KleisleMurphy_371_Code

June 4, 2021

0.1 Import and Ingest

This section contains imports and ingests – notably, data is read in here

[ ]:
```
from google.colab import drive
drive.mount('/content/drive')

import itertools
import pandas as pd
import autograd.numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import multivariate_normal, multinomial
from sklearn.preprocessing import StandardScaler, MinMaxScaler

from autograd import grad, elementwise_grad
# from autograd.scipy.stats import multivariate_normal
from autograd.scipy.special import gammaln

from tqdm import trange

import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions
tfpl = tfp.layers

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

import pymc3 as pm

pa_filename = 'pa_data_v2.csv'
tracking_filename = 'track.csv'
ss_filename = 'sprint_speed_data.csv'
xstats_filename = 'expected_stats.csv'

# read in raw pa-level data
```

```
pa_data = pd.read_csv(f'/content/drive/Shared drives/Phillies Analytics/
↪Documentation and Library/Data/Statcast-IKM/{pa_filename}')
# read in statcast exit data
tracking_df = pd.read_csv(f'/content/drive/Shared drives/Phillies Analytics/
↪Documentation and Library/Data/Statcast-IKM/{tracking_filename}')
# read in sprint speed data
sprint_speed_df = pd.read_csv(f'/content/drive/Shared drives/Phillies Analytics/
↪Documentation and Library/Data/Statcast-IKM/{ss_filename}')
# read in expected stats
xstats_df = pd.read_csv(f'/content/drive/Shared drives/Phillies Analytics/
↪Documentation and Library/Data/Statcast-IKM/{xstats_filename}')
```

0.2 Wrangling

[ ]:
```
## Wrangling and a touch more setup

# these are the known variables
TARGS = [
    'sim_bb',
    'sim_k',
    'sim_if_hit',
    'sim_x1b_gb',
    'sim_x1b_ab',
    'sim_x2b_gb',
    'sim_x2b_ab',
    'sim_x3b',
    'sim_hr',
    'sim_gb_out',
    'sim_ld_out',
    'sim_fb_out',
    'sim_pu_out'
]
# these are the unknowns
X_FEATURES = ['avg_ev', 'avg_la', 'xwoba', 'xslg', 'xba']
X_FEATURES_AUX = ['park_factor', 'hr_pa', 'r_pa']
TUNE_SEASONS = [2016, 2017, 2018, 2019, 2020]
TRACK_MIN = 75 # min pa to be considered here

# prepare the count/multinomial data
mn_data_count = pa_data.copy()
mn_data_count['pa'] = mn_data_count[TARGS].sum(axis=1)
mn_data_count.rename(columns={'batter_id': 'bam_id'}, inplace=True)

# full statcast df
sc_df = tracking_df.\
```

```
  merge(sprint_speed_df, how='inner', left_on=['bam_id', 'season'],␣
↪right_on=['bam_id', 'season'])

combined_df = mn_data_count.\
    merge(sc_df, how='left', left_on=['bam_id', 'season'], right_on=['bam_id',␣
↪'season']).\
    merge(xstats_df, how='left', left_on=['bam_id', 'season'],␣
↪right_on=['bam_id', 'season']).\
    query(f'n_tracked >= {TRACK_MIN} & season >= 2016 & season < 2020')
```

1 Variational Bayes

1.0.1 Scratch Functions

First are the "by-hand" versions of the variational model. These are much slower and are comparatively crude in their optimization (crude gradient ascent/Newton). However, they are included for the sake of the exercise and understanding the computation going on under the hood. #### a.) Scratch Multinomial Logit GLM This is a standard multinomial logit GLM solver, intended for count data. Solvers include Newton and Nesterov.

[ ]:
```
def mnlpmf(y, p):
  kernel_ll = np.multiply(y, np.log(p)).sum(axis=1) # log-likelihood of␣
↪probability component
  choose_ll = gammaln(y.sum(axis=1) + 1) -  gammaln(y + 1).sum(axis=1) # log of␣
↪choose component
  return kernel_ll + choose_ll

def softmax(X, theta):
  logits = np.exp(X @ theta)
  pi = np.divide(logits, logits.sum(axis=1).reshape(-1, 1))
  return pi

class MultinomReg:
    """
    Sklearn-style function to perform multinomial-logit regression with count␣
↪data
    """
    def __init__(self, l2=.01):
        self.is_fit = False
        self.l2 = l2

    def _init_params_fit(self, X, y):
        self.X = X
        self.N, self.P = self.X.shape
        self.y = y
```

```python
        self.w = y.sum(axis=1)
        self.X = np.hstack([np.ones(self.N).reshape(-1, 1), self.X])
        # self.theta = multivariate_normal().rvs((self.X.shape[1], self.y.shape[1]))
        self.theta = np.zeros((self.X.shape[1], self.y.shape[1]))

    def _softmax(self, X=None):
        X = self.X if X is None else X
        logits = np.exp(X.dot(self.theta))
        pi = np.divide(logits, logits.sum(axis=1).reshape(-1, 1))
        return pi

    def _grad(self):
        pi = self._softmax()
        resid = np.add(
            self.y,
            -np.multiply(self.w.reshape(-1, 1), pi)
        )
        grad_lik = self.X.T.dot(resid)
        # grad_reg = 2 * self.l2 * self.theta
        grad_reg = np.vstack([np.zeros(self.theta.shape[1]), 2 * self.l2 * self.theta[1:, :]])
        return np.add(grad_lik/np.sum(self.w), -grad_reg)

    def _fit_gd(self, eta=.01, maxiter=10000, tol=1e-4):
        nabla_log, log_loss = [], [1e3, 1e-2]
        n_iter = 0
        self.maxiter = maxiter
        while n_iter <= maxiter and np.abs(log_loss[-1] - log_loss[-2]) > tol:
            nabla = eta * self._grad()
            nabla_log.append(nabla)
            self.theta = np.add(self.theta, nabla)
            # TODO: prior density here
            log_loss.append(np.sum(multinomial.logpmf(self.y, n=self.w, p=self._softmax())))
            n_iter += 1
        self.n_iter = n_iter
        self.nabla_log = nabla_log
        self.log_loss = np.array(log_loss)[2:]
        self.is_fit = True

    def _fit_nesterov(self, B=.1, eta=.01, maxiter=10000, tol=1e-3):
        log_loss = [1e3, 1e-2]
        n_iter = 0
        self.maxiter = maxiter
        v = [0]
        while n_iter <= maxiter and np.abs(log_loss[-1] - log_loss[-2]) > tol:
```

```
            # calculate change
            v_t = self.theta + eta * self._grad()
            # momentum EQ
            self.theta = v_t + B * (v_t - v[-1])
            # save v for next momentum
            v.append(v_t)
            # TODO: prior density here
            log_loss.append(np.sum(multinomial.logpmf(self.y, n=self.w, p=self._softmax())))
            n_iter += 1
        self.n_iter = n_iter
        self.log_loss = np.array(log_loss)[2:]
        self.is_fit = True

    def fit(self, X, y, method='SGD', **kwargs):
        self._init_params_fit(X, y)
        if method == 'nesterov':
          self._fit_nesterov(**kwargs)
        else:
          self._fit_gd(**kwargs)

    def _score_single_cv(self, X, y, nfolds=5, random_state=2020, **kwargs):
        """"""
        np.random.seed(random_state)
        indices = np.array(range(X.shape[0]))
        indices = np.random.choice(indices, len(indices), replace=False)
        folds = np.array_split(indices, nfolds)
        losses = 0
        k = 0
        for k in range(nfolds):
          # print(k)
          train_indices = np.hstack([folds[i] for i in range(nfolds) if i != k]).flatten()
          val_indices = folds[k]
          self.fit(X[train_indices, :], y[train_indices, :], **kwargs)
          pi_hat = self._softmax(X=np.hstack([np.ones(len(val_indices)).reshape(-1, 1), X[val_indices]]))
          val_loss = np.sum(mnlpmf(y[val_indices, :], pi_hat))
          losses += val_loss
        return losses

    def fit_cv(self, X, y, l2_seq=[1, .1, .01], nfolds=5, **kwargs):
        scores = []
        for l2 in l2_seq:
          self.l2 = l2
          scores.append(self._score_single_cv(X, y, nfolds=nfolds, **kwargs))
        # record scoring
```

```
        self.l2 = l2_seq[np.argmin(scores)]
        self.scores = scores
        self.l2_seq = l2_seq
        # refit
        self.fit(X, y, **kwargs)

    def plot_convergence(self):
        if self.is_fit:
            plt.plot(self.log_loss)
            plt.xlabel('Iteration')
            plt.ylabel('Log Likelihood')
            plt.show()
```

