Configuration Manual

MSc Research Project
Data Analytics

Aleksandra Kalisz
Student ID: x21118876

School of Computing
National College of Ireland

Supervisor: Dr. Catherine Mulwa

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Aleksandra Kalisz
Student ID:	x21118876
Programme:	Data Analytics
Year:	2018
Module:	MSc Research Project
Supervisor:	Dr. Catherine Mulwa
Submission Due Date:	20/12/2018
Project Title:	Configuration Manual
Word Count:	648
Page Count:	15

Signature:	Aleksandra Kalisz
Date:	15th September 2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Aleksandra Kalisz
x21118876

1 Introduction

The Configuration Manual is a complete guide to the reproduction of the research project on "Uncertainty Analysis in Earthquake Prediction using Deep Learning for Improved Risk Management." This documentation is to provide detailed instructions on the configuration of software and hardware, the libraries required, and important sections of the code that must be followed for project reproduction. These guidelines can help researchers replicate the procedures outlined in the study and future research on its results.

2 System Requirements

2.1 Hardware Configuration

Processor	Dual-Core Intel Core i5
Speed	2.6 GHz
Ram	Minimum 8GB

Table 1: System Specifications

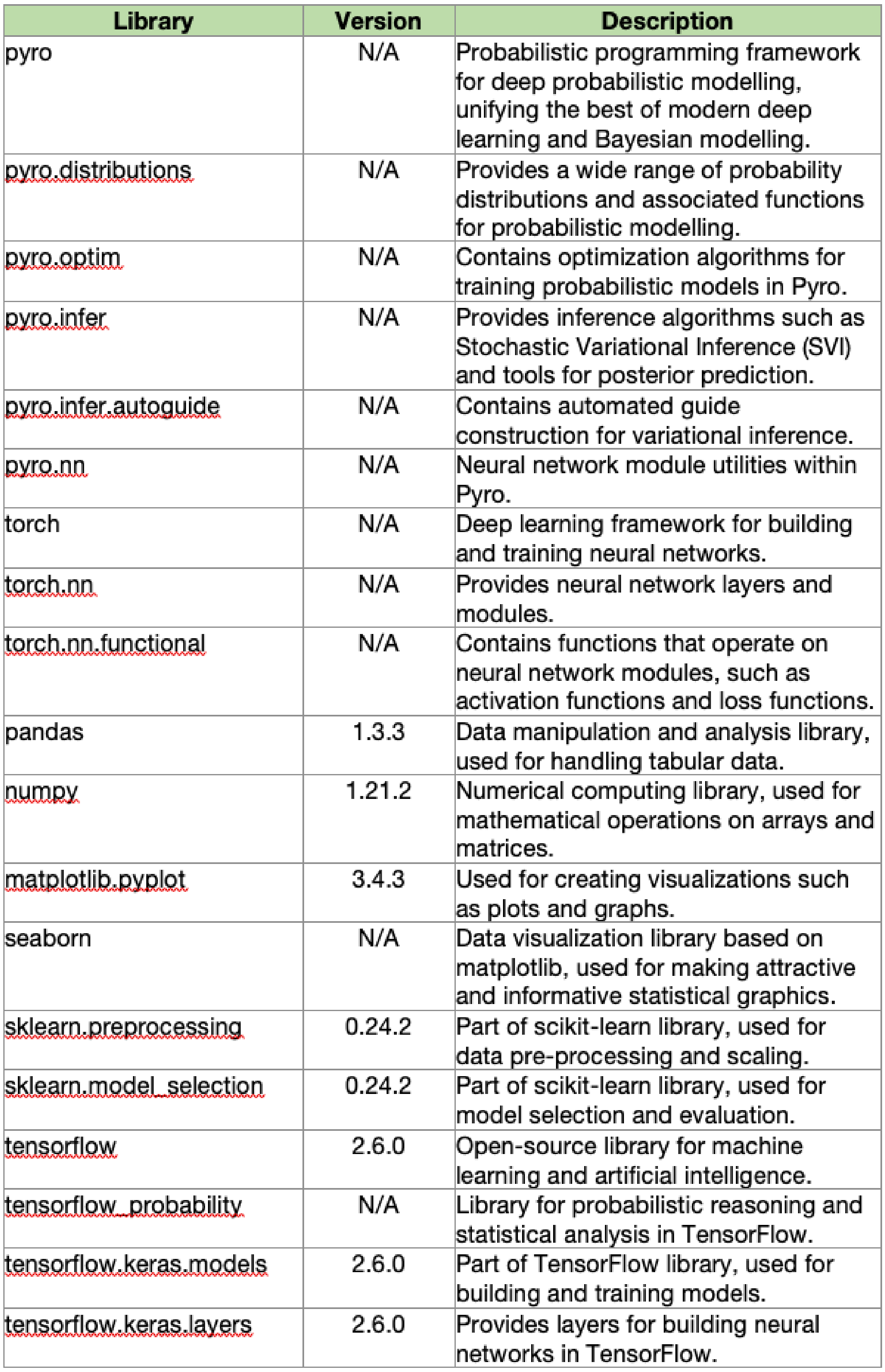
2.2 Software Configuration

Programming Language	**Python**	**V 3.7**
Notebook	**Jupyter Notebook**	**V 6.4.12**
Platform	**TensorFlow**	**V 2.16.1**

Table 2: Software Specifications

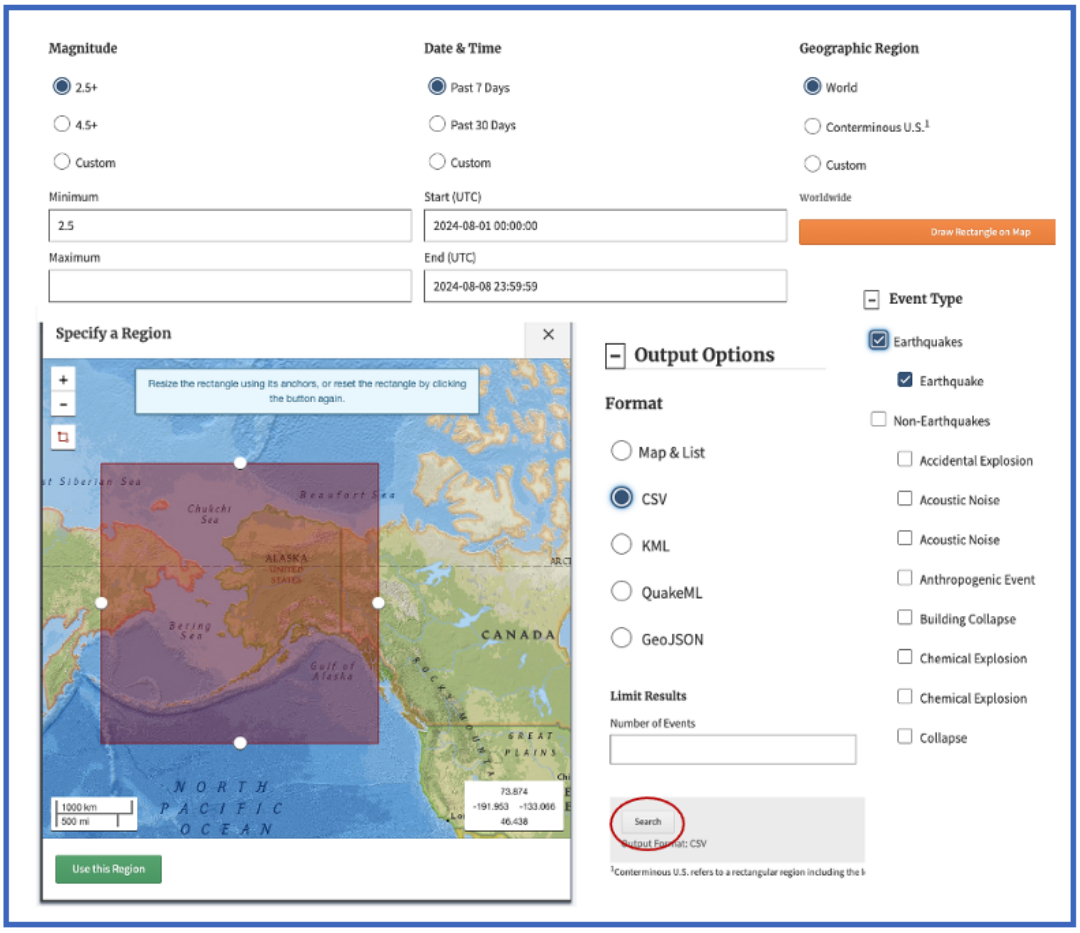
3 The Data

The project was developed using Python in the Jupyter Notebook environment, using the set of libraries (Figure 1)



Library	Version	Description
pyro	N/A	Probabilistic programming framework for deep probabilistic modelling, unifying the best of modern deep learning and Bayesian modelling.
pyro.distributions	N/A	Provides a wide range of probability distributions and associated functions for probabilistic modelling.
pyro.optim	N/A	Contains optimization algorithms for training probabilistic models in Pyro.
pyro.infer	N/A	Provides inference algorithms such as Stochastic Variational Inference (SVI) and tools for posterior prediction.
pyro.infer.autoguide	N/A	Contains automated guide construction for variational inference.
pyro.nn	N/A	Neural network module utilities within Pyro.
torch	N/A	Deep learning framework for building and training neural networks.
torch.nn	N/A	Provides neural network layers and modules.
torch.nn.functional	N/A	Contains functions that operate on neural network modules, such as activation functions and loss functions.
pandas	1.3.3	Data manipulation and analysis library, used for handling tabular data.
numpy	1.21.2	Numerical computing library, used for mathematical operations on arrays and matrices.
matplotlib.pyplot	3.4.3	Used for creating visualizations such as plots and graphs.
seaborn	N/A	Data visualization library based on matplotlib, used for making attractive and informative statistical graphics.
sklearn.preprocessing	0.24.2	Part of scikit-learn library, used for data pre-processing and scaling.
sklearn.model_selection	0.24.2	Part of scikit-learn library, used for model selection and evaluation.
tensorflow	2.6.0	Open-source library for machine learning and artificial intelligence.
tensorflow_probability	N/A	Library for probabilistic reasoning and statistical analysis in TensorFlow.
tensorflow.keras.models	2.6.0	Part of TensorFlow library, used for building and training models.
tensorflow.keras.layers	2.6.0	Provides layers for building neural networks in TensorFlow.

Figure 1: Libraries Table.


