NeMoS

Preparing data

```
binned_current = current.bin_average(bin_size)

print(f"current shape: {binned_current.shape}")
# rate is in Hz, convert to KHz
print(f"current sampling rate: {binned_current.rate/1000.:.02f} KHz")

print(f"\ncount shape: {count.shape}")
print(f"count sampling rate: {count.rate/1000:.02f} KHz")
```

- predictors must be 2d, spikes 1d

```
predictor = np.expand_dims(binned_current, 1)

# check that the dimensionality matches NeMoS expectation
print(f"predictor shape: {predictor.shape}")
print(f"count shape: {count.shape}")
```

Fitting the model

- GLM objects need regularizers and observation models

```
# Initialize the model, specifying the solver. we'll accept the defaults
# for everything else.
model = nmo.glm.GLM(solver_name="LBFGS")
```

- call fit and retrieve parameters

```
model.fit(predictor, count)
```

```
print(f"firing_rate(t) = exp({model.coef_} * current(t) + {model.intercept_})")
```

```
print(f"coef_ shape: {model.coef_.shape}")
print(f"intercept_ shape: {model.intercept_.shape}")
```

- generate model predictions.

```
predicted_fr = model.predict(predictor)
# convert units from spikes/bin to spikes/sec
predicted_fr = predicted_fr / bin_size


# and let's smooth the firing rate the same way that we smoothed the firing rate
smooth_predicted_fr = predicted_fr.smooth(0.05, size_factor=20)
```

- what do we see?

```
# compare observed mean firing rate with the model predicted one
print(f"Observed mean firing rate: {np.mean(firing_rate)} Hz")
print(f"Predicted mean firing rate: {np.mean(predicted_fr)} Hz")
```

- examine tuning curve — what do we see?

```
tuning_curve_model = nap.compute_tuning_curves(predicted_fr, current, bins=15, feature_names=["current"]
fig = doc_plots.tuning_curve_plot(tuning_curve)
tuning_curve_model.plot(color="tomato", label="glm")
fig.legend()
```

Extending the model to use injection history

- choose a length of time over which the neuron integrates the input current

```
current_history_duration_sec = .2
# convert this from sec to bins
current_history_duration = int(current_history_duration_sec / bin_size)
```

```
binned_current[1:]
binned_current[2:]
# etc
```

- define a basis object

```
basis = nmo.basis.RaisedCosineLogConv(
    n_basis_funcs=8, window_size=current_history_duration,
)
```

- create the design matrix
- examine the features it contains

```
# under the hood, this convolves the input with the filter bank visualized above
current_history = basis.compute_features(binned_current)
print(current_history)
```

- create and fit the GLM
- examine the parameters

```
history_model = nmo.glm.GLM(solver_name="LBFGS")
history_model.fit(current_history, count)
```

```
print(f"firing_rate(t) = exp({history_model.coef_} * current(t) + {history_model.intercept_})")
```

```
print(history_model.coef_.shape)
print(history_model.intercept_.shape)
```

- examine the predicted average firing rate and tuning curve
- what do we see?

```
# compare observed mean firing rate with the history_model predicted one
print(f"Observed mean firing rate: {np.mean(count) / bin_size} Hz")
print(f"Predicted mean firing rate (instantaneous current): {np.nanmean(predicted_fr)} Hz")
print(f"Predicted mean firing rate (current history): {np.nanmean(smooth_history_pred_fr)} Hz")
```

```
# Visualize tuning curve
tuning_curve_history_model = nap.compute_tuning_curves(smooth_history_pred_fr, current, bins=15, featu
fig = doc_plots.tuning_curve_plot(tuning_curve)
tuning_curve_history_model.plot(color="tomato", label="glm (current history)")
tuning_curve_model.plot(color="tomato", linestyle='--', label="glm (instantaneous current)")
fig.axes[0].legend()
```

- use log-likelihood to compare models

```
log_likelihood = model.score(predictor, count, score_type="log-likelihood")
print(f"log-likelihood (instantaneous current): {log_likelihood}")
log_likelihood = history_model.score(current_history, count, score_type="log-likelihood")
print(f"log-likelihood (current history): {log_likelihood}")
```

Finishing up

- what if you want to compare models across datasets?

```
r2 = model.score(predictor, count, score_type='pseudo-r2-Cohen')
print(f"pseudo-r2 (instantaneous current): {r2}")
r2 = history_model.score(current_history, count, score_type='pseudo-r2-Cohen')
print(f"pseudo-r2 (current history): {r2}")
```