**G26: Financial Forecasting using Bayesian algorithms in Pyro**

**Problem Statement:**

We are trying to predict the future closing prices of Ethereum for the next 30 days based on the previous 2.5 years closing prices. Traditional financial forecasting methods tend to overfit, requiring complex regularization and extensive data. This project addresses the challenge by employing Pyro, a probabilistic programming framework. Embracing probabilistic perspectives, we aim to enhance reliability, achieve better regularization, and reduce data needs, revolutionizing financial forecasting while focusing on practical applications over intricate theoretical complexities. Our dataset in particular pertains to historical prices of a cryptocurrency (Ethereum).

**Software architecture:**

The solution has a layered architecture:

1. Data Retrieval and Preprocessing: This layer involves fetching Ethereum price data from a database, performing preprocessing steps like data cleaning, normalization, and creating input-output pairs for the predictive models.

2. Modelling: This layer involves building Bayesian based Pyro machine learning models to predict Ethereum prices based on historical data.

3. Training and Evaluation: Models are trained using a subset of the data and evaluated on another subset. The evaluation involves assessing performance metrics like mean square error, mean absolute error, and plotting predictions against actual values for visual analysis.

4. Inference and Forecasting: Once the models are trained, they're utilized for making future price predictions using the test set or unseen data. For this, we use 70% of the data to train our model and the rest 30% to test our results.

The testing component is local, performed within the code environment or on the local machine. The database holds Ethereum price data, fetched from historical records.

The architecture follows a typical pipeline: data preparation, model development, training, evaluation, and forecasting, typical in machine learning projects. The modular nature of the code is such that it can be extended with additional models, evaluation methods, or data sources for more comprehensive analysis and forecasting.

**PoPl aspects:**

1) Split - Train - Test

We use 70% of the dataset in training our model and the next 30% to test it

Our code snippet:

from sklearn.model\_selection import TimeSeriesSplit

from keras.callbacks import ReduceLROnPlateau, ModelCheckpoint

# Define callbacks for reducing learning rate and model checkpoint

reduce\_lr = ReduceLROnPlateau(monitor='val\_loss', factor=0.9, patience=10, min\_lr=0.000001, verbose=0)

checkpointer = ModelCheckpoint(filepath="testtest.hdf5", verbose=0, save\_best\_only=True)

# Splitting data into training and testing sets

X\_train, Y\_train = input\_data[:-30], output\_data[:-30]

X\_test, Y\_test = input\_data[-30:], output\_data[-30:]

# Creating the model using the previously defined function

model = create\_simple\_model(len(X\_train[0]))

# Fitting the model with training data, validating on test data

history = model.fit(X\_train, Y\_train,

epochs=100,

batch\_size=64,

verbose=1,

validation\_data=(X\_test, Y\_test),

callbacks=[reduce\_lr, checkpointer],

shuffle=True)

2) Using the concept of hidden layers

Our code snippet:

import torch

import torch.nn as nn

class Net(torch.nn.Module):

def init(self, n\_feature, n\_hidden):

super(Net, self).init()

self.hidden = torch.nn.Linear(n\_feature, n\_hidden) # hidden layer

self.predict = torch.nn.Linear(n\_hidden, 1) # output layer

def forward(self, x):

x = self.hidden(x)

x = self.predict(x)

return x

first\_layer = len(X\_train[0])

second\_layer = 25

softplus = nn.Softplus()

regression\_model = Net(first\_layer, second\_layer)

3) Using the concept of optimization

Our code snippet:

optimizer = Adam({"lr": 0.001})

svi = SVI(model, guide, optimizer, loss="ELBO")

num\_samples = len(X\_train)

for epoch in range(3000):

total\_loss = 0.0

permutation = torch.randperm(num\_samples)

# shuffle data

shuffled\_data = data[permutation]

# get batch indices

all\_batches = get\_batch\_indices(num\_samples, 64)

for idx, batch\_start in enumerate(all\_batches[:-1]):

batch\_end = all\_batches[idx + 1]

batch\_data = shuffled\_data[batch\_start: batch\_end]

total\_loss += svi.step(batch\_data)

if epoch % 100 == 0:

print(epoch, "average loss {}".format(total\_loss / float(num\_samples)))

4) We are using a Normal Bernoulli distribution

Our code snippet:

import pyro

from pyro.distributions import Normal, Bernoulli # noqa: F401

from pyro.infer import SVI

from pyro.optim import Adam

pyro.get\_param\_store().clear()

5) Using the concept of mean square error and collectivity

Our code snippet:

model.load\_weights('testtest.hdf5')

plt.plot(pyro.param('guide\_mean\_weight').data.numpy()[10])

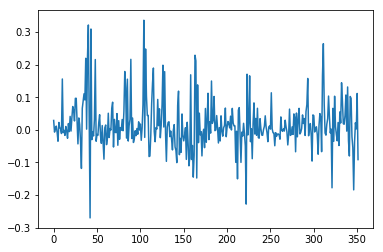
**Potential for future work:**

1. Model Enhancements: Explore different neural network architectures (LSTM, GRU, etc.) or Pyro models for Bayesian regression to enhance prediction accuracy.

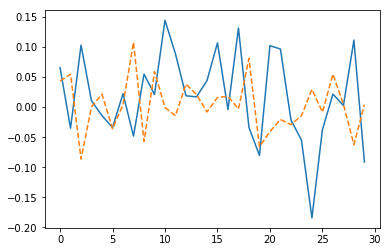
2. Deployability: Focus on model deployment strategies, such as converting models to lighter formats (like TensorFlow Lite) or deploying them as APIs for real-time predictions.

3. Continuous Monitoring and Model Updates: Implement a system to monitor model performance in production and update the model periodically with new data to maintain accuracy.

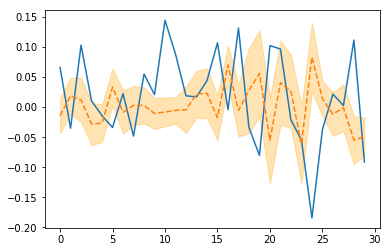
**Results:**

****

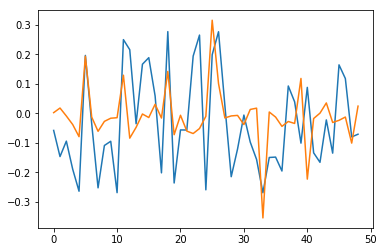
This graph shows the actual variation of our dataset with time.



This graph shows the prediction using Python. As we can see, the results are not regularized (orange dotted line). This is based on a definite deterministic model.



This is based on a probabilistic model. Here, the orange area shows the different probabilities of prediction with the dotted line being the most probable.



This graph shows the mean-weighted sum of the result through our probabilistic model. The blue line represents the actual data whereas the orange one is the deterministic prediction results.