b.) From-Scratch for VB: ELBO Ascent + EM This fits the by hand variational models, both MAP + ELBO ascent and EM.

[ ]:
```
def grad_kl_diag_norm(mu_q, log_sigma2_q, mu_p, log_sigma2_p):
  """
  Computes D_{KL(q || p), where
    - q ~ N(u_n, \vec\sigma_{qn}^2 I); n = 1,...,N
    - p ~ N(u_p, \vec\sigma_p^2 I)
  """
  grad_mu_q = np.multiply(np.add(mu_q,  -mu_p), 1 / np.exp(log_sigma2_p))
  # grad_sigma_q = np.add(-1 / sigma2_q, 1 / sigma2_p)
  grad_log_sigma2_q = np.add(-np.ones(mu_q.shape),
                             np.divide(np.exp(log_sigma2_q), np.
↪exp(log_sigma2_p)))
  return grad_mu_q, 1/2 * grad_log_sigma2_q

def get_logp_reparam_mu(mu_q, sigma2_q, eps, y, theta, x_aux=None):
  z_n = np.hstack([
    np.ones(mu_q.shape[0]).reshape(-1, 1), # Nz
    np.add(mu_q, np.multiply(np.sqrt(sigma2_q), eps))
  ])
  z_n = np.hstack([z_n, x_aux]) if x_aux is not None else z_n
  pi_n = softmax(X=z_n, theta=theta)
  return mnlpmf(y, pi_n)

def get_logp_reparam_log_sigma2(log_sigma2_q, mu_q, eps, y, theta, x_aux=None):
  sigma2_q = np.exp(log_sigma2_q)
  z_n = np.hstack([
    np.ones(mu_q.shape[0]).reshape(-1, 1), # Nz
    np.add(mu_q, np.multiply(np.sqrt(sigma2_q), eps))
  ])
  z_n = np.hstack([z_n, x_aux]) if x_aux is not None else z_n
  pi_n = softmax(X=z_n, theta=theta)
```

```
    return mnlpmf(y, pi_n)

grad_mu_lpr = elementwise_grad(get_logp_reparam_mu)
grad_log_sigma2_lpr = elementwise_grad(get_logp_reparam_log_sigma2)

def _kldiv(mu_z, sigma2_z, mu_p, sigma2_p):
  kldivs_ = np.log(np.divide(
      np.prod(sigma2_p),
      sigma2_z.prod(axis=1)
    )) - mu_z.shape[1] + \
    (np.multiply(
        np.add(mu_z, -mu_p) ** 2,
        1 / sigma2_p
    )).sum(axis=1) + (np.multiply(1/sigma2_p, sigma2_z)).sum(axis=1)
  return .5 * kldivs_

def _log_density(z_n, theta, y):
  """Adds an intercept"""
  z_n_ = np.hstack([
      np.ones(z_n.shape[0]).reshape(-1, 1), # Nz
      z_n,
  ])
  pi_n = softmax(X=z_n_, theta=theta)
  logps = mnlpmf(y, pi_n)
  return logps

def _elbo(mu_z, log_sigma2_z, mu_p, log_sigma2_p, z_n, theta, y):
  sigma2_z, sigma2_p = np.exp(log_sigma2_z), np.exp(log_sigma2_p)
  kld = _kldiv(mu_z, sigma2_z, mu_p, sigma2_p)
  logps = _log_density(z_n, theta, y)
  return kld + logps

class ScratchSCVI:
  def __init__(self, xsup_sc, ysup, yunsup, xsup_aux_sc=None,␣
 ↪xunsup_aux_sc=None):
    self.x_sup = xsup_sc
    self.y_sup = ysup

    self.mu_p = self.x_sup.mean(axis=0)
    self.sigma2_p = self.x_sup.var(axis=0)
    self.log_sigma2_p = np.log(self.sigma2_p)

    self.mu_z = np.zeros((yunsup.shape[0], self.x_sup.shape[1]))
    self.sigma2_z = np.vstack([self.sigma2_p for i in range(self.mu_z.
 ↪shape[0])])
    self.log_sigma2_z = np.log(self.sigma2_z)
```

```
    self.x_sup_aux = xsup_aux_sc
    self.x_unsup_aux = xunsup_aux_sc

    self.y_unsup = yunsup

    self.theta = None

    self.elbo_history = []

  # def get_x_sup(self):
  #   return self.x_sup

  # def get_x_unsup(self):
  #   return self.mu_z

  def get_x_sup(self):
    if self.x_sup_aux is not None and self.x_unsup_aux is not None:
      return np.hstack([self.x_sup, self.x_sup_aux])
    else:
      return self.x_sup

  def get_x_unsup(self):
    if self.x_sup_aux is not None and self.x_unsup_aux is not None:
      return np.hstack([self.mu_z, self.x_unsup_aux])
    else:
      return self.mu_z

  def update_theta(self,
                   l2=.01,
                   eta=.05,
                   maxiter=50000,
                   tol=1e-3,
                   include_unsup=False,
                   plot_fit=False):
    model = MultinomReg(l2=l2)
    if include_unsup:
      X = np.vstack([self.get_x_sup(), self.get_x_unsup()])
      y = np.vstack([self.y_sup, self.y_unsup])
    else:
      X = self.get_x_sup()
      y = self.y_sup
    model.fit(X, y, eta=eta, maxiter=maxiter, tol=tol)
    self.theta = model.theta
    if plot_fit:
      model.plot_convergence()

  def update_varational_params(self, eta=.01, S=100, n_iter=500, verbose=True):
```

```
   range_func = trange if verbose else range
   for iter in range_func(n_iter):
     # gradient wrt KL
     grad_kl = grad_kl_diag_norm(
       mu_q=self.mu_z,
       log_sigma2_q=self.log_sigma2_z,
       mu_p=self.mu_p,
       log_sigma2_p=self.log_sigma2_p
     )

     eps_n = multivariate_normal().rvs(self.mu_z.shape + (S, ))

     grad_mu_z = np.stack([grad_mu_lpr(self.mu_z, self.sigma2_z, eps_n[:, :, j], self.y_unsup, self.theta, self.x_unsup_aux)
                           for j in range(eps_n.shape[-1])]).mean(axis=0)
     full_grad_mu = grad_kl[0] - grad_mu_z
     self.mu_z = self.mu_z - full_grad_mu * eta

     grad_log_sigma2_z = np.stack([grad_log_sigma2_lpr(self.log_sigma2_z, self.mu_z, eps_n[:, :, j], self.y_unsup, self.theta, self.x_unsup_aux)
                                   for j in range(eps_n.shape[-1])]).mean(axis=0)
     full_grad_log_sigma2 = grad_kl[1] - grad_log_sigma2_z

     self.log_sigma2_z = self.log_sigma2_z - full_grad_log_sigma2 * eta
     self.sigma2_z = np.exp(self.log_sigma2_z)