Magnitude

[✓ 2.5+] 4.5+ Custom

Minimum: 2.5

Maximum:

Date & Time

[✓ Past 7 Days] Past 30 Days Custom

Start (UTC): 2024-08-01 00:00:00

End (UTC): 2024-08-08 23:59:59

Geographic Region

[✓ World] Conterminous U.S.[1] Custom

Worldwide

Draw Rectangle on Map

Specify a Region

Resize the rectangle using its anchors, or reset the rectangle by clicking the button again.

73.874
-191.953 -133.066
46.438

Use this Region

Output Options

Format

Map & List [✓ CSV] KML QuakeML GeoJSON

Limit Results

Number of Events:

Search

Output Format: CSV

[1]Conterminous U.S. refers to a rectangular region including the l

Event Type

[✓ Earthquakes]

[✓ Earthquake]

Non-Earthquakes

Accidental Explosion

Acoustic Noise

Acoustic Noise

Anthropogenic Event

Building Collapse

Chemical Explosion

Chemical Explosion

Collapse

Figure 2: Data Search Map.

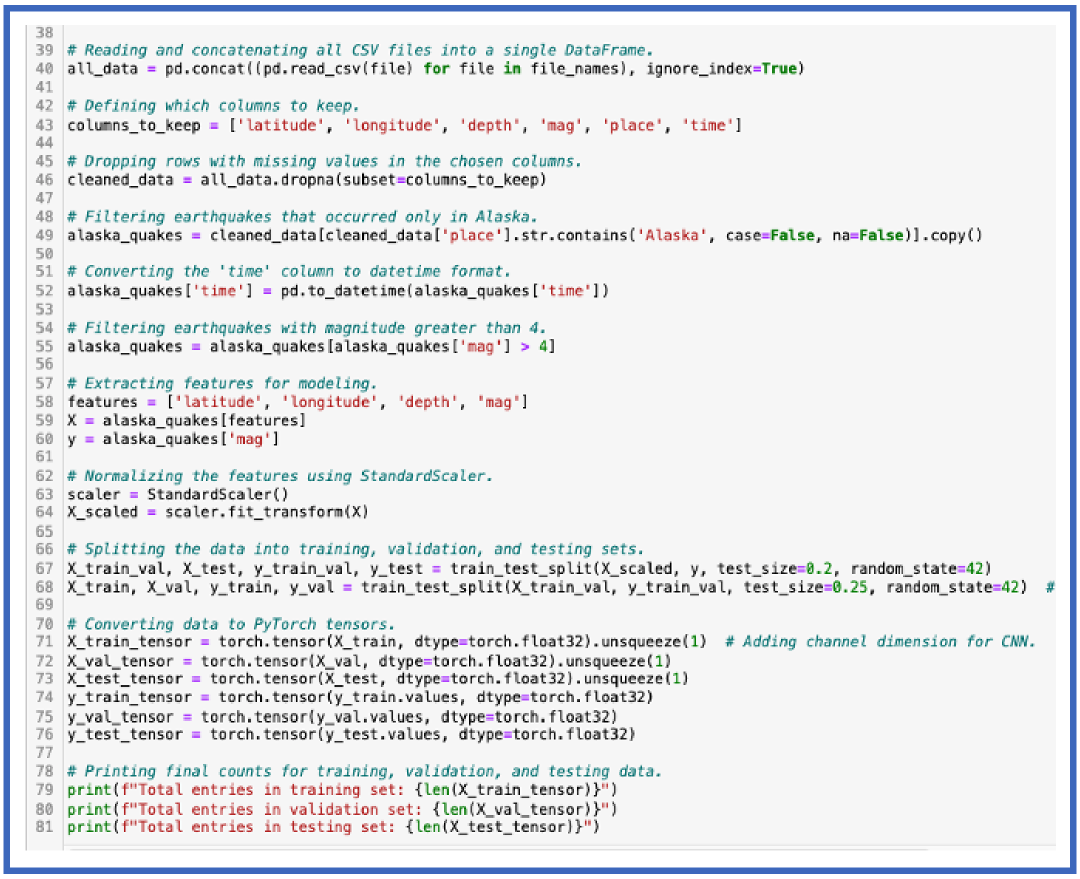
3.1 Data Download

Data is collected for this project from an official website of the United States Geological Survey: at USGS Earthquake Search website -link- .

This web page (Figure 2) gives us the option to customise the search area and dates of historical data. For this project, data were downloaded from January 2004 to December 2023, collecting 102.228 observations.

3.2 Data Pre-processing

Data was cleaned and pre-processed for all the models, including selecting relevant columns, dropping missing values, filtering, normalising data, as splitting into train, validation and train, as well as converted to PyTorch tensors (Fig 3). Count of data after pre-processing: 6850. Data was split 60/20/20, total entries for training set: 4110, validation: 1370, testing: 1370.



```python

# Reading and concatenating all CSV files into a single DataFrame.
all_data = pd.concat((pd.read_csv(file) for file in file_names), ignore_index=True)

# Defining which columns to keep.
columns_to_keep = ['latitude', 'longitude', 'depth', 'mag', 'place', 'time']

# Dropping rows with missing values in the chosen columns.
cleaned_data = all_data.dropna(subset=columns_to_keep)

# Filtering earthquakes that occurred only in Alaska.
alaska_quakes = cleaned_data[cleaned_data['place'].str.contains('Alaska', case=False, na=False)].copy()

# Converting the 'time' column to datetime format.
alaska_quakes['time'] = pd.to_datetime(alaska_quakes['time'])

# Filtering earthquakes with magnitude greater than 4.
alaska_quakes = alaska_quakes[alaska_quakes['mag'] > 4]

# Extracting features for modeling.
features = ['latitude', 'longitude', 'depth', 'mag']
X = alaska_quakes[features]
y = alaska_quakes['mag']

# Normalizing the features using StandardScaler.
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Splitting the data into training, validation, and testing sets.
X_train_val, X_test, y_train_val, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val, test_size=0.25, random_state=42)  #

# Converting data to PyTorch tensors.
X_train_tensor = torch.tensor(X_train, dtype=torch.float32).unsqueeze(1)  # Adding channel dimension for CNN.
X_val_tensor = torch.tensor(X_val, dtype=torch.float32).unsqueeze(1)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32).unsqueeze(1)
y_train_tensor = torch.tensor(y_train.values, dtype=torch.float32)
y_val_tensor = torch.tensor(y_val.values, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test.values, dtype=torch.float32)

# Printing final counts for training, validation, and testing data.
print(f"Total entries in training set: {len(X_train_tensor)}")
print(f"Total entries in validation set: {len(X_val_tensor)}")
print(f"Total entries in testing set: {len(X_test_tensor)}")
```

Figure 3: Data Pre-processing.

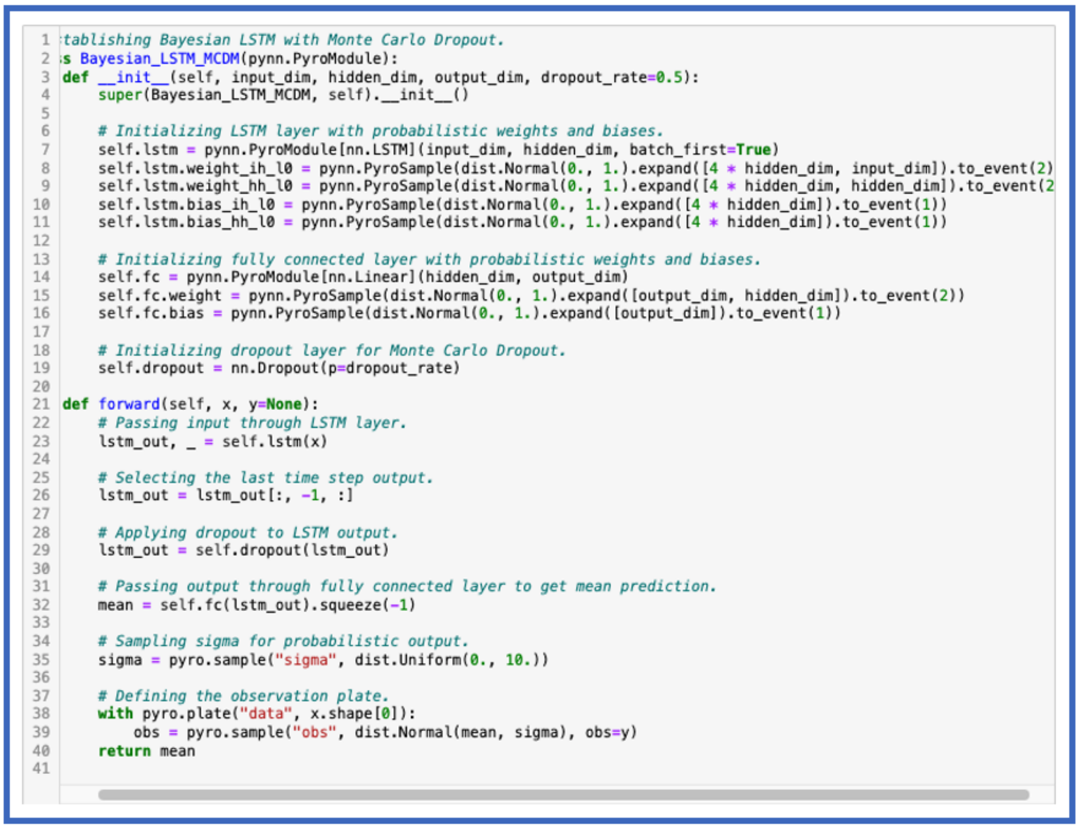
4 Model Implementation

Four primary models were developed: Long Short-Term Memory (LSTM) network, Convolutional Neural Network (CNN), Temporal Convolutional Memory (TCM) and Hybrid model of CNN/LSTM.

4.1 Bayesian LSTM Model

The below code defines a Bayesian LSTM Model with Monte Carlo Dropout (Figure 4). The input data is passed through the LSTM, then the last output passes through the dropout layer, which randomly drops some connections during training. Finally, data passes through a fully connected layer to product predictions.

