Conformal Prediction in Limit Order Books: Calibration and Uncertainty Quantification of DeepLOB

Imperial College London

Fabio Rossi

CID: 2514536
Word count: 2783

Contents

List of Tables

Listings

Abstract

This paper explores the application of conformal prediction to enhance deep learning models for price prediction in limit order books (LOBs). We employ a conformal wrapper by customizing TorchCP package to calibrate DeepLOB, a popular deep learning architecture for mid-price forecasting. The paper explore different score functions such as Adaptive Prediction Sets (APS), Regularized APS (RAPS), and Soft APS (SAPS), and uses Temperature Scaling. The results demonstrate that calibrating DeepLOB can significantly improve its performance, interpretability and reliability, improving accuracy from 75% to 88% when acting only on prediction sets of size one. SAPS method achieves 92% prediction coverage with an average set size of 1.6 reducing Log Loss of the original model from 1.4 to 0.7. This study's results highlight the importance of machine learning calibration to improve models reliability and informativeness for better risk management and more informed algorithmic trading decisions.

1 Introduction

The limit order book (LOB) is an inventory of all outstanding limit orders for a specific financial instrument that is usually arranged by price levels and is updated in real-time as new orders are added, executed, or canceled (Gould et al. [2013]). Understanding and being able to forecast the dynamics of limit order books is crucial for market participants, such as traders, market makers, and regulators O'Hara [2005] and over the years different models have been developed to capture various aspects of limit order books, for example setting the optimal bid and ask based on market depth and order arrival Glosten and Milgrom [1985] and Kyle [1985].

Conventional econometric models, which assume linear correlations and normalcy of returns, are based on strong assumptions that are not necessarily represented in financial markets. Conversely, deep learning models have shown to be more successful in identifying intricate patterns and nonlinear relationships in high-dimensional LOB data, providing a more adaptable method that sacrifices simplicity and interpretability in favour of fewer assumptions Sirignano and Cont [2019]. The availability of LOB data and computing resources to train deep learning models has led to an upsurge in research output on this area, with DeepLOB being the most influential model (Zhang et al. [2019])..With the use of CNNs, DeepLOB is able to reduce the spatial component of the limit order book (depth) to a single data point, then the temporal component of these data points is then captured by LSTMs, and the output is then passed to a dense layer that makes predictions about the future price direction.

At the same time, there has been an increased attention to probabilistic machine learning, which tries to improve the interpretability and accuracy of model predictions. To provide an example, machine learning practitioners sometimes mistakenly believe that true probabilities are the same as the output of sigmoid and softmax functions, whereas in reality, these functions are just approximations of real probabilities and do not share the same properties of probability distributions. However, when the outputs are well-calibrated, these approximations can be handled with some degree of confidence almost as true probabilities. In contexts like the financial markets, where relying solely on data mining techniques like ROC curves and AUC to validate model performance can be extremely dangerous, as the majority of machine learning models are not well-calibrated, especially as complexity rises Guo et al. [2017]. Some of theses issues can be eased by conformal prediction, a framework for constructing prediction sets with guaranteed coverage under the assumption of exchangeability. The main idea is to assess how well a new example conforms to the previously observed data by minimizing a nonconformity measure, which quantifies the "strangeness" of an example relative to a set of other examples.

Conformal prediction is particularly valuable in financial applications, including LOB prediction, for several reasons:

- Distribution-free guarantees: Financial data often exhibit non-stationary and heavy-tailed distributions, conformal prediction's validity holds without assumptions about the underlying distribution.
- Uncertainty quantification: Conformal prediction provides well-calibrated prediction sets and prediction intervals, which are more informative than a point prediciton.
- Model-agnostic approach: Conformal prediction can be applied as a wrapper around any predictive model, without the need to re-train a model whose training can be expensive and time consuming.
- Interpretability: This is a key issue for regulated entities and conformal predictors can provide consistent logits thresholds, which combined with the prediction sets and calibrated probabilities can help to understand better model outputs.