     if self.x_unsup_aux is None:
       self.elbo_history.append(
           np.sum(_elbo(self.mu_z, self.log_sigma2_z, self.mu_p, self.log_sigma2_p, self.mu_z, self.theta, self.y_unsup))
       )
     else:
       self.elbo_history.append(
           np.sum(_elbo(self.mu_z, self.log_sigma2_z, self.mu_p, self.log_sigma2_p,
                        np.hstack([self.mu_z, self.x_unsup_aux]), self.theta, self.y_unsup))
       )

 def gradient_ascent(self, l2, eta_theta=.05, eta_z=.025, S=100, n_iter=1000):
   self.update_theta(l2=l2, eta=eta_theta, maxiter=50000, tol=1e-5, include_unsup=False, plot_fit=True)
   self.update_varational_params(eta=eta_z, S=S, n_iter=n_iter, verbose=True)
```

```python
  def exp_max(self, l2, n_step=10, eta_theta=.05, eta_z=.025, S=100,␣
↪n_iter=100):
    self.update_theta(l2=l2, eta=eta_theta, maxiter=50000, tol=1e-3,␣
↪include_unsup=False, plot_fit=False)
    for i in range(n_step):
      print(i)
      self.update_varational_params(eta=eta_z, S=S, n_iter=n_iter, verbose=True)
      self.update_theta(l2=l2, eta=eta_theta, maxiter=50000, tol=1e-3,␣
↪include_unsup=True, plot_fit=False)

  def predict_z(self, mu=0, sd=1):
    """"""
    return self.mu_z * sd + mu

  def get_post_pred_interval(self, q=[.025, .5, .975], mu=0, sd=1):
    """"""
    result = []
    for i in range(self.mu_z.shape[0]):
      posterior_sample = multivariate_normal(self.mu_z[i, :], np.diag(self.
↪sigma2_z[i, :])).rvs(50000, random_state=2020) * sd + mu
      result.append(np.stack([np.quantile(posterior_sample, j, axis=0) for j in␣
↪q]))
    result = np.stack(result)
    return result

  def score_loss(self, ytrue):
    xhat = self.get_x_unsup()
    xhat = np.hstack([np.ones(xhat.shape[0]).reshape(-1, 1), xhat])
    yhat = softmax(xhat, self.theta)
    return np.sum(mnlpmf(ytrue, yhat)), np.sum(mnlpmf(ytrue, yhat))/np.
↪sum(ytrue)

  def plot_elbo_history(self):
    plt.plot(self.elbo_history)
    plt.xlabel('Iteration')
    plt.ylabel('ELBO')
    plt.show()


class TuneScratchSCVI:
  """
  Class for tuning
  """
  def __init__(self):
    self.scalers_x = {}
    self.scalers_aux = {}
```

```
    self.fit_results = {}
    self.cv_results_loss = []
    self.cv_results_sc = []
    self.cv_results_unsc = []
    self.combo_df = None
    self.tune_seasons = None
    self.x_features = None
    self.x_features_aux = None
    self.targs = None

  def tune(self,
           data_df,
           l2_seq=(.1, .01, .005, .001),
           tune_seasons=(2016, 2017, 2018),
           targs=TARGS,
           x_features=X_FEATURES,
           x_features_aux=None,
           method='gradient_ascent'):

    combo_df = data_df.copy().query(f"season >= {np.min(tune_seasons)} & season <= {np.max(tune_seasons)}")

    for l2 in l2_seq:
      cv_mses_sc, cv_mses_unsc, cv_losses, fits = [], [], [], {}

      for szn in [2018]:
        # extract a held out season
        combo_df_unsup = combo_df.loc[combo_df.season == szn, :]
        combo_df_sup = combo_df.loc[combo_df.season != szn, :]

        # processing
        xsup, ysup = combo_df_sup[x_features].values, combo_df_sup[targs].values
        scaler_xsup = StandardScaler()
        xsup_sc = scaler_xsup.fit_transform(xsup)

        yunsup = combo_df_unsup[targs].values
        xunsup_sc = np.zeros(combo_df_unsup[x_features].values.shape)

        # process auxiliary variables
        xsup_aux, xunsup_aux = combo_df_sup[x_features_aux].values, combo_df_unsup[x_features_aux].values
        scaler_aux = StandardScaler()
        xsup_aux_sc = scaler_aux.fit_transform(xsup_aux)
        xunsup_aux_sc = scaler_aux.transform(xunsup_aux)

        # save the scalers
        self.scalers_x[szn] = scaler_xsup
```

```
        self.scalers_aux[szn] = scaler_aux

        # fit the model
        fit_cv = SCVI(xsup_sc, ysup, yunsup, xsup_aux_sc=xsup_aux_sc,␣
↪xunsup_aux_sc=xunsup_aux_sc)
        if method == 'EM' or method == 'em':
          fit_cv.exp_max(l2=l2)
        else:
          fit_cv.gradient_ascent(l2=l2)

        # plot and save
        fit_cv.plot_elbo_history()
        fits[szn] = fit_cv

        # extract true results
        y_true_unsc = combo_df_unsup[x_features].values
        y_true_sc = scaler_xsup.transform(combo_df_unsup[x_features].values)

        # scoring
        cv_mse_unsc = ((y_true_unsc - fit_cv.predict_z(mu=scaler_xsup.mean_,␣
↪sd=scaler_xsup.scale_)) ** 2).sum(axis=0)
        cv_mse_sc = ((y_true_sc - fit_cv.mu_z) ** 2).sum(axis=0)
        cv_log_loss = fit_cv.score_loss(yunsup)

        # save
        cv_mses_sc.append(cv_mse_sc)
        cv_mses_unsc.append(cv_mse_unsc)
        cv_losses.append(cv_log_loss)

      self.cv_results_unsc.append(cv_mses_unsc)
      self.cv_results_sc.append(cv_mses_sc)
      self.cv_results_loss.append(cv_losses)
      self.fit_results[l2] = fits
      self.combo_df = combo_df
      self.x_features = x_features
      self.x_features_aux = x_features_aux
      self.targs = targs
      self.tune_seasons = tune_seasons
      self.yunsup = yunsup
```