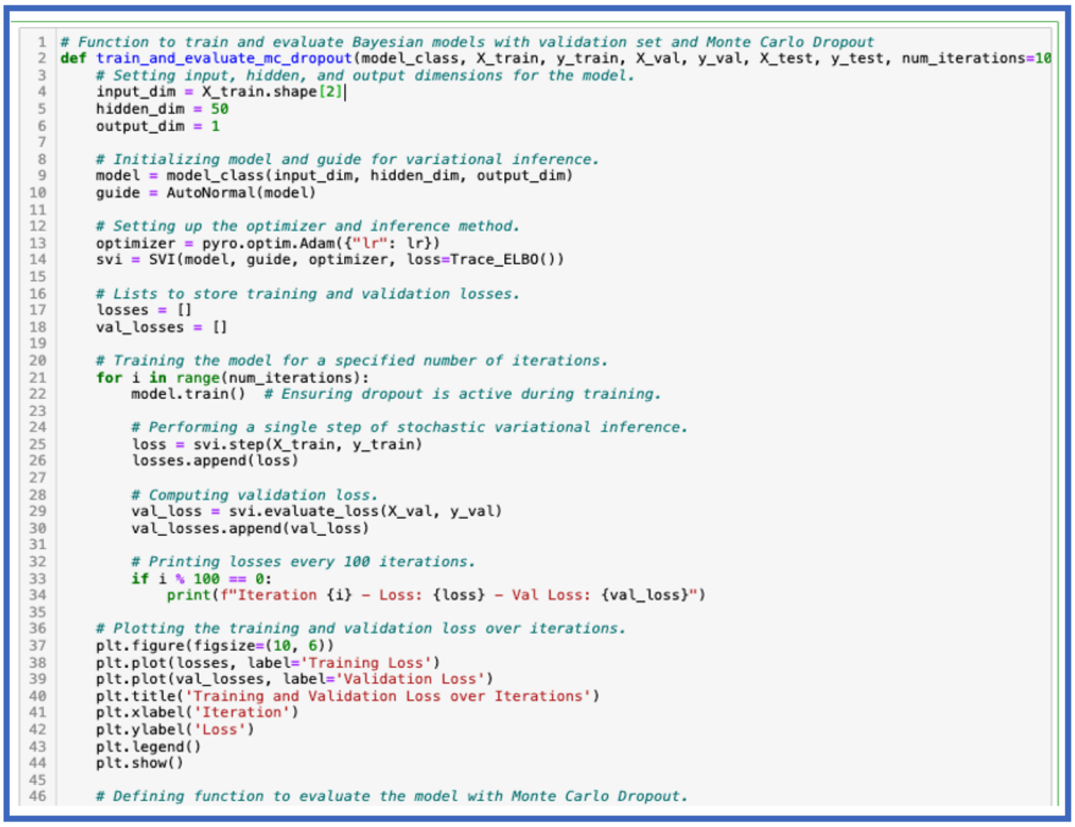
Code was researched at(link).



```
tablishing Bayesian LSTM with Monte Carlo Dropout.
s Bayesian_LSTM_MCDM(pynn.PyroModule):
def __init__(self, input_dim, hidden_dim, output_dim, dropout_rate=0.5):
    super(Bayesian_LSTM_MCDM, self).__init__()

    # Initializing LSTM layer with probabilistic weights and biases.
    self.lstm = pynn.PyroModule[nn.LSTM](input_dim, hidden_dim, batch_first=True)
    self.lstm.weight_ih_l0 = pynn.PyroSample(dist.Normal(0., 1.).expand([4 * hidden_dim, input_dim]).to_event(2)
    self.lstm.weight_hh_l0 = pynn.PyroSample(dist.Normal(0., 1.).expand([4 * hidden_dim, hidden_dim]).to_event(2
    self.lstm.bias_ih_l0 = pynn.PyroSample(dist.Normal(0., 1.).expand([4 * hidden_dim]).to_event(1))
    self.lstm.bias_hh_l0 = pynn.PyroSample(dist.Normal(0., 1.).expand([4 * hidden_dim]).to_event(1))

    # Initializing fully connected layer with probabilistic weights and biases.
    self.fc = pynn.PyroModule[nn.Linear](hidden_dim, output_dim)
    self.fc.weight = pynn.PyroSample(dist.Normal(0., 1.).expand([output_dim, hidden_dim]).to_event(2))
    self.fc.bias = pynn.PyroSample(dist.Normal(0., 1.).expand([output_dim]).to_event(1))

    # Initializing dropout layer for Monte Carlo Dropout.
    self.dropout = nn.Dropout(p=dropout_rate)

def forward(self, x, y=None):
    # Passing input through LSTM layer.
    lstm_out, _ = self.lstm(x)

    # Selecting the last time step output.
    lstm_out = lstm_out[:, -1, :]

    # Applying dropout to LSTM output.
    lstm_out = self.dropout(lstm_out)

    # Passing output through fully connected layer to get mean prediction.
    mean = self.fc(lstm_out).squeeze(-1)

    # Sampling sigma for probabilistic output.
    sigma = pyro.sample("sigma", dist.Uniform(0., 10.))

    # Defining the observation plate.
    with pyro.plate("data", x.shape[0]):
        obs = pyro.sample("obs", dist.Normal(mean, sigma), obs=y)
    return mean
```

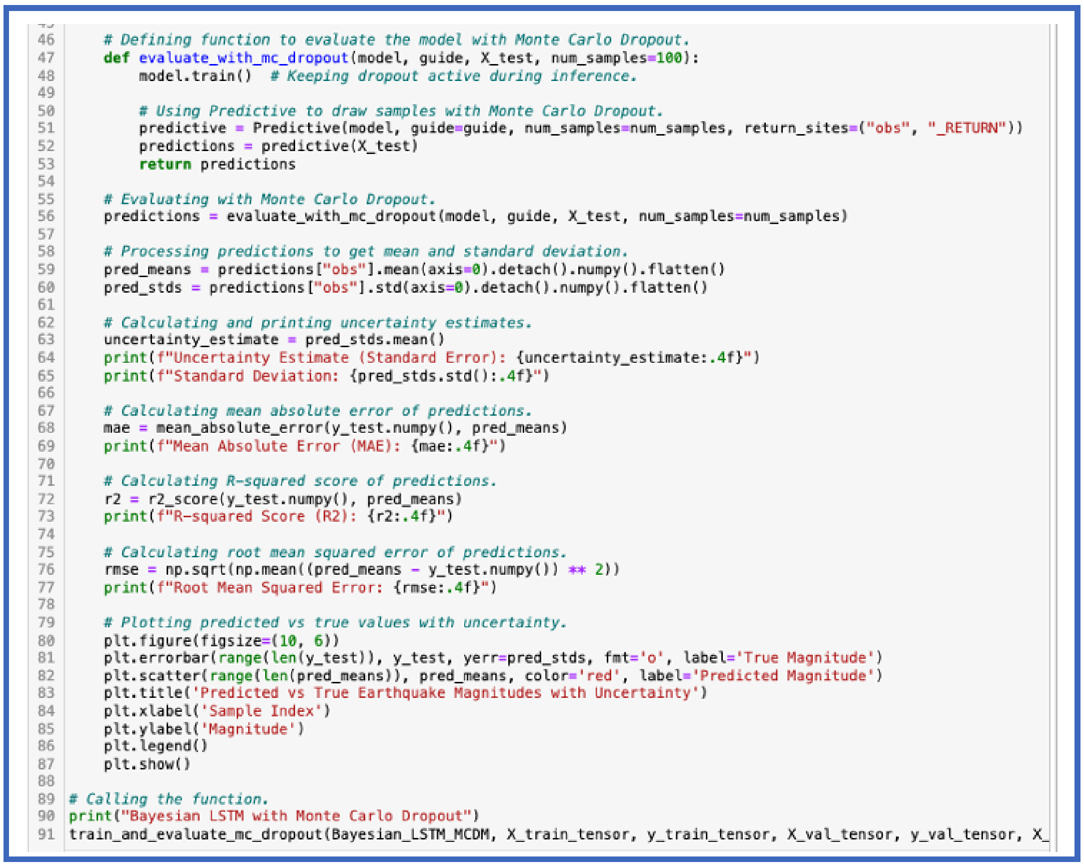
Figure 4: Bayesian LSTM Model Definition

The function train and evaluate (Figure 5) and (Figure 6) begin by setting models dimensions, initialising models guide, setting up optimiser and inference method. The model is trained and validated, evaluated with Monte Carlo Dropout, predictions and performance metrics are calculated.



```python
# Function to train and evaluate Bayesian models with validation set and Monte Carlo Dropout
def train_and_evaluate_mc_dropout(model_class, X_train, y_train, X_val, y_val, X_test, y_test, num_iterations=10
    # Setting input, hidden, and output dimensions for the model.
    input_dim = X_train.shape[2]
    hidden_dim = 50
    output_dim = 1

    # Initializing model and guide for variational inference.
    model = model_class(input_dim, hidden_dim, output_dim)
    guide = AutoNormal(model)

    # Setting up the optimizer and inference method.
    optimizer = pyro.optim.Adam({"lr": lr})
    svi = SVI(model, guide, optimizer, loss=Trace_ELBO())

    # Lists to store training and validation losses.
    losses = []
    val_losses = []

    # Training the model for a specified number of iterations.
    for i in range(num_iterations):
        model.train()  # Ensuring dropout is active during training.

        # Performing a single step of stochastic variational inference.
        loss = svi.step(X_train, y_train)
        losses.append(loss)

        # Computing validation loss.
        val_loss = svi.evaluate_loss(X_val, y_val)
        val_losses.append(val_loss)

        # Printing losses every 100 iterations.
        if i % 100 == 0:
            print(f"Iteration {i} - Loss: {loss} - Val Loss: {val_loss}")

    # Plotting the training and validation loss over iterations.
    plt.figure(figsize=(10, 6))
    plt.plot(losses, label='Training Loss')
    plt.plot(val_losses, label='Validation Loss')
    plt.title('Training and Validation Loss over Iterations')
    plt.xlabel('Iteration')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()

    # Defining function to evaluate the model with Monte Carlo Dropout.
```