2 Related Work

2.1 Deep Learning in Limit Order Books

Over the past 10 years, the rise in popularity, capacity, and availability of deep learning models has brought significant improvements in mid-price LOB forecasting. Although much success has been registered in the use of LSTM networks due to their capacity to capture long-term dependencies on time series data, newer architectures like Transformers are also showing promising results. Zhang et al. [2019] introduced DeepLOB, an effective hybrid model with convolutional and LSTM layers, which performs better in the mid-price prediction task: the first ones extract spatial characteristics from the LOB, while the second ones record temporal dependencies. In the wake of DeepLOB, dozens of studies introduced similar architectures to try and improve the model of Zhang et al. [2019], such as Tsantekidis et al. [2020] with DeepOB, which deepens the architecture and introduces residual connections; Huang et al. [2021] with AttnLOB, using attention mechanisms to focus only on the most relevant features in the LOB; and Ye et al. [2020] with LOB-Net, using a dual-stream network for processing both order book and order flow information. In this regard, each of these models addresses some particular issues in LOB modeling: the ability to capture multi-scale temporal dynamics, incorporation of a variety of data sources, improvement of feature extraction, and increasing the model's capacity to direct much of its attention toward the most relevant segments of the input data.

2.2 Transformers Architectures

Bilokon et al. [2023] presented a comparative analysis between Transformer architectures and LSTM-based models for the LOB prediction problem. The study shows that transformer-based models achieve very small improvements when it comes to absolute price sequence prediction. For the tasks of predicting price movement and predicting difference sequences, the LSTM-based models yield far more reliable and better performances. A new unique architecture for Transformer-based models, modified for financial prediction, and a new model, DLSTM, are introduced. Based on this result, LSTM-based architectures are more effective for most financial time series prediction applications. Lately, the most favored alternative, as far as time series modeling is concerned, is the Mamba models, first introduced by Gu et al. [2023]. The main innovation of Mamba is this selective state space layer, allowing for the modeling of long-range dependencies within time series. While relatively new in the field, some researchers believe that Mamba models can do better than LSTM networks, as shown by Zhang and Li [2023], although their application at the industrial level is quite minimal.

2.3 Conformal Prediction in Financial Markets

Conformal prediction is a relatively young field, having been invented by Vovk et al. [2005] in the 2000s, so the relevant literature about its applications in financial markets is scarce. The application of conformal prediction to the calibration of various machine learning models for forecasting energy prices was explored by Amjad and Zhou [2022], and regarding market making, Luetkebohmert et al. [2021] applied conformal prediction to forecast intervals of market makers' net positions. Limit order book analysis based on conformal prediction has been little researched and not well-founded, as most deep learning frameworks do not implement rigorous uncertainty quantification. The present paper fills this gap by investigating how conformal prediction may improve DeepLOB, which became a benchmark model in LOB prediction. Our objectives are to calibrate the model, provide statistically reliable estimates of uncertainty, and evaluate the impact on overall performance.

3 Methodology

3.1 Data

We use the FI-2010 dataset and the LSE dataset , both used in Zhang et al. [2019] as a train and test dataset respectively, for a fair comparison with the original model.

3.1.1 FI-2010 Dataset

The FI-2010 dataset was introduced by Ntakaris et al. [2018]. It includes information from five stocks on the Nasdaq Nordic stock market, spanning ten consecutive trading days and each sample has 40 attributes, which correspond to ten levels of the limit order book (LOB), including volume and bid/ask prices. We make use of the dataset's rolling 5-day standardised version.

3.1.2 London Stock Exchange (LSE) Dataset

One year of data (January 3, 2017 to December 24, 2017) and five equities are covered by the LSE dataset and to avoid bidding times, trading hours are limited to 08:30:00 - 16:00:00 and the dataset is structured similarly to FI-2010, with 40 attributes per timestamp.

3.1.3 Data Preprocessing

We utilise the normalised FI-2010 dataset and apply the same z-score normalisation to the LSE dataset in order to guarantee the model's efficacy under various market circumstances and across different products. Specifically, we feed our model with the 100 most recent LOB states for each of the two datasets, yielding an input shape of (100, 40) for every sample.

To construct our target feature we adopt the method described in Zhang et al. [2019]:
1. Calculate the mean of previous and future k mid-prices:

$$m^-(t) = \frac{1}{k}\sum_{i=0}^{k} p_{t-i}, \quad m^+(t) = \frac{1}{k}\sum_{i=1}^{k} p_{t+i}$$

2. Compute the percentage change:

$$l_t = \frac{m^+(t) - m^-(t)}{m^-(t)}$$

3. Assign labels based on a threshold α:
   - If $l_t > \alpha$: up (+1)
   - If $l_t < -\alpha$: down (-1)
   - Otherwise: stationary (0)

This method allows us to focus on price direction, simplifying our forecast to a classification task.

3.2 Model

We use the pre-trained model from the original publication Zhang et al. [2019] in this paper. In order to forecast mid-price fluctuations in limit order books, DeepLOB is a hybrid deep neural network that incorporates long short-term memory (LSTM) networks and convolutional neural networks (CNNs).