1.1 Tensorflow Implementation

The code here achieves exactly that above – however, TFP and its associated libraries are leveraged for speed and performance.

Note we remake the classes, to avoid conflict between `autograd.numpy` and regular `numpy`.

[ ]:
```
def make_batch(x, y, batch_size=32):
  result_batch = tf.data.Dataset.from_tensor_slices((x, y)).batch(batch_size)
  return result_batch

def elbo_loss(y_true, vi_approx, p_dist, prior_dist, alpha=1):
  """
  x_true = batch of data examples
  - output of encoder, a distributional MVN object
  -output of decoder, independent bernoulli object
  """
  return tf.reduce_mean(
      tfd.kl_divergence(vi_approx, prior_dist) * tf.constant(np.float32(alpha))
        - p_dist.log_prob(y_true)
  )

@tf.function
def compute_gradient(q_z_mu, q_z_log_sigma2, ytrue, w, p_model, prior_dist,␣
↪supervised_data=None, alpha=1000):
  """
  supervised_data should be (x, y, PA)
  """
  with tf.GradientTape() as tape:
    q_z = tfd.MultivariateNormalDiag(loc=q_z_mu, scale_diag=tf.math.
↪exp(q_z_log_sigma2))
    q_z_sample = q_z.sample()
    pi_hat = p_model(q_z_sample)
    # tfkl doesn't exist, so we have to build ourselves
    p_multinom = tfd.Multinomial(total_count=w, probs=pi_hat)
    # compute loss
    loss_sample = elbo_loss(
        y_true=ytrue,
        vi_approx=q_z,
        p_dist=p_multinom,
        prior_dist=prior_dist,
        alpha=alpha
    )
    if supervised_data is not None:
      pi_hat_sup = p_model(supervised_data[0])
      p_multinom_sup = tfd.Multinomial(total_count=supervised_data[2],␣
↪probs=pi_hat_sup)
      loss_sample = loss_sample - tf.reduce_mean(p_multinom_sup.
↪log_prob(supervised_data[1]))

  if supervised_data is not None:
    grads = tape.gradient(target=loss_sample, sources=[q_z_mu,␣
↪q_z_log_sigma2]+p_model.trainable_variables)
  else:
```

```
    grads = tape.gradient(target=loss_sample, sources=[q_z_mu, q_z_log_sigma2])
  return loss_sample, grads

class TensorflowSCVI:
  def __init__(self, xsup_sc, ysup, yunsup, xsup_aux_sc=None,␣
↪xunsup_aux_sc=None):
    self.x_sup = xsup_sc
    self.y_sup = ysup

    self.mu_p = self.x_sup.mean(axis=0)
    self.sigma2_p = self.x_sup.var(axis=0)
    self.log_sigma2_p = np.log(self.sigma2_p)

    self.mu_z = np.zeros((yunsup.shape[0], self.x_sup.shape[1]))
    self.sigma2_z = np.vstack([self.sigma2_p for i in range(self.mu_z.
↪shape[0])])
    self.log_sigma2_z = np.log(self.sigma2_z)
    # TODO implement the auxiliaries
    self.x_sup_aux = xsup_aux_sc
    self.x_unsup_aux = xunsup_aux_sc

    self.y_unsup = yunsup
    self.model = None

  def get_x_sup(self):
    if self.x_sup_aux is not None and self.x_unsup_aux is not None:
      return np.hstack([self.x_sup, self.x_sup_aux])
    else:
      return self.x_sup

  def get_x_unsup(self):
    if self.x_sup_aux is not None and self.x_unsup_aux is not None:
      return np.hstack([self.mu_z, self.x_unsup_aux])
    else:
      return self.mu_z

  def update_theta(self,
                   l2=.01,
                   eta=.05,
                   maxiter=1000,
                   include_unsup=False,
                   plot_fit=False):

    if include_unsup:
      X = np.vstack([self.get_x_sup(), self.get_x_unsup()])
      y = np.vstack([self.y_sup, self.y_unsup])
    else:
```

```
        X = self.get_x_sup()
        y = self.y_sup
      # horizontal dims
      K, L = y.shape[1], X.shape[1]
      model = Sequential([Dense(K,
                                activation='softmax',
                                input_shape=(L, ),
                                kernel_regularizer=tf.keras.regularizers.l2(l2),
                                bias_regularizer=tf.keras.regularizers.l2(l2)
                                )])
      model.compile(optimizer='adam', loss='categorical_crossentropy')
      model.fit(X,
                np.divide(y, np.sum(y, axis=1).reshape(-1,1)), sample_weight=y.
↪sum(axis=1), epochs=maxiter, verbose=0)
      self.model = model

  def _obtain_solution(self, semi_supervised=False, maxiter=10000, l2=.01,␣
↪alpha=1000):
      # this is for the unsupervised part
      N, L = self.y_unsup.shape[0], self.get_x_sup().shape[1]