Figure 5: Train and Validate

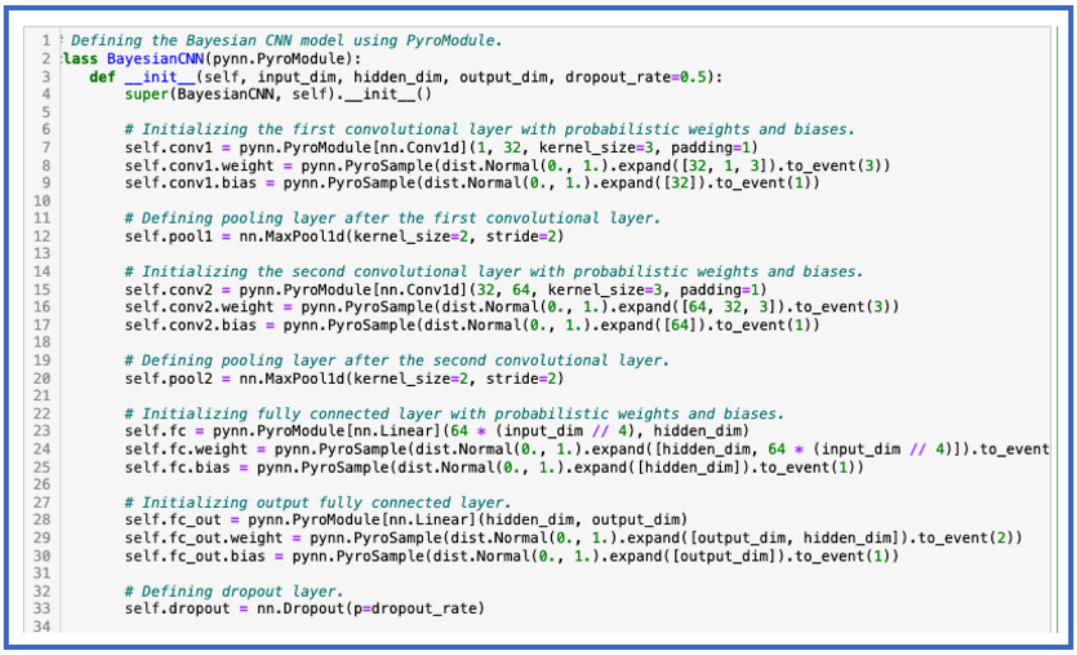


```python
    # Defining function to evaluate the model with Monte Carlo Dropout.
    def evaluate_with_mc_dropout(model, guide, X_test, num_samples=100):
        model.train()  # Keeping dropout active during inference.

        # Using Predictive to draw samples with Monte Carlo Dropout.
        predictive = Predictive(model, guide=guide, num_samples=num_samples, return_sites=("obs", "_RETURN"))
        predictions = predictive(X_test)
        return predictions

    # Evaluating with Monte Carlo Dropout.
    predictions = evaluate_with_mc_dropout(model, guide, X_test, num_samples=num_samples)

    # Processing predictions to get mean and standard deviation.
    pred_means = predictions["obs"].mean(axis=0).detach().numpy().flatten()
    pred_stds = predictions["obs"].std(axis=0).detach().numpy().flatten()

    # Calculating and printing uncertainty estimates.
    uncertainty_estimate = pred_stds.mean()
    print(f"Uncertainty Estimate (Standard Error): {uncertainty_estimate:.4f}")
    print(f"Standard Deviation: {pred_stds.std():.4f}")

    # Calculating mean absolute error of predictions.
    mae = mean_absolute_error(y_test.numpy(), pred_means)
    print(f"Mean Absolute Error (MAE): {mae:.4f}")

    # Calculating R-squared score of predictions.
    r2 = r2_score(y_test.numpy(), pred_means)
    print(f"R-squared Score (R2): {r2:.4f}")

    # Calculating root mean squared error of predictions.
    rmse = np.sqrt(np.mean((pred_means - y_test.numpy()) ** 2))
    print(f"Root Mean Squared Error: {rmse:.4f}")

    # Plotting predicted vs true values with uncertainty.
    plt.figure(figsize=(10, 6))
    plt.errorbar(range(len(y_test)), y_test, yerr=pred_stds, fmt='o', label='True Magnitude')
    plt.scatter(range(len(pred_means)), pred_means, color='red', label='Predicted Magnitude')
    plt.title('Predicted vs True Earthquake Magnitudes with Uncertainty')
    plt.xlabel('Sample Index')
    plt.ylabel('Magnitude')
    plt.legend()
    plt.show()

# Calling the function.
print("Bayesian LSTM with Monte Carlo Dropout")
train_and_evaluate_mc_dropout(Bayesian_LSTM_MCDM, X_train_tensor, y_train_tensor, X_val_tensor, y_val_tensor, X_
```

Figure 6: Bayesian LSTM Model Definition

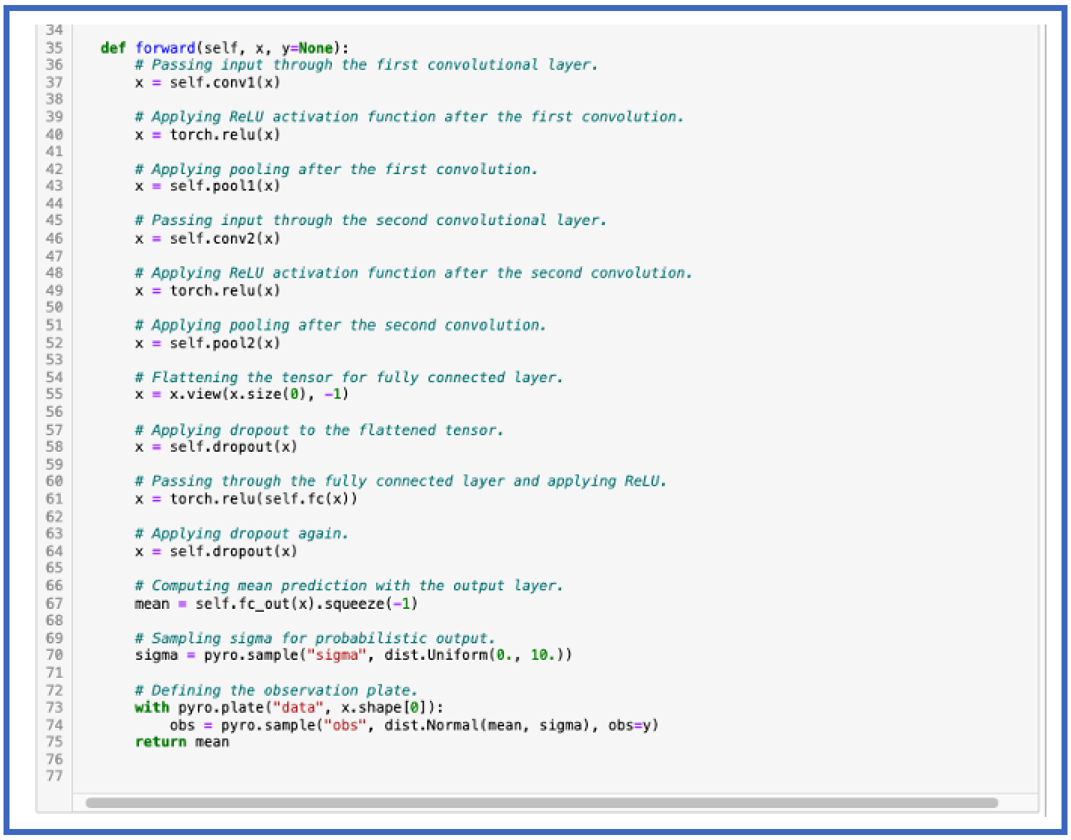
4.2 Bayesian CNN Model

This section shows CNN Model (Figure 7) being defined, first Convolutional layer is being initialised and probabilistic weights and biases are set. After data passes through first Pooling Layer, second Convolutional Layer and second Pooling layer until reaches Fully Connected Layer and Output Layer. Forward Method (Figure 8)specifies how the data will be processed through the Neural Network Layer to generate the results.



```python
 Defining the Bayesian CNN model using PyroModule.
lass BayesianCNN(pynn.PyroModule):
    def __init__(self, input_dim, hidden_dim, output_dim, dropout_rate=0.5):
        super(BayesianCNN, self).__init__()

        # Initializing the first convolutional layer with probabilistic weights and biases.
        self.conv1 = pynn.PyroModule[nn.Conv1d](1, 32, kernel_size=3, padding=1)
        self.conv1.weight = pynn.PyroSample(dist.Normal(0., 1.).expand([32, 1, 3]).to_event(3))
        self.conv1.bias = pynn.PyroSample(dist.Normal(0., 1.).expand([32]).to_event(1))

        # Defining pooling layer after the first convolutional layer.
        self.pool1 = nn.MaxPool1d(kernel_size=2, stride=2)

        # Initializing the second convolutional layer with probabilistic weights and biases.
        self.conv2 = pynn.PyroModule[nn.Conv1d](32, 64, kernel_size=3, padding=1)
        self.conv2.weight = pynn.PyroSample(dist.Normal(0., 1.).expand([64, 32, 3]).to_event(3))
        self.conv2.bias = pynn.PyroSample(dist.Normal(0., 1.).expand([64]).to_event(1))

        # Defining pooling layer after the second convolutional layer.
        self.pool2 = nn.MaxPool1d(kernel_size=2, stride=2)

        # Initializing fully connected layer with probabilistic weights and biases.
        self.fc = pynn.PyroModule[nn.Linear](64 * (input_dim // 4), hidden_dim)
        self.fc.weight = pynn.PyroSample(dist.Normal(0., 1.).expand([hidden_dim, 64 * (input_dim // 4)]).to_event
        self.fc.bias = pynn.PyroSample(dist.Normal(0., 1.).expand([hidden_dim]).to_event(1))

        # Initializing output fully connected layer.
        self.fc_out = pynn.PyroModule[nn.Linear](hidden_dim, output_dim)
        self.fc_out.weight = pynn.PyroSample(dist.Normal(0., 1.).expand([output_dim, hidden_dim]).to_event(2))
        self.fc_out.bias = pynn.PyroSample(dist.Normal(0., 1.).expand([output_dim]).to_event(1))

        # Defining dropout layer.
        self.dropout = nn.Dropout(p=dropout_rate)
```

