline upon which to compare subsequent algorithms. It uses all 5,000 training and validation samples as training data, since it does not need a validation set with which to learn a Q(x) covariance function. Aside from estimated latent state means and covariances, the KF also yields the transition matrix A, the process noise covariance Q (referred to in the notebook as G), and the initial estimate covariance P_0 (referred to as VZ or V0).

4.3 Neural Network regression

The next regression method is the DKF using NN regression. All DKF methods involve learning the transformation $f:R^{10}\rightarrow R^2$ (i.e., the transformation from observation to velocity). The NN consists of a feedforward network with 2 hidden layers, each having 10 neurons, with each

hidden neuron having a hyperbolic tangent function. The output layer has a linear output function, since the latent states are continuous and take both positive and negative values. The network trains on the 3,500 training samples over the course of 4,000 iterations, using a learning rate of 1e-3, and l2-norm regularization of strength 1e-4.

Burkhart *et al.* (2020) used a neural network with one hidden layer of 10 neurons, with each hidden neuron having a hyperbolic tangent function. However, the authors trained their smaller network using MATLAB's Levenberg-Marquardt (LM) optimization algorithm with Bayesian regularization (BR) (Foresee and Hagan, 1997). The LMBR algorithm automatically calculates l2 weight penalty strength in an iterative fashion, while incorporating information from the inverse Hessian of the loss function (i.e., the loss function's curvature), resulting in fast, generalizable neural networks. However, resources for LMBR optimization in Python are scarce, and calculation of the inverse Hessian does not bode well for large datasets. Hence, a more common neural network architecture and optimization method (gradient descent with l2 weight minimization) was chosen.

Once the neural network learns the function $f:R^{10} \rightarrow R^2$, the NN generates a prediction of latent states using the validation data. The optimal bandwidth of the radial basis kernel for covariance estimation is then found by minimizing the leave-one-out mean squared error of NW kernel regression using the validation set and the outer product of the validation residuals (Z_i - $f(X_i)$ for all $\{X_i, Z_i\}$ pairs in the validation data). Next, the NN generates a prediction of latent states using the test data, while NW kernel regression estimates state covariances using the test data, validation data, and validation residuals. Finally, the DKF-NN predictions are made by passing the network's predicted states and covariances, along with the aforementioned Kalman filter parameters A (state transition matrix), Q (G, the process noise covariance), and VZ (where VZ equals the stationary covariance of $p(z_t)$ when not conditioned on any z_{t-1}).

4.4 Long Short-Term Memory regression

The other neural network used to evaluate the DKF is a long short-term memory (LSTM) neural network. At the cost of accuracy, an LSTM architecture was chosen that was simpler than what the authors had implemented. The architecture implemented here consists of one LSTM layer with 2 hidden states, one fully-connected hidden layer of 10 neurons, and an output layer of 2 neurons. All but the final neurons use a rectified linear unit activation function. Unlike all other DKF, EKF, and UKF methods, DKF-LSTM regression partitions the training and validation differently. The training data corresponds to the first 3,500/5,000 indices in the training + validation data, while the validation data corresponds to the last 1,500/5,000 indices in the training + validation data. In other words, the DKF-LSTM regression skips the shuffling of training and validation data.

The authors did not use the DKF to process LSTM estimated states. However, in this implementation, the LSTM is trained using the 3,500 training samples, and then NW kernel regression learns the covariance function $Q:R^{10}\rightarrow S_2$ using the validation data, as in the DKF-NN method. Next, the LSTM's estimated latent states and the NW estimated covariances evaluated from the test data are passed into the DKF algorithm with the Kalman filter parameters A, Q (G), and VZ (as before), thus yielding the DKF-LSTM estimates.