      prior_z = tfd.MultivariateNormalDiag(loc=tf.zeros(L))
      ytrue = self.y_unsup.astype('float32')
      w = ytrue.sum(axis=1)
      q_z_mu = tf.Variable(tf.zeros((N, L)), trainable=True)
      q_z_log_sigma2 = tf.Variable(tf.zeros((N, L)), trainable=True)

      # prepare to sample
      opt = tf.keras.optimizers.Adam()
      elbo_history = []

      if semi_supervised:
        # bundle supervised component
        supervised_data = (self.x_sup.astype('float32'),
                           self.y_sup.astype('float32'),
                           self.y_sup.astype('float32').sum(axis=1))
        # make model
        self.model = Sequential([Dense(self.y_sup.shape[1],
                                       activation='softmax',
                                       input_shape=(L, ),
                                       kernel_regularizer=tf.keras.regularizers.
↪l2(l2),
                                       bias_regularizer=tf.keras.regularizers.
↪l2(l2))])
        self.model.compile(optimizer='adam', loss='categorical_crossentropy')

        # fit
        for epoch in trange(maxiter):
```

```
        # TODO: implement batching for larger datasets
        loss, grads = compute_gradient(q_z_mu, q_z_log_sigma2, ytrue, w, self.
↪model, prior_z, supervised_data, alpha=alpha)
        elbo_history.append(loss.numpy())
        opt.apply_gradients(zip(
            grads,
            [q_z_mu, q_z_log_sigma2] + self.model.trainable_variables
        ))

    else:
      # assume self.model has already been fitted
      for epoch in trange(maxiter):
        # TODO: implement batching for larger datasets
        loss, grads = compute_gradient(q_z_mu, q_z_log_sigma2, ytrue, w, self.
↪model, prior_z, alpha=alpha)
        elbo_history.append(loss.numpy())
        opt.apply_gradients(zip(
            grads,
            [q_z_mu, q_z_log_sigma2]
        ))
    self.mu_z = q_z_mu.numpy()
    self.log_sigma2_z = q_z_log_sigma2.numpy()
    self.sigma2_z = np.exp(self.log_sigma2_z)
    self.elbo_history = -np.array(elbo_history) * self.y_unsup.shape[0]

  # def gradient_ascent(self, maxiter=10000):
  #   N, L = self.y_unsup.shape[0], self.get_x_sup().shape[1]
  #   prior_z = tfd.MultivariateNormalDiag(loc=tf.zeros(L))
  #   ytrue = self.y_unsup.astype('float32')
  #   w = ytrue.sum(axis=1)
  #   q_z_mu = tf.Variable(tf.zeros((N, L)), trainable=True)
  #   q_z_log_sigma2 = tf.Variable(tf.zeros((N, L)), trainable=True)

  #   opt = tf.keras.optimizers.Adam()
  #   elbo_history = []
  #   for epoch in trange(maxiter):
  #     # TODO: implement batching for larger datasets
  #     loss, grads = compute_gradient(q_z_mu, q_z_log_sigma2, ytrue, w, self.
↪model, prior_z)
  #     elbo_history.append(loss.numpy())
  #     opt.apply_gradients(zip(
  #         grads,
  #         [q_z_mu, q_z_log_sigma2]
  #     ))
  #   self.mu_z = q_z_mu.numpy()
  #   self.log_sigma2_z = q_z_log_sigma2.numpy()
  #   self.sigma2_z = np.exp(self.log_sigma2_z)
```

```
#   self.elbo_history = -np.array(elbo_history) * self.y_unsup.shape[0]

def exp_max(self, maxiter=10000):
  pass

def predict_z(self, mu=0, sd=1):
  """"""
  return self.mu_z * sd + mu

def get_post_pred_interval(self, q=[.025, .5, .975], mu=0, sd=1, S=50000):
  """"""
  result = []
  for i in range(self.mu_z.shape[0]):
    posterior_sample = multivariate_normal(self.mu_z[i, :], np.diag(self.
↪sigma2_z[i, :])).rvs(S, random_state=2020) * sd + mu
    result.append(np.stack([np.quantile(posterior_sample, j, axis=0) for j in␣
↪q]))
  result = np.stack(result)
  return result

def score_loss(self, ytrue):
  xhat = self.get_x_unsup()
  xhat = np.hstack([np.ones(xhat.shape[0]).reshape(-1, 1), xhat])
  yhat = softmax(xhat, self.theta)
  return np.sum(mnlpmf(ytrue, yhat)), np.sum(mnlpmf(ytrue, yhat))/np.
↪sum(ytrue)

def plot_elbo_history(self):
  plt.rcParams['figure.figsize'] = [5, 3]
  plt.plot(self.elbo_history)
  plt.xlabel('Iteration')
  plt.ylabel('ELBO')
  plt.show()
```

1.2 Partitions

This chunk of code is devoted to data partitioning, splitting the train (2016-2018) set vs. the test (2019) set. You can call the tuning functions above, or just write a loop, to tune on 2018.