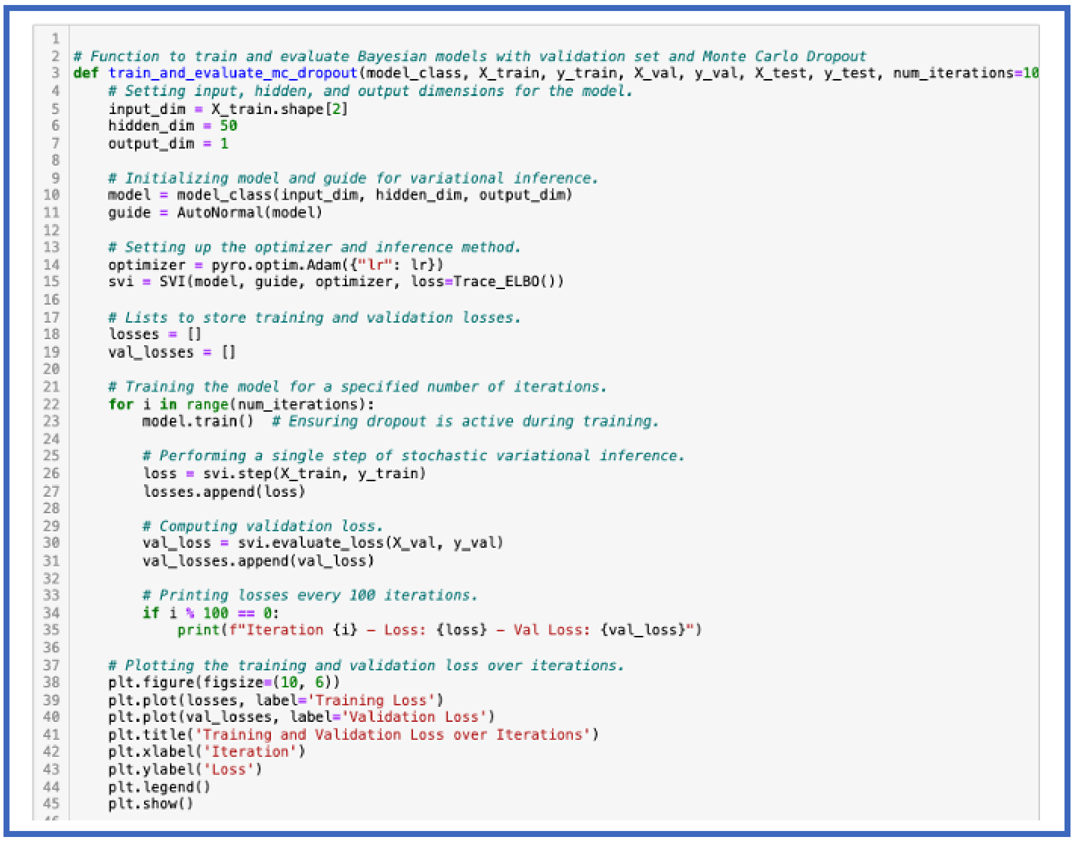
Figure 7: CNN Model Definition



```python
    def forward(self, x, y=None):
        # Passing input through the first convolutional layer.
        x = self.conv1(x)

        # Applying ReLU activation function after the first convolution.
        x = torch.relu(x)

        # Applying pooling after the first convolution.
        x = self.pool1(x)

        # Passing input through the second convolutional layer.
        x = self.conv2(x)

        # Applying ReLU activation function after the second convolution.
        x = torch.relu(x)

        # Applying pooling after the second convolution.
        x = self.pool2(x)

        # Flattening the tensor for fully connected layer.
        x = x.view(x.size(0), -1)

        # Applying dropout to the flattened tensor.
        x = self.dropout(x)

        # Passing through the fully connected layer and applying ReLU.
        x = torch.relu(self.fc(x))

        # Applying dropout again.
        x = self.dropout(x)

        # Computing mean prediction with the output layer.
        mean = self.fc_out(x).squeeze(-1)

        # Sampling sigma for probabilistic output.
        sigma = pyro.sample("sigma", dist.Uniform(0., 10.))

        # Defining the observation plate.
        with pyro.plate("data", x.shape[0]):
            obs = pyro.sample("obs", dist.Normal(mean, sigma), obs=y)
        return mean
```

Figure 8: CNN Forward Method

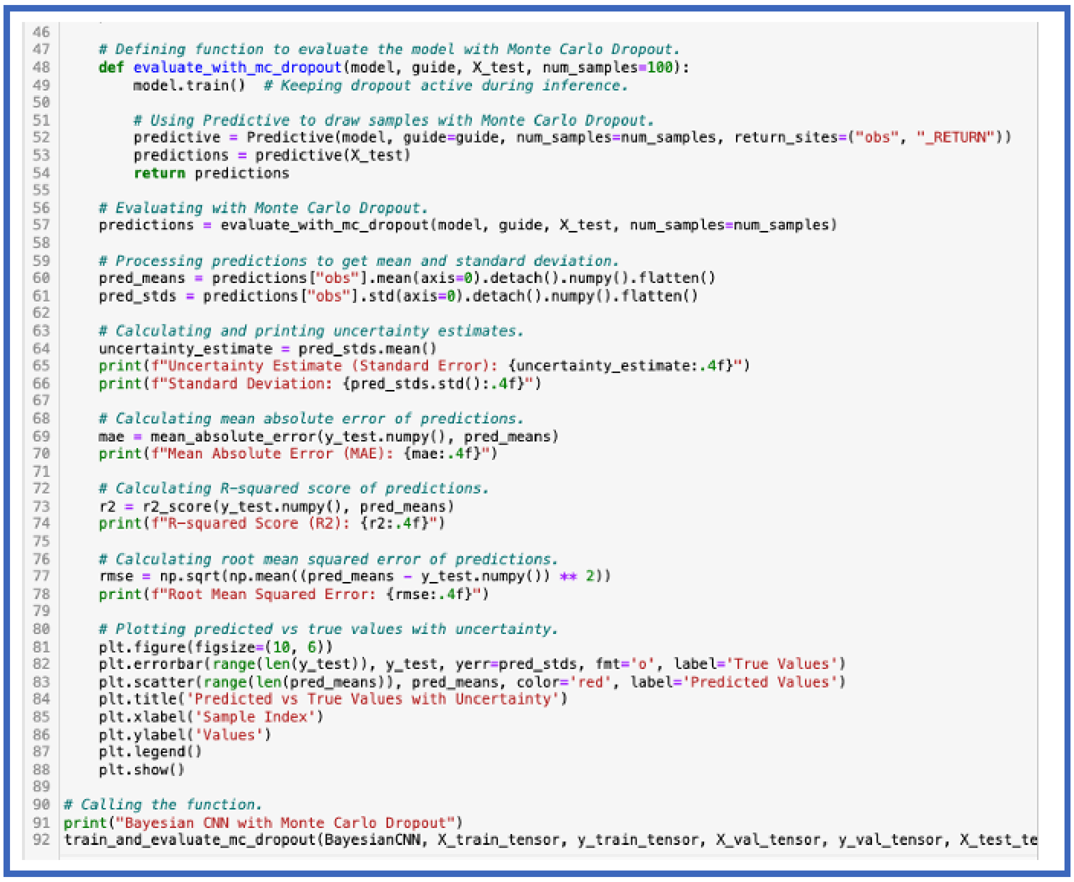
Training and validation (Figure 9) are performed by calculating input, hidden, and output dimensions for the model, initialising model and guide, setting up the optimiser and creating the empty list to store training and validation losses, executing variations and compute validation losses.



```python

# Function to train and evaluate Bayesian models with validation set and Monte Carlo Dropout
def train_and_evaluate_mc_dropout(model_class, X_train, y_train, X_val, y_val, X_test, y_test, num_iterations=10
    # Setting input, hidden, and output dimensions for the model.
    input_dim = X_train.shape[2]
    hidden_dim = 50
    output_dim = 1

    # Initializing model and guide for variational inference.
    model = model_class(input_dim, hidden_dim, output_dim)
    guide = AutoNormal(model)

    # Setting up the optimizer and inference method.
    optimizer = pyro.optim.Adam({"lr": lr})
    svi = SVI(model, guide, optimizer, loss=Trace_ELBO())

    # Lists to store training and validation losses.
    losses = []
    val_losses = []

    # Training the model for a specified number of iterations.
    for i in range(num_iterations):
        model.train()  # Ensuring dropout is active during training.

        # Performing a single step of stochastic variational inference.
        loss = svi.step(X_train, y_train)
        losses.append(loss)

        # Computing validation loss.
        val_loss = svi.evaluate_loss(X_val, y_val)
        val_losses.append(val_loss)

        # Printing losses every 100 iterations.
        if i % 100 == 0:
            print(f"Iteration {i} - Loss: {loss} - Val Loss: {val_loss}")

    # Plotting the training and validation loss over iterations.
    plt.figure(figsize=(10, 6))
    plt.plot(losses, label='Training Loss')
    plt.plot(val_losses, label='Validation Loss')
    plt.title('Training and Validation Loss over Iterations')
    plt.xlabel('Iteration')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()
```

Figure 9: Train and Validate CNN Model

Function to evaluate the model during inference is established. Predictive sampling is set up and predictions are generated, calculating means, standard deviation and performance metrics (Figure 10).