The model architecture consists of:
- Convolutional layers to extract spatial features from the LOB data

- LSTM layers to capture temporal dependencies
- Fully connected layers for final prediction

3.3 Calibration

Currently, most conformal prediction libraries only accept scikit-learn models, so we implement a conformal prediction wrapper for PyTorch, adjusting the TorchCP library Riquelme et al. [2022]. Since we need to calibrate our model on the output of a softmax function, we employ a class-wise predictor Shi et al. [2013] and explore several conformal score functions:

- Adaptive Prediction Sets (APS) Romano et al. [2020]: APS dynamically adjusts the size of prediction sets based on the difficulty of each instance, while mantaining coverage.
- Regularized Adaptive Prediction Sets (RAPS) Angelopoulos et al. [2022]: An extension of APS, it incorporates a regularization term to balance between coverage and set size in the prediction sets.
- Soft Adaptive Prediction Sets (SAPS) Cauchois et al. [2021]: SAPS further generalizes APS by employing a split method, which requires more data but often leads to more stable results.

4 Evaluation

To evaluate our conformal predictor applied to the DeepLOB model, we employ various metrics to assess both the probabilistic performance and the classification accuracy. We use probabilistic evaluation metrics for hyper-parameters tuning and to study the level calibration against the original model, as well as classification metrics to understand the effect of calibrating the base model on performance (Brier [1950], Good [1952], Murphy [1973], Powers [2011]). We test different metrics to optimize our hyper-parameters making sure the coverage is always met by penalizing values that violate it.

We evaluate two versions of our calibrated model, in one we sample at random from the prediction sets to choose our point prediction, in the other we only take into account sets with a single label. The logic behind this is that, in a real scenario we will not trade if all the labels are present in a prediction set and our strategy will be much more complex when we have two labels.

4.1 Probabilistic Evaluation Metrics

4.1.1 Test Data Coverage

Test data coverage measures the proportion of test samples for which the true label is included in the prediction set. It is defined as:

$$\text{Coverage} = \frac{1}{n}\sum_{i=1}^{n} \mathbb{1}\{y_i \in \Gamma^\epsilon(X_i)\} \tag{1}$$

where n is the number of test samples, y_i is the true label, and $\Gamma^\epsilon(X_i)$ is the prediction set for sample X_i at significance level ϵ. This metric helps us verify if the conformal predictor is well-calibrated, as the empirical coverage should be close to the desired confidence level $1-\epsilon$.

4.1.2 Average Set Size

Average set size quantifies the average number of labels in the prediction sets:

$$\text{Average Set Size} = \frac{1}{n}\sum_{i=1}^{n} |\Gamma^{\epsilon}(X_i)| \tag{2}$$

This metric helps us assess the efficiency of the conformal predictor. A smaller average set size indicates more informative predictions.

4.1.3 Size-1 Set Ratio

This metric measures the proportion of prediction sets containing only one label:

$$\text{Size-1 Set Ratio} = \frac{1}{n}\sum_{i=1}^{n} \mathbb{1}\{|\Gamma^{\epsilon}(X_i)| = 1\} \tag{3}$$

A higher percentage of sets with just a label indicates higher accuracy resulting in a simpler decision making process.

4.1.4 Brier Score

The Brier score measures the accuracy of probabilistic predictions:

$$\text{Brier Score} = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{k}(f_{ij} - o_{ij})^2 \tag{4}$$

where f_{ij} is the predicted probability of class j for sample i, and o_{ij} is 1 if the true class of sample i is j, and 0 otherwise. Lower Brier scores indicate better calibrated probabilities.

4.1.5 Log Loss

Log loss, also known as cross-entropy loss, measures the performance of a classification model where the prediction is a probability value between 0 and 1:

$$\text{Log Loss} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{k} y_{ij}\log(p_{ij}) \tag{5}$$

where y_{ij} is 1 if observation i belongs to class j and 0 otherwise, and p_{ij} is the predicted probability. Lower log loss values indicate better performance.

4.2 Classification Evaluation Metrics

4.2.1 Accuracy

Accuracy measures the overall correctness of the model:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{6}$$

4.2.2 Precision

Precision measures the model's ability to avoid labeling negative instances as positive:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{7}$$

4.2.3 Recall

Recall measures the model's ability to find all positive instances:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{8}$$

4.2.4 F1-Score

F1-Score is the harmonic mean of precision and recall, providing a balanced measure of the model's performance:

$$\text{F1-Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{9}$$

5 Results

5.1 Calibration

We found that across techniques, there were comparable Brier Scores and Log Losses and consistent coverage rates when optimising for either Brier Score (Table 1) or Log Loss (Table 2). On the other hand, the number of singletons dramatically dropped, particularly with greater coverage where, for example, the Size-1 Set Ratio varied between techniques at $\alpha = 0.10$, ranging from 0.087 to 0.097.