4.5 Nadaraya-Watson Kernel regression

Nadaraya-Watson kernel regression for DKF evaluation functions very similarly to how NW regression estimated the state covariance function $Q:R^{10}\rightarrow S_2$ in earlier methods. First, the bandwidth of the radial basis kernel is optimized to minimize the leave-one out mean squared error in the estimated function $f:R^{10}\rightarrow R^2$ using only the 3,5000 training samples. Next, the learned function f() is used to predict the velocities (states) of all 1,5000 validation samples. The resulting residuals (Z_i - $f(X_i)$ for all $\{X_i, Z_i\}$ pairs in the validation data) are then used to learn Q(x) in the same fashion as in the neural network evaluations of the DKF. Once the optimal bandwidths of NW-regression estimates of f() and Q() have been found, NW kernel regression generates estimates of latent states and state covariances for the 1,000 test samples. The final DKF-NW estimates are then made by passing the predicted states and covariances, along with the linear Kalman filter parameters A, Q (G), and VZ into the DFK algorithm, as in earlier methods.

4.6 Gaussian Process regression

The final method of DKF evaluation consists of Gaussian process regression for learning the function $f:R^{10}\rightarrow R^2$. The authors utilized the GPML package in MATLAB to train their Gaussian process models (Ramussen and Nickisch, 2010). However, due to the difficulty of using GPML in Python, the GaussianProcessRegressor (GPR) class from scikit-learn was used instead. While the GPR class and the GPML package are both based on the work done by Ramussen *et al.*, GPR regression is less customizable and flexible than GPML regression, thus resulting in a decrease in accuracy compared to the authors' work (Burkhart *et al.*, 2020). Additionally, the authors implemented separate 1-dimensional Gaussian process regressions for each dimension of the R^2 latent states, while the regression implemented here fits only one 2-dimensional regression for both dimensions of the latent states. The decision to unify the Gaussian process regressions stems from both a significant improvement in runtime and a lack of accuracy decrease compared to learning two distinct regressions. The Gaussian process regression implemented here uses a radial basis kernel (as the authors do), but its parameters (e.g., bandwidth) are mostly fixed at runtime, because GPR parameter optimization often produced very large bandwidths and large runtimes with no significant boost in performance.

Once the GPR class fits a regression $f:R^{10}\rightarrow R^2$ onto the training data, the residuals of the estimated function f() are calculated on the validation data, as in previous DKF evaluation

methods. The residuals and the validation data are then used to learn $Q:R^{10}\rightarrow S_2$ via NW kernel regression. The GPR regression function f() and the NW kernel regression function Q() are then used to estimate the latent states and covariances of the test data; the DKF algorithm yields the final DKF-GP estimates by using the estimated state and covariances and the Kalman parameters A, Q (G), and VZ (as in previous methods).

4.7 Extended Kalman Filter

Usage of the Extended Kalman Filter (EKF) and Unscented Kalman Filter (EKF) begins with learning the state-to-observation function $f:R^2\rightarrow R^{10}$, as opposed to DKF methods, which involve learning the transformation from observation to state. To this end, a neural network with the same architecture as that of the DKF-NN method is used (2 hidden layers of 10 neurons, with each hidden neuron having a hyperbolic tangent activation function), albeit with an input layer of 2 neurons and an output layer of 10 neurons. The same optimization method and parameters, and training data are also used (4000 iterations, a learning rate of 1e-3, and an l2 weight norm penalty of 1e-4). As in the DKF-NN method, the authors had used a simpler neural network (1 layer of 10 hidden neurons) using LMBR optimization, but here a more common neural network architecture was chosen due to the difficulty of using LMBR optimization for neural networks in Python. For both the EKF and the UKF, the observation noise (R) is estimated as the covariance of the residuals evaluated over the validation data (i.e., the covariance of the array of X_i - $f(Z_i)$ for all $\{X_i, Z_i\}$ pairs in the validation data).