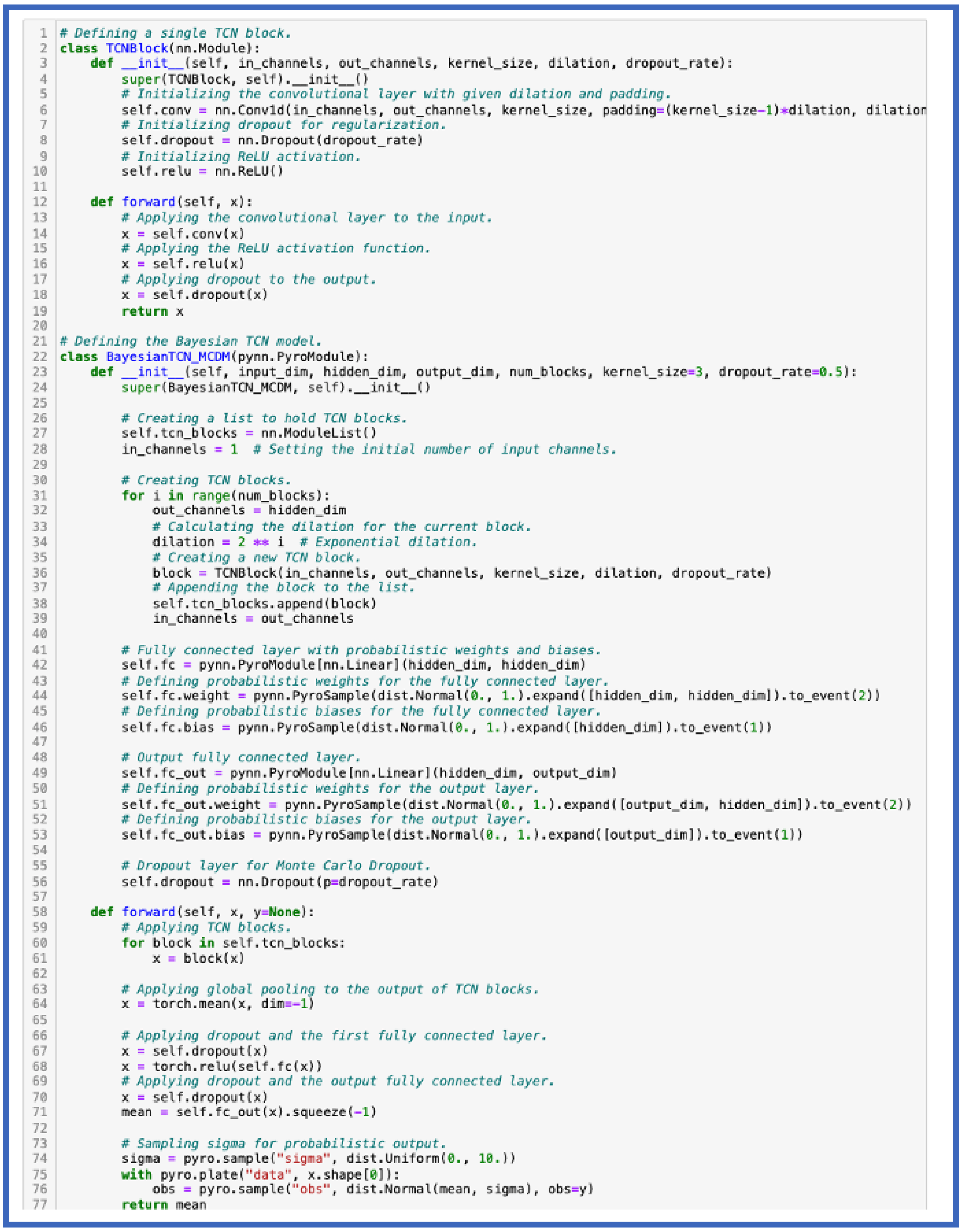
```python

    # Defining function to evaluate the model with Monte Carlo Dropout.
    def evaluate_with_mc_dropout(model, guide, X_test, num_samples=100):
        model.train()  # Keeping dropout active during inference.

        # Using Predictive to draw samples with Monte Carlo Dropout.
        predictive = Predictive(model, guide=guide, num_samples=num_samples, return_sites=("obs", "_RETURN"))
        predictions = predictive(X_test)
        return predictions

    # Evaluating with Monte Carlo Dropout.
    predictions = evaluate_with_mc_dropout(model, guide, X_test, num_samples=num_samples)

    # Processing predictions to get mean and standard deviation.
    pred_means = predictions["obs"].mean(axis=0).detach().numpy().flatten()
    pred_stds = predictions["obs"].std(axis=0).detach().numpy().flatten()

    # Calculating and printing uncertainty estimates.
    uncertainty_estimate = pred_stds.mean()
    print(f"Uncertainty Estimate (Standard Error): {uncertainty_estimate:.4f}")
    print(f"Standard Deviation: {pred_stds.std():.4f}")

    # Calculating mean absolute error of predictions.
    mae = mean_absolute_error(y_test.numpy(), pred_means)
    print(f"Mean Absolute Error (MAE): {mae:.4f}")

    # Calculating R-squared score of predictions.
    r2 = r2_score(y_test.numpy(), pred_means)
    print(f"R-squared Score (R2): {r2:.4f}")

    # Calculating root mean squared error of predictions.
    rmse = np.sqrt(np.mean((pred_means - y_test.numpy()) ** 2))
    print(f"Root Mean Squared Error: {rmse:.4f}")

    # Plotting predicted vs true values with uncertainty.
    plt.figure(figsize=(10, 6))
    plt.errorbar(range(len(y_test)), y_test, yerr=pred_stds, fmt='o', label='True Values')
    plt.scatter(range(len(pred_means)), pred_means, color='red', label='Predicted Values')
    plt.title('Predicted vs True Values with Uncertainty')
    plt.xlabel('Sample Index')
    plt.ylabel('Values')
    plt.legend()
    plt.show()

# Calling the function.
print("Bayesian CNN with Monte Carlo Dropout")
train_and_evaluate_mc_dropout(BayesianCNN, X_train_tensor, y_train_tensor, X_val_tensor, y_val_tensor, X_test_te
```

Figure 10: Evaluation of CNN Model

4.3 Bayesian TCN Model

Below code initialises TCN block with parameters for input, output channel, kernel size and dropout rate (Figure 11). The Convolutional layer is set up and dropout layer is initialised with ReLU function being defined and transforms the input to capture temporal dependencies. The data travels through the TCN block, the rate of dilation rises fast, allowing the network to gather information across multiple time scales. After TCN block, the output passes through global mean pooling combining temporal features. Then pooled tensor is sent through dropout layer and fully connected layer with ReLU activation. The model samples 'sigma' from a uniform distribution to produce forecast uncertainty. Finally, the forward method outputs the mean predictions.

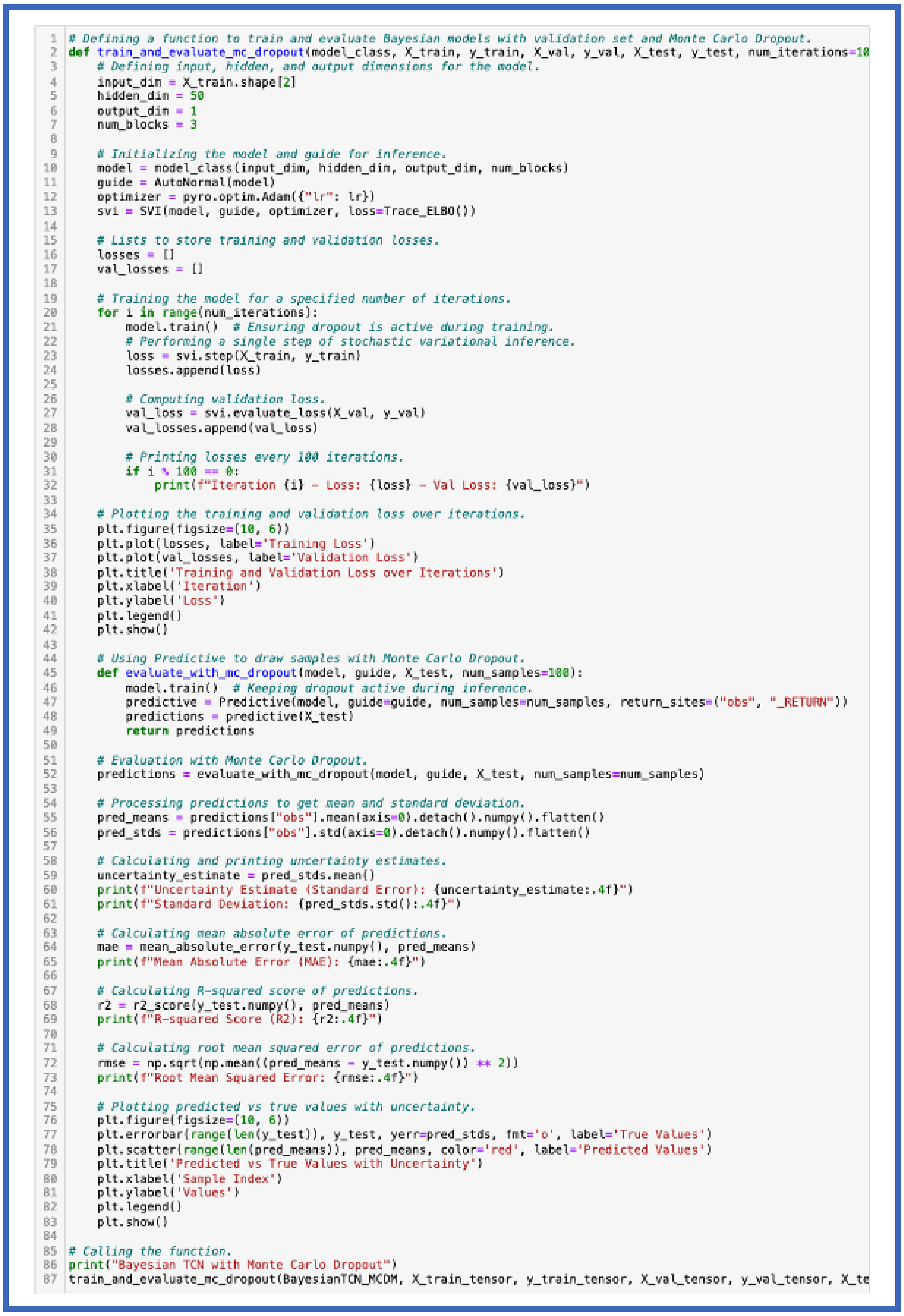
Code was researched at: (link).



```python
# Defining a single TCN block.
class TCNBlock(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, dilation, dropout_rate):
        super(TCNBlock, self).__init__()
        # Initializing the convolutional layer with given dilation and padding.
        self.conv = nn.Conv1d(in_channels, out_channels, kernel_size, padding=(kernel_size-1)*dilation, dilation
        # Initializing dropout for regularization.
        self.dropout = nn.Dropout(dropout_rate)
        # Initializing ReLU activation.
        self.relu = nn.ReLU()

    def forward(self, x):
        # Applying the convolutional layer to the input.
        x = self.conv(x)
        # Applying the ReLU activation function.
        x = self.relu(x)
        # Applying dropout to the output.
        x = self.dropout(x)
        return x

# Defining the Bayesian TCN model.
class BayesianTCN_MCDM(pynn.PyroModule):
    def __init__(self, input_dim, hidden_dim, output_dim, num_blocks, kernel_size=3, dropout_rate=0.5):
        super(BayesianTCN_MCDM, self).__init__()

        # Creating a list to hold TCN blocks.
        self.tcn_blocks = nn.ModuleList()
        in_channels = 1  # Setting the initial number of input channels.

        # Creating TCN blocks.
        for i in range(num_blocks):
            out_channels = hidden_dim
            # Calculating the dilation for the current block.
            dilation = 2 ** i  # Exponential dilation.
            # Creating a new TCN block.
            block = TCNBlock(in_channels, out_channels, kernel_size, dilation, dropout_rate)
            # Appending the block to the list.
            self.tcn_blocks.append(block)
            in_channels = out_channels

        # Fully connected layer with probabilistic weights and biases.
        self.fc = pynn.PyroModule[nn.Linear](hidden_dim, hidden_dim)
        # Defining probabilistic weights for the fully connected layer.
        self.fc.weight = pynn.PyroSample(dist.Normal(0., 1.).expand([hidden_dim, hidden_dim]).to_event(2))
        # Defining probabilistic biases for the fully connected layer.
        self.fc.bias = pynn.PyroSample(dist.Normal(0., 1.).expand([hidden_dim]).to_event(1))

        # Output fully connected layer.
        self.fc_out = pynn.PyroModule[nn.Linear](hidden_dim, output_dim)
        # Defining probabilistic weights for the output layer.
        self.fc_out.weight = pynn.PyroSample(dist.Normal(0., 1.).expand([output_dim, hidden_dim]).to_event(2))
        # Defining probabilistic biases for the output layer.
        self.fc_out.bias = pynn.PyroSample(dist.Normal(0., 1.).expand([output_dim]).to_event(1))

        # Dropout layer for Monte Carlo Dropout.
        self.dropout = nn.Dropout(p=dropout_rate)

    def forward(self, x, y=None):
        # Applying TCN blocks.
        for block in self.tcn_blocks:
            x = block(x)

        # Applying global pooling to the output of TCN blocks.
        x = torch.mean(x, dim=-1)

        # Applying dropout and the first fully connected layer.
        x = self.dropout(x)
        x = torch.relu(self.fc(x))
        # Applying dropout and the output fully connected layer.
        x = self.dropout(x)
        mean = self.fc_out(x).squeeze(-1)

        # Sampling sigma for probabilistic output.
        sigma = pyro.sample("sigma", dist.Uniform(0., 10.))
        with pyro.plate("data", x.shape[0]):
            obs = pyro.sample("obs", dist.Normal(mean, sigma), obs=y)
        return mean
```

Figure 11: TCN Model

Train and evaluate process (Figure 12) begins with initialisation of specified input, hidden, and output dimensions together with TCN blocks. Adam optimiser is configured, and variational guide is employed for Bayesian inference with a specific learning rate, During training model runs through set amount of iterations with drop out set to avoid overfitting. Using the training data, variational inference is performed in each iteration, updating models parameters as well as calculating the training loss. After training, losses are plotted across iterations. Also, evaluation metrics are calculated.