On the other hand, we obtained substantially higher proportions of singleton sets when we tuned to minimise the Size-1 Set Ratio (Table 3) and SAPS obtained a Size-1 Set Ratio of 0.631 at $\alpha = 0.10$, although at the expense of slightly greater Brier Scores and Log Losses.

As for score functions, SAPS was the best by far at maximising Size-1 Set Ratio but behaved similarly to the others when fine-tuned on Brier Score and Log Loss. As for our hyper-parameters, we almost always saw Temperatures less than 1, especially when maximizing Size-1 Set Ratio which decreased smaller probabilities and increase larger ones, effectively denoising the output of the original model.

Although single label are more practical, well-calibrated probabilities (as evaluated by Brier Score and Log Loss) are crucial for evaluating model uncertainty. This decision-making utility is prioritised by the Size-1 Set Ratio optimisation, but some probabilistic calibration quality may be lost in the process.

SAPS often provided a good balance in this trade-off such as at $\alpha = 0.20$ when optimizing for Brier Score, SAPS achieved a Size-1 Set Ratio of 0.607, higher than APS (0.473) and RAPS (0.461), while maintaining comparable Brier Scores and Log Losses.

Table 1: Comparison of APS, RAPS, and SAPS across different metrics and alpha values by fine-tuning hyper-parameters to minimize Brier Score Loss.

Alpha	Coverage Rate			Brier Score			Log Loss			Average Size			Size-1 Set Ratio		
	APS	RAPS	SAPS	APS	RAPS	SAPS	APS	RAPS	SAPS	APS	RAPS	SAPS	APS	RAPS	SAPS
0.10	0.925	0.924	0.924	0.129	0.128	0.128	0.703	0.702	0.703	2.116	2.106	2.096	0.092	0.088	0.097
0.15	0.890	0.888	0.890	0.128	0.128	0.128	0.703	0.701	0.703	1.816	1.808	1.803	0.258	0.261	0.270
0.20	0.852	0.853	0.850	0.129	0.129	0.129	0.706	0.703	0.703	1.536	1.541	1.456	0.473	0.461	0.607
0.25	0.815	0.816	0.818	0.129	0.129	0.128	0.704	0.703	0.700	1.269	1.274	1.354	0.731	0.726	0.646

Table 2: Comparison of APS, RAPS, and SAPS across different metrics and alpha values by fine-tuning hyper-parameters to minimize Log Loss.

Alpha	Coverage Rate			Brier Score			Log Loss			Average Size			Size-1 Set Ratio		
	APS	RAPS	SAPS	APS	RAPS	SAPS	APS	RAPS	SAPS	APS	RAPS	SAPS	APS	RAPS	SAPS
0.10	0.924	0.924	0.924	0.128	0.128	0.128	0.701	0.701	0.702	2.106	2.107	2.115	0.095	0.088	0.087
0.15	0.888	0.890	0.889	0.128	0.129	0.128	0.700	0.704	0.701	1.810	1.833	1.808	0.265	0.245	0.264
0.20	0.854	0.854	0.855	0.128	0.128	0.128	0.702	0.699	0.701	1.536	1.544	1.565	0.473	0.460	0.448
0.25	0.814	0.814	0.815	0.128	0.128	0.128	0.700	0.702	0.700	1.259	1.257	1.319	0.741	0.743	0.682

Table 3: Comparison of APS, RAPS, and SAPS across different metrics and alpha values by fine-tuning hyper-parameters to minimize Size-1 Set Ratio.

Alpha	Coverage Rate			Brier Score			Log Loss			Average Size			Size-1 Set Ratio		
	APS	RAPS	SAPS	APS	RAPS	SAPS	APS	RAPS	SAPS	APS	RAPS	SAPS	APS	RAPS	SAPS
0.10	0.923	0.924	0.916	0.160	0.157	0.143	2.065	1.411	0.770	1.987	2.031	1.693	0.222	0.161	0.631
0.15	0.886	0.886	0.882	0.160	0.159	0.166	2.033	1.950	0.863	1.628	1.677	1.517	0.506	0.389	0.725
0.20	0.849	0.851	0.844	0.160	0.155	0.128	2.041	1.231	0.695	1.342	1.377	1.360	0.770	0.694	0.789
0.25	0.812	0.813	0.810	0.160	0.158	0.210	2.064	1.548	1.045	1.189	1.188	1.211	0.876	0.868	0.865

5.2 Forecasting

For the next analysis we will focus on then model ($SAPS, \alpha = 0.1, Temperature = 0.9, \lambda = 0.0007$) that achieved the highest share of single sets while maintaining low Log Loss and Brier Loss. The comparison between the base DeepLOB model and the conformal model (using SAPS at $\alpha = 0.1$) further illustrates this trade-off (Tables and 5).