[ ]:
```
x_features=X_FEATURES
x_features_aux=X_FEATURES_AUX
targs=TARGS

combo_df_unsup = combined_df.query('season == 2019')
```

```
combo_df_sup = combined_df.query('season >= 2016 & season <= 2018')

# processing
xsup, ysup = combo_df_sup[x_features].values, combo_df_sup[targs].values
scaler_xsup = StandardScaler()
xsup_sc = scaler_xsup.fit_transform(xsup)

yunsup = combo_df_unsup[targs].values
xunsup_sc = np.zeros(combo_df_unsup[x_features].values.shape)

# process auxiliary variables
xsup_aux, xunsup_aux = combo_df_sup[x_features_aux].values,␣
↪combo_df_unsup[x_features_aux].values
scaler_aux = StandardScaler()
xsup_aux_sc = scaler_aux.fit_transform(xsup_aux)
xunsup_aux_sc = scaler_aux.transform(xunsup_aux)
```

1.3 Fit + Validate Gradient Ascent on 2019 (TF)

Optimal tune settings plugged in already. First, we fit both of the models.

[ ]:
```
mod_tf_ga = TensorflowSCVI(xsup_sc, ysup, yunsup)
mod_tf_ga.update_theta(maxiter=1000, l2=.005)
mod_tf_ga._obtain_solution(maxiter=20000, alpha=1)

mod_tf_em = TensorflowSCVI(xsup_sc, ysup, yunsup)
mod_tf_em._obtain_solution(maxiter=20000, semi_supervised=True, l2=.001,␣
↪alpha=1)
```

We then plot the ELBOs vs. iterations

[ ]:
```
mod_tf_ga.plot_elbo_history()
mod_tf_em.plot_elbo_history()
```

We then do some RMSE and WMAE scoring, albeit weighted by PA.

[ ]:
```
# Basic error scoring

def _wrmse(yhat, df_test, scaler):
  residuals = df_test[X_FEATURES].values - (yhat * scaler.scale_ + scaler.mean_)
  wresiduals = residuals ** 2 * df_test.pa_tbf.values.reshape(-1, 1)
  wrmse = np.sqrt(wresiduals.sum(axis=0)/ np.sum(df_test.pa_tbf.values))
  return wrmse

def _wmae(yhat, df_test, scaler):
  residuals = df_test[X_FEATURES].values - (yhat * scaler.scale_ + scaler.mean_)
  wresiduals = np.abs(residuals)  * df_test.pa_tbf.values.reshape(-1, 1)
```

```
  mae = wresiduals.sum(axis=0)/ np.sum(df_test.pa_tbf.values)
  return mae

def _wresid(yhat, df_test, scaler):
  residuals = df_test[X_FEATURES].values - (yhat * scaler.scale_ + scaler.mean_)
  wresiduals = residuals  * df_test.pa_tbf.values.reshape(-1, 1)
  wresid = wresiduals.sum(axis=0) / np.sum(df_test.pa_tbf.values)
  return wresid

# def get_vi_wrmse(fit, df_test, scaler):
#   return _wrmse(fit.mu_z, df_test, scaler)
#   # residuals = df_test[X_FEATURES].values - (fit.mu_z * scaler.scale_ +␣
 ↪scaler.mean_)
#   # wresiduals = residuals ** 2 * df_test.pa_tbf.values.reshape(-1, 1)
#   # wrmse = np.sqrt(wresiduals.sum(axis=0)/ np.sum(df_test.pa_tbf.values))
#   # return wrmse

# def get_vi_wmae(fit, df_test, scaler):
#   return _wmae(fit.mu_z, )
#   # residuals = df_test[X_FEATURES].values - (fit.mu_z * scaler.scale_ +␣
 ↪scaler.mean_)
#   # wresiduals = np.abs(residuals)  * df_test.pa_tbf.values.reshape(-1, 1)
#   # mae = wresiduals.sum(axis=0)/ np.sum(df_test.pa_tbf.values)
#   # return mae

# def get_vi_wresid(fit, df_test, scaler):
#   residuals = df_test[X_FEATURES].values - (fit.mu_z * scaler.scale_ + scaler.
 ↪mean_)
#   wresiduals = residuals  * df_test.pa_tbf.values.reshape(-1, 1)
#   wresid = wresiduals.sum(axis=0) / np.sum(df_test.pa_tbf.values)
#   return wresid

result_ga = np.vstack(
    [_wrmse(mod_tf_ga.mu_z, combined_df.query('season == 2019'), scaler_xsup),
     _wmae(mod_tf_ga.mu_z, combined_df.query('season == 2019'), scaler_xsup),
     _wresid(mod_tf_ga.mu_z, combined_df.query('season == 2019'), scaler_xsup)])
result_ga = pd.DataFrame(result_ga, columns=X_FEATURES, index=['wrmse', 'wmae',␣
 ↪'wbias'])
print(result_ga)


result_em = np.vstack(
    [_wrmse(mod_tf_em.mu_z, combined_df.query('season == 2019'), scaler_xsup),
     _wmae(mod_tf_em.mu_z, combined_df.query('season == 2019'), scaler_xsup),
     _wresid(mod_tf_em.mu_z, combined_df.query('season == 2019'), scaler_xsup)])
result_em = pd.DataFrame(result_em, columns=X_FEATURES, index=['wrmse', 'wmae',␣
 ↪'wbias'])
```

```
print(result_em)
```

Posterior Predictive Checks: We then do posterior predictive checks, with 90% PIs.

```
[ ]: # ppc's
def post_pred_check_vi(fit, df_test, scaler, **kwargs):
  post_pred_intervals = fit.get_post_pred_interval(mu=scaler.mean_, sd=scaler.
↪scale_, **kwargs)
  pp_lwr, pp_upr = post_pred_intervals[:, 0, :], post_pred_intervals[:, 2, :]
  return ((df_test[X_FEATURES].values >= pp_lwr) * (df_test[X_FEATURES].values␣
↪<= pp_upr)).mean(axis=0)

print(post_pred_check_vi(mod_tf_ga, combined_df.query('season == 2019'),␣
↪scaler_xsup, q=[.05, .5, .95]))
print(post_pred_check_vi(mod_tf_em, combined_df.query('season == 2019'),␣
↪scaler_xsup, q=[.05, .5, .95]))
```

```
[ ]: # scatters
def get_posterior_predictive(fit, df_test, scaler, S=100000, **kwargs):
  result = []
  sd, mu = scaler.scale_, scaler.mean_
  for i in range(fit.mu_z.shape[0]):
    posterior_sample = multivariate_normal(fit.mu_z[i, :], np.diag(fit.
↪sigma2_z[i, :])).rvs(S, random_state=2020) * sd + mu
    result.append(posterior_sample)
  result = np.stack(result)
  return result

pp_ga = get_posterior_predictive(mod_tf_ga, combined_df.query('season ==␣
↪2019'), scaler_xsup)
pp_em = get_posterior_predictive(mod_tf_em, combined_df.query('season ==␣
↪2019'), scaler_xsup)
```