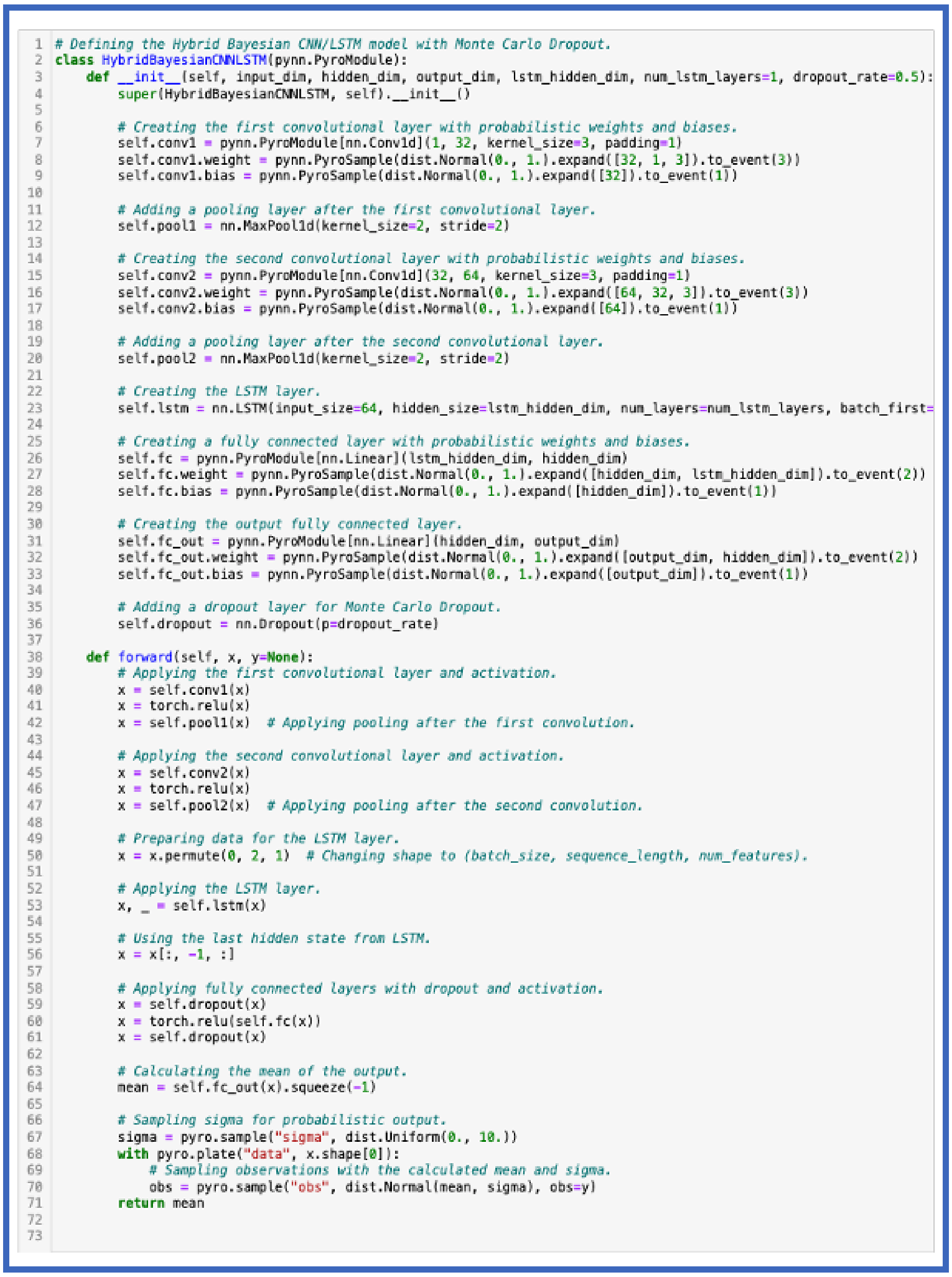
```python
# Defining a function to train and evaluate Bayesian models with validation set and Monte Carlo Dropout.
def train_and_evaluate_mc_dropout(model_class, X_train, y_train, X_val, y_val, X_test, y_test, num_iterations=10
    # Defining input, hidden, and output dimensions for the model.
    input_dim = X_train.shape[2]
    hidden_dim = 50
    output_dim = 1
    num_blocks = 3

    # Initializing the model and guide for inference.
    model = model_class(input_dim, hidden_dim, output_dim, num_blocks)
    guide = AutoNormal(model)
    optimizer = pyro.optim.Adam({"lr": lr})
    svi = SVI(model, guide, optimizer, loss=Trace_ELBO())

    # Lists to store training and validation losses.
    losses = []
    val_losses = []

    # Training the model for a specified number of iterations.
    for i in range(num_iterations):
        model.train()  # Ensuring dropout is active during training.
        # Performing a single step of stochastic variational inference.
        loss = svi.step(X_train, y_train)
        losses.append(loss)

        # Computing validation loss.
        val_loss = svi.evaluate_loss(X_val, y_val)
        val_losses.append(val_loss)

        # Printing losses every 100 iterations.
        if i % 100 == 0:
            print(f"Iteration {i} - Loss: {loss} - Val Loss: {val_loss}")

    # Plotting the training and validation loss over iterations.
    plt.figure(figsize=(10, 6))
    plt.plot(losses, label='Training Loss')
    plt.plot(val_losses, label='Validation Loss')
    plt.title('Training and Validation Loss over Iterations')
    plt.xlabel('Iteration')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()

    # Using Predictive to draw samples with Monte Carlo Dropout.
    def evaluate_with_mc_dropout(model, guide, X_test, num_samples=100):
        model.train()  # Keeping dropout active during inference.
        predictive = Predictive(model, guide=guide, num_samples=num_samples, return_sites=("obs", "_RETURN"))
        predictions = predictive(X_test)
        return predictions

    # Evaluation with Monte Carlo Dropout.
    predictions = evaluate_with_mc_dropout(model, guide, X_test, num_samples=num_samples)

    # Processing predictions to get mean and standard deviation.
    pred_means = predictions["obs"].mean(axis=0).detach().numpy().flatten()
    pred_stds = predictions["obs"].std(axis=0).detach().numpy().flatten()

    # Calculating and printing uncertainty estimates.
    uncertainty_estimate = pred_stds.mean()
    print(f"Uncertainty Estimate (Standard Error): {uncertainty_estimate:.4f}")
    print(f"Standard Deviation: {pred_stds.std():.4f}")

    # Calculating mean absolute error of predictions.
    mae = mean_absolute_error(y_test.numpy(), pred_means)
    print(f"Mean Absolute Error (MAE): {mae:.4f}")

    # Calculating R-squared score of predictions.
    r2 = r2_score(y_test.numpy(), pred_means)
    print(f"R-squared Score (R2): {r2:.4f}")

    # Calculating root mean squared error of predictions.
    rmse = np.sqrt(np.mean((pred_means - y_test.numpy()) ** 2))
    print(f"Root Mean Squared Error: {rmse:.4f}")

    # Plotting predicted vs true values with uncertainty.
    plt.figure(figsize=(10, 6))
    plt.errorbar(range(len(y_test)), y_test, yerr=pred_stds, fmt='o', label='True Values')
    plt.scatter(range(len(pred_means)), pred_means, color='red', label='Predicted Values')
    plt.title('Predicted vs True Values with Uncertainty')
    plt.xlabel('Sample Index')
    plt.ylabel('Values')
    plt.legend()
    plt.show()

# Calling the function.
print("Bayesian TCN with Monte Carlo Dropout")
train_and_evaluate_mc_dropout(BayesianTCN_MCDM, X_train_tensor, y_train_tensor, X_val_tensor, y_val_tensor, X_te
```

Figure 12: TCN Model Evaluation

4.4 Hybrid CNN/LSTM Model

The below code for a Hybrid Bayesian CNN and LSTM Model (Figure 13) shows data passing through a series of Convolutional Layers to capture spacial and temporal dependencies. The data undergoes a round of convolution, activation and pooling. This extracts higher level features from the input to pass it to LSTM Layer, transforming to the shape suitable for temporal dependencies where they are captured and into a hidden state which is passed to a series of fully connected layers. in the fully connected layer, data is exposed to dropout which sets some weights to zero randomly, to help with overfitting. Before the output layer, another dropout layer is applied to improve the model. Finally, the model reaches the output layer, where the mean of the predictions is calculated. The model calculates uncertainty in predictions using samples of standard deviation from a uniform distribution.

Code was researched at (link).