The conformal model showed improved calibration metrics with a lower Brier Score (0.1432 vs 0.1454) and a substantially lower Log Loss (0.7701 vs 1.4306). However, only 62.89% of its prediction sets contained a single label, compared to 100% for the base model (Table 4).

In classification performance, the unfiltered conformal model showed similar results to the base model which is expected since we just rely upon the highest logit to chose our label when we have a set size different from one. However, the filtered conformal model, which only makes predictions when the prediction set contains a single label, showed significantly improved performance across all metrics. This highlights the potential for using conformal prediction to abstain from making predictions when uncertainty is high, leading to more reliable forecasts but at the cost of reduced activity.

This analysis underscores the importance of carefully considering the specific requirements of the financial application when applying conformal prediction. If the primary goal is to have well-calibrated probability estimates and a clear representation of uncertainty, optimizing for Brier Score or Log Loss may be preferable. However, if the application requires single-label predictions for immediate decision-making, optimizing for the Size-1 Set Ratio or using a filtered approach may be more appropriate, albeit with some sacrifice in probabilistic calibration quality.

Table 4: Comparison of Calibration Metrics between Base Model and Conformal Model using SAPS at $\alpha = 0.1$.

Metric	**Base Model**	**Conformal Model (SAPS, $\alpha = 0.1$)**
Brier Score	0.1454	0.1432
Log Loss	1.4306	0.7701
Percentage of Prediction Sets of Size 1	100%	62.89%

Table 5: Comparison of Classification Metrics between Base Model, Conformal Model (Unfiltered), and Filtered Conformal Model.

Metric	**Base Model**	**Conformal Model (Unfiltered)**	**Filtered Conformal Model**
Model-Level Accuracy	0.7535	0.7531	0.8767
Label-Level Metrics			
Label 0			
Accuracy	0.8213	0.8212	0.9129
Precision	0.7341	0.7340	0.8695
Recall	0.7523	0.7521	0.8688
F1-Score	0.7431	0.7429	0.8691
Label 1			
Accuracy	0.8554	0.8551	0.9242
Precision	0.8074	0.8065	0.8956
Recall	0.7622	0.7622	0.8969
F1-Score	0.7841	0.7837	0.8962
Label 2			
Accuracy	0.8302	0.8301	0.9163
Precision	0.7204	0.7203	0.8617
Recall	0.7451	0.7451	0.8610
F1-Score	0.7325	0.7324	0.8613

6 Conclusion

This paper demonstrates the advantages of applying conformal prediction to calibrate deep learning models, exemplified by the application to the DeepLOB limit order book forecasting task. By employing methods like Temperature Scaling Adaptive Prediction Sets (APS), Relaxed Adaptive Prediction Sets (RAPS), and Split-Adaptive Prediction Sets (SAPS), we were able to enhance the reliability, interpretability and performance of the DeepLOB model.

When optimized for decision-making simplicity, so by maximising the number of prediction sets of size 1, the conformal model achieved a significant improvement in classification accuracy, increasing from 75.35% for the base DeepLOB model to 87.67% for the filtered conformal model (Table 5).

This study shows how conformal prediction can be used to support a range of tasks in finance, including risk management, market making, and algorithmic trading. More robust and informed decision-making can be facilitated by the prediction sets, which include prediction sets with coverage guarantees by construction. These sets can capture more information that a simple point prediction, such as price direction, volatility and uncertainty.

In conclusion, this work introduces a conformal DeepLOB model and demonstrates an effective application of conformal prediction with deep learning for financial markets and limit order book forecasting.

Acknowledgments

I would like to acknowledge the assistance of GPT-4 OpenAI [2024], a LLM developed by OpenAI, for helping refine the writing of this paper.