The EKF uses the function $f:R^2\rightarrow R^{10}$ learned by the neural network as its observation function. The Jacobian of the function is available through PyTorch, and is supplied to the EKF algorithm. The EKF uses the residual-calculated R and the Kalman filter parameters A (the state transition matrix), Q (G, the process noise), and an initial covariance estimate P_0 equal to VZ. The EKF predictions over the test data are then predicted iteratively using the aforementioned Jacobian and observation function.

4.8 Unscented Kalman Filter

The UKF uses the same neural network state-to-observation transformation $f:R^{10}\rightarrow R^2$ as the EKF. While the UKF can explicitly operate with nonlinear state transitions (unlike the EKF), the state transition function $F:R^2\rightarrow R^2$ is $F(z) = Az$, where A is the state transition matrix learned from the Kalman filter, as in all other prediction methods here. The UKF uses the same matrix parameters as the EKF (the process noise Q, the observation noise R, and the initial state covariance P_0). UFK predictions are generated over the test data in the same fashion as EKF predictions. One key difference between the UKF algorithm here and the UKF algorithm used by the authors (Burkhart *et al.*, 2020) is that MATLAB differs from Python in how the number of sample (sigma) points are calculated.

5. Results

After evaluating the Kalman filter and its Discriminative, Extended, and Unscented variants on 6 trials (where each trial has new training, validation, and test data), the normalized root-mean square error (nRMSE) and the mean absolute angle error (MAAE) of each method are calculated with respect to the test data. The nRMSE is defined as the root-mean square error between the test data and its estimate divided by the root-mean square of the test data. Thus, the nRMSE has a value of 1 when the provided estimate is simply all zeros.

	Trial 0	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Avg.
Kalman	0.765	0.945	0.788	0.792	0.779	0.761	0.805
DKF-NW	-19%	-18%	-9%	-21%	-19%	-20%	-18%
DKF-GP	-7%	-10%	-8%	-9%	-12%	-9%	-9%
DKF-NN	-23%	-10%	-5%	-14%	-20%	-21%	-15%
DKF-LSTM	-8%	+30%	-6%	-4%	-10%	-12%	-1%
EKF	+2%	+8%	+9%	+19%	+17%	+12%	+11%
UKF	+1%	-2%	+7%	+19%	+9%	+12%	+7%

Table 1: Normalized Root-Mean Square Error (nRMSE)

The DKF-NW, DKF-NN, and EKF regression methods performed as effectively as they had in the work of Burkhart *et al.* (2020). The DKF-NW proved to be especially robust, configurable, and interpretable. However, most of the other methods performed worse than their counterparts in the authors' work, on average. The only exception was the UKF, which performed better than the authors' UKF, likely due to the differences in neural network architecture (caused by the inaccessibility of LMBR optimization in Python). The DKF-GP likely performed worse because of the wider array of optimization options in the GPML MATLAB package compared to those in the GPR scikit-learn class. Despite using the DKF to improve the accuracy of LSTM predictions, the DKF-LSTM underperformed compared to the LSTM architecture implemented by the authors without the DKF. The DKF-LSTM likely underperformed due to either its relatively simple architecture, overfitting, or strong nonstationarities in the training and validation data that were exacerbated by the non-random distribution of training data in DKF-LSTM training.

The MAAE is defined as the magnitude of the angular difference between estimated and true latent states in the test data. Each latent state is a 2-dimensional velocity, so the angle of state Z_i is the arctangent of ($Z_i[1]$ / $Z_i[0]$). The MAAE is in radians. Note that 45° = π/4 radians, which is approximately 0.79 radians, meaning that all the methods used have significant angular error. The MAAE can be a useful metric in evaluating neural decoder performance, especially if brain-computer interface cursor speed is adjustable.