For fun, we plot Hoskins, Knapp, Harper, and Hernandez against their posterior predictives.

```
[ ]: def plot_select_players(samples, select_df, figsize=[11, 4.5]):
  plt.rcParams['figure.figsize'] = figsize
  for j in range(samples.shape[-1]):
    fig, ax = plt.subplots(1, 1)
    binz = np.linspace(min(combined_df[X_FEATURES[j]].values),
                       max(combined_df[X_FEATURES[j]].values) + 1e-10,
                       130)
    mat = np.zeros((130-1, samples.shape[0]))
    obs_vals = np.zeros(samples.shape[0])

    for i in range(samples.shape[0]):
      prop = np.array([np.mean(
```

```
        (samples[i, :, j] >= binz[z]) & (samples[i, :, j] < binz[z + 1])
        ) for z in range(len(binz) - 1)])
      mat[:, i] = prop
      obs_vals[i] = select_df[X_FEATURES[j]].iloc[i]
    mat = pd.DataFrame(mat, columns=select_df.name.to_list())
    img = ax.imshow(mat, interpolation='nearest', aspect='auto',
                extent=[-.5, samples.shape[0]-.5, min(binz), max(binz)],␣
 ↪origin='lower')
    ax.set_xticks(list(range(samples.shape[0])))
    ax.set_xticklabels(select_df.name.to_list(), fontsize=14)
    ax.scatter(range(samples.shape[0]), obs_vals, color='red', marker='o',␣
 ↪s=200)
    plt.yticks(fontsize=14)
    plt.ylabel(X_FEATURES[j])
    plt.colorbar(img)
    plt.show()

selected_names = ['Bryce Harper', 'Andrew Knapp', 'Cesar Hernandez',  'Rhys␣
 ↪Hoskins']
select_df = combined_df.query('season == 2019').loc[combined_df.query('season␣
 ↪== 2019').name.isin(selected_names), :]
select_idx = np.where(combined_df.query('season == 2019').name.
 ↪isin(selected_names))

select_pp_ga, select_pp_em  = pp_ga[select_idx[0], :, :], pp_em[select_idx[0], :
 ↪, :]
# GA method I
plot_select_players(select_pp_ga, select_df, figsize=[16, 6])
```

[ ]:
```
plot_select_players(select_pp_em, select_df, figsize=[16, 6])
```

Bayesian Neural Net Below is code for the Bayesian Neural Net, fit through `pymc3`. Note that I spun up a version for Tensorflow, though this is not used in the report

The `process_nn()` function processed data for the network; what follows are the `tf` and `pymc3` fit classes.

Feature scaling: since the activation is `tanh()`, we scale the inputs (multinomial outcomes now) to $[-1, 1]$.

[ ]:
```
x_unsc = combined_df.query(f'season != {season_holdout}')[targs]
w = combined_df.query(f'season != {season_holdout}').pa_tbf.values
x_unsc.loc[:, tuple(targs)] = np.divide(x_unsc[targs].values, w.reshape(-1, 1))
y_unsc = combined_df.query(f'season != {season_holdout}')[x_features]

feature_scaler, target_scaler = MinMaxScaler(), StandardScaler()
x_sc = feature_scaler.fit_transform(x_unsc.values) * 2  - 1
y_sc = target_scaler.fit_transform(y_unsc.values)
```

```
xhat_unsc = combined_df.query(f'season == {season_holdout}')[targs]
what = combined_df.query(f'season == {season_holdout}').pa_tbf.values
xhat_unsc.loc[:, tuple(targs)] = np.divide(xhat_unsc[targs].values, what.
↪reshape(-1, 1))
yhat_unsc = combined_df.query(f'season == {season_holdout}')[x_features]

xhat_sc = feature_scaler.transform(xhat_unsc.values) * 2 - 1
yhat_sc = target_scaler.transform(yhat_unsc.values)
```