Appendix

For better replication please reach out to be granted access to the github repository.

```python
import warnings
import math
import torch

from torchcp.classification.predictors.base import BasePredictor
from torchcp.utils.common import calculate_conformal_value
import torch.nn.functional as F

import numpy as np


class SplitPredictor(BasePredictor):
    """
    Split Conformal Prediction (Vovk et a., 2005).
    Book: https://link.springer.com/book/10.1007/978-3-031-06649-8.

    :param score_function: non-conformity score function.
    :param model: a pytorch model.
    :param temperature: the temperature of Temperature Scaling.
    """
    def __init__(self, score_function, model=None, temperature=1):
        super().__init__(score_function, model, temperature)

    #############################
    # The calibration process
    ############################
    def calibrate(self, cal_dataloader, alpha):
        self._model.eval()
        logits_list = []
        labels_list = []
        with torch.no_grad():
            for examples in cal_dataloader:
                tmp_x, tmp_labels = examples[0].to(self._device), examples[1].to(self.
                    _device)
                tmp_logits = self._logits_transformation(self._model(tmp_x)).detach()
                logits_list.append(tmp_logits)
                labels_list.append(tmp_labels)
            logits = torch.cat(logits_list).float()
            labels = torch.cat(labels_list)
        self.calculate_threshold(logits, labels, alpha)

    def calculate_threshold(self, logits, labels, alpha):
        logits = logits.to(self._device)
        labels = labels.to(self._device)
        scores = self.score_function(logits, labels)
        self.q_hat = self._calculate_conformal_value(scores, alpha)

```

```
    def _calculate_conformal_value(self, scores, alpha):
        return calculate_conformal_value(scores, alpha)

    #############################
    # The prediction process
    ############################
    def predict(self, x_batch):
        """
        The input of score function is softmax probability.

        :param x_batch: a batch of instances.
        """
        self._model.eval()
        if self._model != None:
            x_batch = self._model(x_batch.to(self._device)).float()
        x_batch = self._logits_transformation(x_batch).detach()
        sets = self.predict_with_logits(x_batch)
        return sets

    def predict_with_logits(self, logits, q_hat=None):
        """
        The input of score function is softmax probability.
        if q_hat is not given by the function 'self.calibrate', the construction progress of prediction set is a naive method.

        :param logits: model output before softmax.
        :param q_hat: the conformal threshold.

        :return: prediction sets
        """


        scores = self.score_function(logits).to(self._device)
        if q_hat is None:
            q_hat = self.q_hat


        S = self._generate_prediction_set(scores, q_hat)

        return S

    def predict_probabilities(self, x_batch):
        """
        Custom method not included in the original library.
        Directly returns the softmax probabilities.

        :param x_batch: a batch of instances.
        :return: softmax probabilities
        """
        self._model.eval()
        if self._model is not None:
            x_batch = self._model(x_batch.to(self._device)).float()

        x_batch = self._logits_transformation(x_batch).detach()
        probabilities = F.softmax(x_batch, dim=1)

        return probabilities
```

```

    #############################
    # The evaluation process
    ############################

    def evaluate(self, val_dataloader):
        '''Modified method included in the original library.'''
        prediction_sets = []
        probs_sets  = []
        labels_list = []
        with torch.no_grad():
            for examples in val_dataloader:
                tmp_x, tmp_label = examples[0].to(self._device), examples[1].to(self._device)
                prediction_sets_batch = self.predict(tmp_x)
                prediction_sets.extend(prediction_sets_batch)
                probs_sets_batch = self.predict_probabilities(tmp_x)
                probs_sets.append(probs_sets_batch)
                labels_list.append(tmp_label)
        val_labels = torch.cat(labels_list)



        res_dict = {"Coverage_rate": self._metric('coverage_rate')(prediction_sets, val_labels),
                    "Average_size": self._metric('average_size')(prediction_sets, val_labels),
                    "Unilable_share":self._metric('unilabel_set_pct')(prediction_sets, val_labels),
                    "Multiclass_brier_score":self._metric('multiclass_brier_score_loss')(probs_sets, val_labels),
                    "Log_loss":self._metric('log_loss')(probs_sets, val_labels)}
        return res_dict
```