	Trial 0	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Avg.
Kalman	0.884	0.957	1.026	0.930	0.966	0.926	0.948
DKF-NW	-14%	0%	-21%	-15%	-25%	-29%	-17%
DKF-GP	-2%	+12%	-18%	-4%	-15%	-20%	-8%
DKF-NN	-9%	0%	-15%	-10%	-17%	-22%	-12%
DKF-LSTM	-4%	+6%	-12%	-7%	-14%	-11%	-7%
EKF	-1%	+8%	+2%	+4%	-4%	0%	+2%
UKF	-2%	+5%	+1%	+2%	-9%	-3%	-1%

Table 2: Mean Absolute Angle Error

The MAAE results fall closer to those of Burkhart *et al.* (2020). The DKF-GP likely performed worse due to its aforementioned differences with respect to the authors' DKF-GP method, while the EKF likely performed worse because of differences in neural network architecture. Surprisingly, the DKF-LSTM had significantly underperformed with respect to nRMSE, but performed as expected with respect to MAAE.

6. Conclusion

The Discriminative Kalman Filter (DKF) provides an exciting breakthrough in Bayesian filtering, especially when the observation model is known (Burkhart *et al.*, 2020). However, when the observation is unknown (as is the case in neural decoding), it must be learned. In their Experiment 4.5, Burkhart *et al.* evaluated the DKF using neural network (NN) regression, Nadaraya-Watson (NW) kernel regression, and Gaussian process (GP) regression, along with a long short-term memory (LSTM) neural network operating without the DKF framework. In this project, the DKF-NN, DKF-NW, and DKF-GP regression methods were evaluated, along with a DKF-LSTM method that combined LSTM regression with the DKF algorithm. While it is likely, and the authors themselves concede, that an LSTM could surpass DKF decoding if its optimal

parameters and architecture were found, the DKF-NW method was the most robust, both in this evaluation and the authors'. The DKF-NW outperformed the other methods in both normalized root-mean square error (nRMSE) and mean absolute angular error (MAAE).

Many of the filtering methods implemented here in Python had reduced nRMSEs compared to their counterparts in the authors' MATLAB work. Although some methods (such as the DKF-LSTM) were intentionally more simple than the authors' implementations, the largest discrepancies in algorithm implementation came from the difficulty of reproducing their MATLAB processes exactly in Python. The most significant algorithms that were difficult to reproduce in Python were the GPML Gaussian process regression package and the neural network Levenberg-Marquardt optimization algorithm with Bayesian regularization (LMBR).

Attempting to reproduce the results of Burkhart *et al*. (2020) was a rigorous introduction not only to real-world Bayesian filtering in the context of neuroengineering, but also to the broad statistical field of nonlinear regression. Understanding the limitations and assumptions behind the cutting edge of neural engineering is invaluable to understanding how to advance the field.

References

Burkhart, M. C., Brandman, D. M., Franco, B., Hochberg, L. R., Harrison, M. T. (2020). The Discriminative Kalman Filter for Bayesian Filtering with Nonlinear and Non-Gaussian Observation Models. *Neural Computation*, 32(5): 969–1017. doi:10.1162/neco_a_01275.

Burkhart, M. C., Brandman, D. M., Vargas-Irwin, C. E., Harrison, M.T. (2016). The discriminative Kalman filter for nonlinear and non-Gaussian sequential Bayesian filtering. *ArXiv*, abs/1608.06622.

Flint, R. D., *et al*. (2012). Accurate decoding of reaching movements from field potentials in the absence of spikes. *Journal of Neural Engineering* 9(4):046006. doi:10.1088/1741-2560/9/4/046006.

Foresee, F.D., Hagan, M.T. (1997). Gauss-Newton approximation to Bayesian learning. *International Conference on Neural Networks*, volume 3, pages 1930–1935.

Rasmussen, C.E., Nickisch, H. (2010). Gaussian processes for machine learning (GPML) toolbox. *Journal of Machine Learning Research*, 11:3011–3015.

Walker, B., Kording, K. (2013). The Database for Reaching Experiments and Models. *PLOS One* 8(11): e78747. doi:10.1371/journal.pone.0078747.