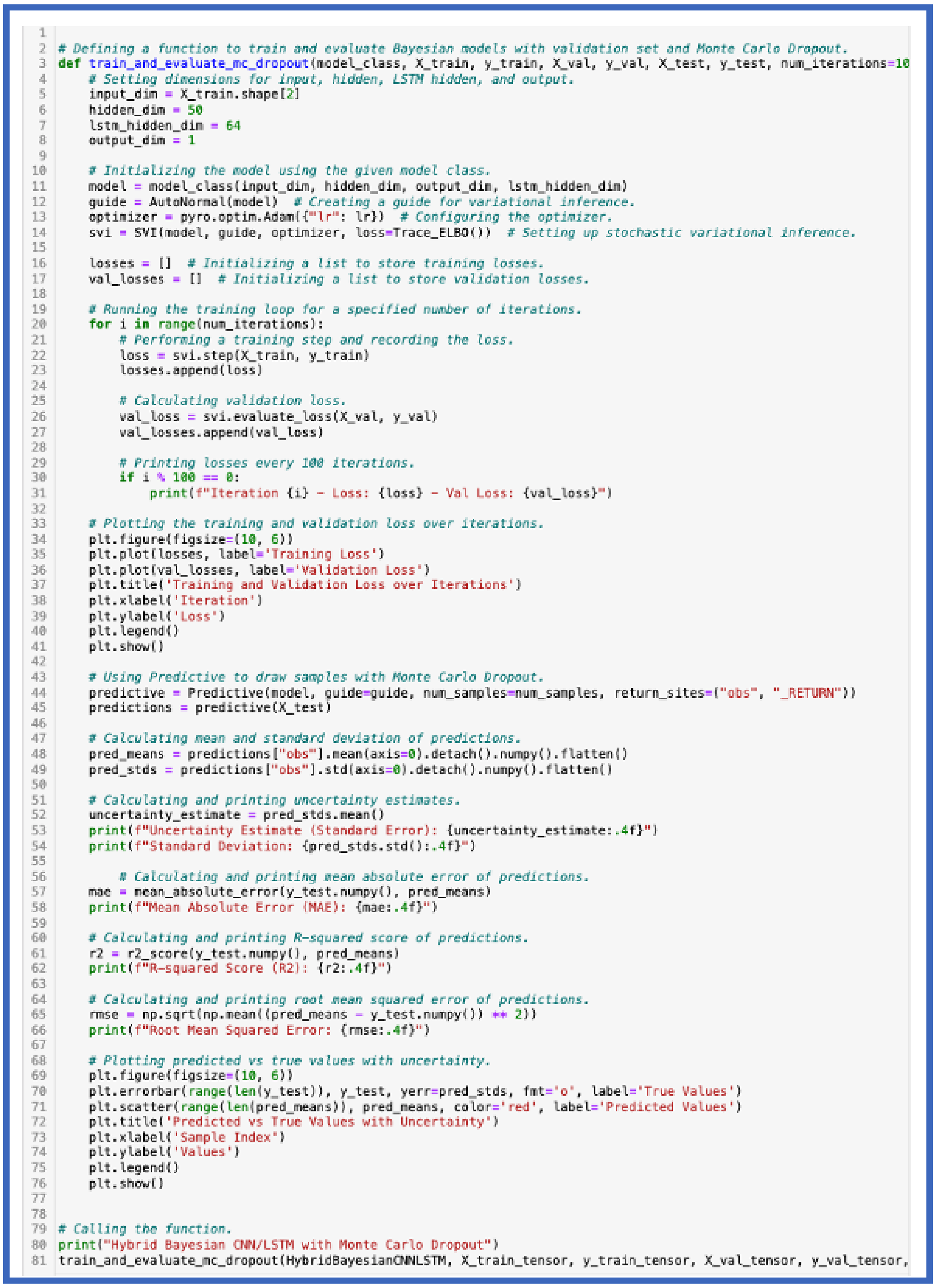


```python
# Defining the Hybrid Bayesian CNN/LSTM model with Monte Carlo Dropout.
class HybridBayesianCNNLSTM(pynn.PyroModule):
    def __init__(self, input_dim, hidden_dim, output_dim, lstm_hidden_dim, num_lstm_layers=1, dropout_rate=0.5):
        super(HybridBayesianCNNLSTM, self).__init__()

        # Creating the first convolutional layer with probabilistic weights and biases.
        self.conv1 = pynn.PyroModule[nn.Conv1d](1, 32, kernel_size=3, padding=1)
        self.conv1.weight = pynn.PyroSample(dist.Normal(0., 1.).expand([32, 1, 3]).to_event(3))
        self.conv1.bias = pynn.PyroSample(dist.Normal(0., 1.).expand([32]).to_event(1))

        # Adding a pooling layer after the first convolutional layer.
        self.pool1 = nn.MaxPool1d(kernel_size=2, stride=2)

        # Creating the second convolutional layer with probabilistic weights and biases.
        self.conv2 = pynn.PyroModule[nn.Conv1d](32, 64, kernel_size=3, padding=1)
        self.conv2.weight = pynn.PyroSample(dist.Normal(0., 1.).expand([64, 32, 3]).to_event(3))
        self.conv2.bias = pynn.PyroSample(dist.Normal(0., 1.).expand([64]).to_event(1))

        # Adding a pooling layer after the second convolutional layer.
        self.pool2 = nn.MaxPool1d(kernel_size=2, stride=2)

        # Creating the LSTM layer.
        self.lstm = nn.LSTM(input_size=64, hidden_size=lstm_hidden_dim, num_layers=num_lstm_layers, batch_first=

        # Creating a fully connected layer with probabilistic weights and biases.
        self.fc = pynn.PyroModule[nn.Linear](lstm_hidden_dim, hidden_dim)
        self.fc.weight = pynn.PyroSample(dist.Normal(0., 1.).expand([hidden_dim, lstm_hidden_dim]).to_event(2))
        self.fc.bias = pynn.PyroSample(dist.Normal(0., 1.).expand([hidden_dim]).to_event(1))

        # Creating the output fully connected layer.
        self.fc_out = pynn.PyroModule[nn.Linear](hidden_dim, output_dim)
        self.fc_out.weight = pynn.PyroSample(dist.Normal(0., 1.).expand([output_dim, hidden_dim]).to_event(2))
        self.fc_out.bias = pynn.PyroSample(dist.Normal(0., 1.).expand([output_dim]).to_event(1))

        # Adding a dropout layer for Monte Carlo Dropout.
        self.dropout = nn.Dropout(p=dropout_rate)

    def forward(self, x, y=None):
        # Applying the first convolutional layer and activation.
        x = self.conv1(x)
        x = torch.relu(x)
        x = self.pool1(x)  # Applying pooling after the first convolution.

        # Applying the second convolutional layer and activation.
        x = self.conv2(x)
        x = torch.relu(x)
        x = self.pool2(x)  # Applying pooling after the second convolution.

        # Preparing data for the LSTM layer.
        x = x.permute(0, 2, 1)  # Changing shape to (batch_size, sequence_length, num_features).

        # Applying the LSTM layer.
        x, _ = self.lstm(x)

        # Using the last hidden state from LSTM.
        x = x[:, -1, :]

        # Applying fully connected layers with dropout and activation.
        x = self.dropout(x)
        x = torch.relu(self.fc(x))
        x = self.dropout(x)

        # Calculating the mean of the output.
        mean = self.fc_out(x).squeeze(-1)

        # Sampling sigma for probabilistic output.
        sigma = pyro.sample("sigma", dist.Uniform(0., 10.))
        with pyro.plate("data", x.shape[0]):
            # Sampling observations with the calculated mean and sigma.
            obs = pyro.sample("obs", dist.Normal(mean, sigma), obs=y)
        return mean
```

Figure 13: Hybrid CNN/LSTM Model

The training and evaluation of Hybrid Bayesian CNN/LSTM Model (Figure 14) begins by setting dimensions for the model's input, hidden layer's, LSTM hidden units and dropout. To allow Bayesian learning, a variational inference guide is set up. The function configures an optimiser and Stochastic Variational Inference to model parameters. The training loop runs for 1000 iterations where the model is trained and losses are recorded, also validation loss is calculated and losses are printed to monitor the losses. The model is evaluated with performance metrics.

Code was researched at (link).



```python

# Defining a function to train and evaluate Bayesian models with validation set and Monte Carlo Dropout.
def train_and_evaluate_mc_dropout(model_class, X_train, y_train, X_val, y_val, X_test, y_test, num_iterations=10
    # Setting dimensions for input, hidden, LSTM hidden, and output.
    input_dim = X_train.shape[2]
    hidden_dim = 50
    lstm_hidden_dim = 64
    output_dim = 1

    # Initializing the model using the given model class.
    model = model_class(input_dim, hidden_dim, output_dim, lstm_hidden_dim)
    guide = AutoNormal(model)  # Creating a guide for variational inference.
    optimizer = pyro.optim.Adam({"lr": lr})  # Configuring the optimizer.
    svi = SVI(model, guide, optimizer, loss=Trace_ELBO())  # Setting up stochastic variational inference.

    losses = []  # Initializing a list to store training losses.
    val_losses = []  # Initializing a list to store validation losses.

    # Running the training loop for a specified number of iterations.
    for i in range(num_iterations):
        # Performing a training step and recording the loss.
        loss = svi.step(X_train, y_train)
        losses.append(loss)

        # Calculating validation loss.
        val_loss = svi.evaluate_loss(X_val, y_val)
        val_losses.append(val_loss)

        # Printing losses every 100 iterations.
        if i % 100 == 0:
            print(f"Iteration {i} - Loss: {loss} - Val Loss: {val_loss}")

    # Plotting the training and validation loss over iterations.
    plt.figure(figsize=(10, 6))
    plt.plot(losses, label='Training Loss')
    plt.plot(val_losses, label='Validation Loss')
    plt.title('Training and Validation Loss over Iterations')
    plt.xlabel('Iteration')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()

    # Using Predictive to draw samples with Monte Carlo Dropout.
    predictive = Predictive(model, guide=guide, num_samples=num_samples, return_sites=("obs", "_RETURN"))
    predictions = predictive(X_test)

    # Calculating mean and standard deviation of predictions.
    pred_means = predictions["obs"].mean(axis=0).detach().numpy().flatten()
    pred_stds = predictions["obs"].std(axis=0).detach().numpy().flatten()

    # Calculating and printing uncertainty estimates.
    uncertainty_estimate = pred_stds.mean()
    print(f"Uncertainty Estimate (Standard Error): {uncertainty_estimate:.4f}")
    print(f"Standard Deviation: {pred_stds.std():.4f}")

        # Calculating and printing mean absolute error of predictions.
    mae = mean_absolute_error(y_test.numpy(), pred_means)
    print(f"Mean Absolute Error (MAE): {mae:.4f}")

    # Calculating and printing R-squared score of predictions.
    r2 = r2_score(y_test.numpy(), pred_means)
    print(f"R-squared Score (R2): {r2:.4f}")

    # Calculating and printing root mean squared error of predictions.
    rmse = np.sqrt(np.mean((pred_means - y_test.numpy()) ** 2))
    print(f"Root Mean Squared Error: {rmse:.4f}")

    # Plotting predicted vs true values with uncertainty.
    plt.figure(figsize=(10, 6))
    plt.errorbar(range(len(y_test)), y_test, yerr=pred_stds, fmt='o', label='True Values')
    plt.scatter(range(len(pred_means)), pred_means, color='red', label='Predicted Values')
    plt.title('Predicted vs True Values with Uncertainty')
    plt.xlabel('Sample Index')
    plt.ylabel('Values')
    plt.legend()
    plt.show()


# Calling the function.
print("Hybrid Bayesian CNN/LSTM with Monte Carlo Dropout")
train_and_evaluate_mc_dropout(HybridBayesianCNNLSTM, X_train_tensor, y_train_tensor, X_val_tensor, y_val_tensor,
```

Figure 14: Hybrid Model Evaluation