Listing 1: custom_split_predictor.py

```
import torch.nn as nn
import torch
from utils.constants import DEVICE

class deeplob(nn.Module):
    '''
    DeepLOB model as described in Zhang et. al. 2018
    '''
    def __init__(self, y_len):
        super().__init__()
        self.y_len = y_len

        # convolution blocks
        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=32, kernel_size=(1,2), stride=(1,2)),
            nn.LeakyReLU(negative_slope=0.01),
            nn.BatchNorm2d(32),
            nn.Conv2d(in_channels=32, out_channels=32, kernel_size=(4,1)),
            nn.LeakyReLU(negative_slope=0.01),
            nn.BatchNorm2d(32),
```

```
            nn.Conv2d(in_channels=32, out_channels=32, kernel_size=(4,1)),
            nn.LeakyReLU(negative_slope=0.01),
            nn.BatchNorm2d(32),
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(in_channels=32, out_channels=32, kernel_size=(1,2), stride=(1,2)),
            nn.Tanh(),
            nn.BatchNorm2d(32),
            nn.Conv2d(in_channels=32, out_channels=32, kernel_size=(4,1)),
            nn.Tanh(),
            nn.BatchNorm2d(32),
            nn.Conv2d(in_channels=32, out_channels=32, kernel_size=(4,1)),
            nn.Tanh(),
            nn.BatchNorm2d(32),
        )
        self.conv3 = nn.Sequential(
            nn.Conv2d(in_channels=32, out_channels=32, kernel_size=(1,10)),
            nn.LeakyReLU(negative_slope=0.01),
            nn.BatchNorm2d(32),
            nn.Conv2d(in_channels=32, out_channels=32, kernel_size=(4,1)),
            nn.LeakyReLU(negative_slope=0.01),
            nn.BatchNorm2d(32),
            nn.Conv2d(in_channels=32, out_channels=32, kernel_size=(4,1)),
            nn.LeakyReLU(negative_slope=0.01),
            nn.BatchNorm2d(32),
        )

        # inception moduels
        self.inp1 = nn.Sequential(
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=(1,1), padding='same'),
            nn.LeakyReLU(negative_slope=0.01),
            nn.BatchNorm2d(64),
            nn.Conv2d(in_channels=64, out_channels=64, kernel_size=(3,1), padding='same'),
            nn.LeakyReLU(negative_slope=0.01),
            nn.BatchNorm2d(64),
        )
        self.inp2 = nn.Sequential(
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=(1,1), padding='same'),
            nn.LeakyReLU(negative_slope=0.01),
            nn.BatchNorm2d(64),
            nn.Conv2d(in_channels=64, out_channels=64, kernel_size=(5,1), padding='same'),
            nn.LeakyReLU(negative_slope=0.01),
            nn.BatchNorm2d(64),
        )
        self.inp3 = nn.Sequential(
            nn.MaxPool2d((3, 1), stride=(1, 1), padding=(1, 0)),
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=(1,1), padding='same'),
            nn.LeakyReLU(negative_slope=0.01),
            nn.BatchNorm2d(64),
        )

```

```python
        # lstm layers
        self.lstm = nn.LSTM(input_size=192, hidden_size=64, num_layers=1, batch_first=
            True)
        self.fc1 = nn.Linear(64, self.y_len)

    def forward(self, x):
        # h0: (number of hidden layers, batch size, hidden size)
        h0 = torch.zeros(1, x.size(0), 64).to(DEVICE)
        c0 = torch.zeros(1, x.size(0), 64).to(DEVICE)

        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)

        x_inp1 = self.inp1(x)
        x_inp2 = self.inp2(x)
        x_inp3 = self.inp3(x)

        x = torch.cat((x_inp1, x_inp2, x_inp3), dim=1)

#         x = torch.transpose(x, 1, 2)
        x = x.permute(0, 2, 1, 3)
        x = torch.reshape(x, (-1, x.shape[1], x.shape[2]))

        x, _ = self.lstm(x, (h0, c0))
        x = x[:, -1, :]
        x = self.fc1(x)
        forecast_y = torch.softmax(x, dim=1)

        return forecast_y
```

Listing 2: DeepLOB.py

```python

from torch.utils import data
import torch
import numpy as np


class LobDataset(data.Dataset):
    """Characterizes a dataset for PyTorch"""
    def __init__(self, data, k, num_classes, T):
        """Initialization"""
        self.k = k
        self.num_classes = num_classes
        self.T = T

        x = self._prepare_x(data)
        y = self._get_label(data)
        x, y = self._data_classification(x, y, self.T)
        y = y[:,self.k] - 1
        self.length = len(x)

        x = torch.from_numpy(x)
        self.x = torch.unsqueeze(x, 1).float()
        self.y = torch.from_numpy(y).short()