[ ]:
```
import theano.tensor as tt

def process_nn(combined_df, season_holdout, targs=TARGS, x_features=X_FEATURES):
  x_unsc = combined_df.query(f'season != {season_holdout}')[targs]
  w = combined_df.query(f'season != {season_holdout}').pa_tbf.values
  x_unsc.loc[:, tuple(targs)] = np.divide(x_unsc[targs].values, w.reshape(-1,␣
↪1))
  y_unsc = combined_df.query(f'season != {season_holdout}')[x_features]

  feature_scaler, target_scaler = MinMaxScaler(), StandardScaler()
  x_sc = feature_scaler.fit_transform(x_unsc.values) * 2  - 1
  y_sc = target_scaler.fit_transform(y_unsc.values)


  xhat_unsc = combined_df.query(f'season == {season_holdout}')[targs]
  what = combined_df.query(f'season == {season_holdout}').pa_tbf.values
  xhat_unsc.loc[:, tuple(targs)] = np.divide(xhat_unsc[targs].values, what.
↪reshape(-1, 1))
  yhat_unsc = combined_df.query(f'season == {season_holdout}')[x_features]

  xhat_sc = feature_scaler.transform(xhat_unsc.values) * 2 - 1
  yhat_sc = target_scaler.transform(yhat_unsc.values)

  return dict(
      df=combined_df,
      x_sc=x_sc,
      y_sc=y_sc,
      x_unsc=x_unsc,
      y_unsc=y_unsc,
      w=w,
      feature_scaler=feature_scaler,
      target_scaler=target_scaler,
      what=what,
      xhat_sc=xhat_sc,
      yhat_sc=yhat_sc
  )
```

```python
class tfFFNN:
  def __init__(self):
    self.model = None

  def build(self, X, y, arch, l2_theta, l2_beta, activation='relu',␣
 ↪loss='mean_squared_error'):
    input_layer = Input(shape=(X.shape[-1], ))
    ctr = 0
    for lyr in arch:
      if ctr == 0:
        hidden_layer = Dense(lyr,
                             activation=activation,
                             kernel_regularizer=l2(l2_theta[ctr]),
                             bias_regularizer=l2(l2_beta[ctr]))(input_layer)
      else:
        hidden_layer = Dense(lyr,
                             activation=activation,
                             kernel_regularizer=l2(l2_theta[ctr]),
                             bias_regularizer=l2(l2_beta[ctr]))(hidden_layer)
      ctr += 1
    output_layer = Dense(y.shape[-1],
                         kernel_regularizer=l2(l2_theta[ctr]),
                         bias_regularizer=l2(l2_beta[ctr]))(hidden_layer)

    model = Model(inputs=[input_layer],
                  outputs=[output_layer])
    model.compile(optimizer='adam',
                  loss=loss,
                  metrics=[loss])
    self.model = model

  def fit(self, **kwargs):
    if self.model is not None:
      self.model.fit(**kwargs)

  def get_opt_epochs(self):
    return np.argmin(self.model.history.history['val_loss']) + 1

  def get_val_loss(self, loss_key='val_mean_squared_error'):
    return np.min(self.model.history.history[loss_key])

class BayesFFNN:
  def __init__(self, X, y, hidden=10):
    with pm.Model() as net:
      net_in, net_out = pm.Data('input', X), pm.Data('output', y)
```

```
    # hyperpriors
    sigma_1_ = pm.InverseGamma('sigma1_prior', alpha=1, beta=1)
    sigma_2_ = pm.InverseGamma('sigma2_prior', alpha=1, beta=1)
    sigma_out = pm.InverseGamma('sigma_prior', alpha=1, beta=1)

    sigma_bias1_ = pm.InverseGamma('sigma1_bias_ prior', alpha=1, beta=1)
    sigma_bias2_ = pm.InverseGamma('sigma2_bias_prior', alpha=1, beta=1)

    theta_1 = pm.Normal("theta1", 0, sigma=pm.math.sqrt(sigma_1_), shape=(X.shape[1], hidden))
    theta_out = pm.Normal("theta_out", 0, sigma=pm.math.sqrt(sigma_2_), shape=(hidden, y.shape[1]))

    alpha_1 = pm.Normal('alpha1', 0, sigma=pm.math.sqrt(sigma_bias1_), shape=hidden)
    alpha_2 = pm.Normal('alpha2', 0, sigma=pm.math.sqrt(sigma_bias2_), shape=y.shape[1])

    # layer_hidden = tt.nnet.relu(pm.math.dot(net_in, theta_1) + alpha_1)
    layer_hidden = pm.math.tanh(pm.math.dot(net_in, theta_1) + alpha_1)
    layer_out = pm.math.dot(layer_hidden, theta_out) + alpha_2

    output = pm.Normal('output_norm', mu=layer_out, sigma=pm.math.sqrt(sigma_out), shape=(y.shape[1]), observed=net_out, total_size=X.shape[0])
  self.model = net

def _fit_approx(self, n=100000):
  with self.model:
    self.inference = pm.ADVI()
    self.approximation = pm.fit(n=n, method=self.inference)

def _plot_elbo_fit_approx(self):
  plt.rcParams['figure.figsize'] = [5, 5]
  plt.plot(-self.inference.hist)
  plt.ylabel("ELBO")
  plt.xlabel("Iteration")
  plt.title('ELBO vs. Iteration')

def _fit_nuts(self, n=500):
  with self.model:
    self.trace_nuts = pm.sample(n, return_inferencedata=False)

def _sample_approx_posterior_predictive(self, Xhat, yhat, ndraw=1000, ndraw_trace=5000):

  trace = self.approximation.sample(draws=ndraw_trace)
```

```
    pm.set_data(new_data={"input": Xhat, "output": yhat}, model=self.model)
    post_pred_sample = pm.sample_posterior_predictive(
        trace, samples=ndraw, progressbar=True, model=self.model
    )
    return post_pred_sample['output_norm']
```

[ ]:
```
nn_data = process_nn(combined_df, season_holdout=2019)
mod_nn = BayesFFNN(nn_data['x_sc'], nn_data['y_sc'], hidden=10)
mod_nn._fit_approx(n=100000)
```

These steps are effectively a repeat of the above, albeit for the neural net.

[ ]:
```
mod_nn._plot_elbo_fit_approx()
```

[ ]:
```
# sample posterior predictive
yhat_nn_pp = mod_nn._sample_approx_posterior_predictive(
    nn_data['xhat_sc'],
    nn_data['yhat_sc'],
    ndraw=10000,
    ndraw_trace=100000)
```

[ ]:
```
# basic scoring
result_nn = np.vstack(
    [_wrmse(yhat_nn_pp.mean(axis=0), combined_df.query('season == 2019'),␣
 ↪nn_data['target_scaler']),
     _wmae(yhat_nn_pp.mean(axis=0), combined_df.query('season == 2019'),␣
 ↪nn_data['target_scaler']),
     _wresid(yhat_nn_pp.mean(axis=0), combined_df.query('season == 2019'),␣
 ↪nn_data['target_scaler'])])
result_nn = pd.DataFrame(np.round(result_nn, 3), columns=X_FEATURES,␣
 ↪index=['wrmse', 'wmae', 'wbias'])
result_nn
```

[ ]:
```
# posterior predictive checks

pp_lwr, pp_upr = np.quantile(yhat_nn_pp, .05, axis=0), np.quantile(yhat_nn_pp, .
 ↪95, axis=0)
pp_lwr = pp_lwr * nn_data['target_scaler'].scale_ + nn_data['target_scaler'].
 ↪mean_
pp_upr = pp_upr * nn_data['target_scaler'].scale_ + nn_data['target_scaler'].
 ↪mean_
capture_rates = ((combined_df.query('season == 2019')[X_FEATURES].values >=␣
 ↪pp_lwr) * (combined_df.query('season == 2019')[X_FEATURES].values <=␣
 ↪pp_upr)).mean(axis=0)
np.round(capture_rates, 3)
```

```
[ ]: plot_select_players(
         nn_data['target_scaler'].inverse_transform(yhat_nn_pp[:, select_idx[0], :]).
      ↪swapaxes(0, 1),
         select_df,
         figsize=[16, 6])
```