```

```
    def __len__(self):
        """Denotes the total number of samples"""
        return self.length

    def __getitem__(self, index):
        """Generates samples of data"""
        return self.x[index], self.y[index]

    @staticmethod
    def _prepare_x(data):
        df1 = data[:40, :].T
        return np.array(df1)

    @staticmethod
    def _get_label(data):
        lob = data[-5:, :].T
        return lob

    @staticmethod
    def _data_classification(X, Y, T):
        [N, D] = X.shape
        df = np.array(X)

        dY = np.array(Y)

        dataY = dY[T - 1:N]

        dataX = np.zeros((N - T + 1, T, D))
        for i in range(T, N + 1):
            dataX[i - T] = df[i - T:i, :]

        return dataX, dataY
```

Listing 3: torch_dfs.py

```
from typing import Any

import numpy as np

from torchcp.utils.registry import Registry
from sklearn.metrics import brier_score_loss as sklearn_brier_score_loss
from sklearn.metrics import log_loss as  sklearn_log_loss
METRICS_REGISTRY_CLASSIFICATION = Registry("METRICS")


##########################################
# Marginal coverage metric
##########################################

@METRICS_REGISTRY_CLASSIFICATION.register()
def coverage_rate(prediction_sets, labels):
    labels = labels.cpu()
    cvg = 0
    for index, ele in enumerate(zip(prediction_sets, labels)):
        if ele[1] in ele[0]:
            cvg += 1
    return cvg / len(prediction_sets)
```

```


@METRICS_REGISTRY_CLASSIFICATION.register()
def average_size(prediction_sets, labels):
    labels = labels.cpu()
    avg_size = 0
    for index, ele in enumerate(prediction_sets):
        avg_size += len(ele)
    return avg_size / len(prediction_sets)



#########################################
# Conditional coverage metric
#########################################

@METRICS_REGISTRY_CLASSIFICATION.register()
def CovGap(prediction_sets, labels, alpha, num_classes):
    labels = labels.cpu()
    rate_classes = []
    for k in range(num_classes):
        idx = np.where(labels == k)[0]
        selected_preds = [prediction_sets[i] for i in idx]
        if len(labels[labels == k]) != 0:
            rate_classes.append(coverage_rate(selected_preds, labels[labels == k]))
    rate_classes = np.array(rate_classes)
    return np.mean(np.abs(rate_classes - (1 - alpha))) * 100


@METRICS_REGISTRY_CLASSIFICATION.register()
def VioClasses(prediction_sets, labels, alpha, num_classes):
    labels = labels.cpu()
    violation_nums = 0
    for k in range(num_classes):
        if len(labels[labels == k]) == 0:
            violation_nums += 1
        else:
            idx = np.where(labels == k)[0]
            selected_preds = [prediction_sets[i] for i in idx]
            if coverage_rate(selected_preds, labels[labels == k]) < 1 - alpha:
                violation_nums += 1
    return violation_nums


@METRICS_REGISTRY_CLASSIFICATION.register()
def DiffViolation(logits, prediction_sets, labels, alpha, num_classes):
    labels = labels.cpu()
    strata_diff = [[1, 1], [2, 3], [4, 6], [7, 10], [11, 100], [101, 1000]]
    correct_array = np.zeros(len(labels))
    size_array = np.zeros(len(labels))
    topk = []
    for index, ele in enumerate(logits):
        I = ele.argsort(descending=True)
        target = labels[index]
        topk.append(np.where((I - target.view(-1, 1).numpy()) == 0)[1] + 1)
        correct_array[index] = 1 if labels[index] in prediction_sets[index] else 0
        size_array[index] = len(prediction_sets[index])
```

```
    topk = np.concatenate(topk)

    ccss_diff = {}
    diff_violation = -1

    for stratum in strata_diff:

        temp_index = np.argwhere((topk >= stratum[0]) & (topk <= stratum[1]))
        ccss_diff[str(stratum)] = {}
        ccss_diff[str(stratum)]['cnt'] = len(temp_index)
        if len(temp_index) == 0:
            ccss_diff[str(stratum)]['cvg'] = 0
            ccss_diff[str(stratum)]['sz'] = 0
        else:
            temp_index = temp_index[:, 0]
            cvg = np.round(np.mean(correct_array[temp_index]), 3)
            sz = np.round(np.mean(size_array[temp_index]), 3)

            ccss_diff[str(stratum)]['cvg'] = cvg
            ccss_diff[str(stratum)]['sz'] = sz
            stratum_violation = max(0, (1 - alpha) - cvg)
            diff_violation = max(diff_violation, stratum_violation)

    diff_violation_one = 0
    for i in range(1, num_classes + 1):
        temp_index = np.argwhere(topk == i)
        if len(temp_index) > 0:
            temp_index = temp_index[:, 0]
            stratum_violation = max(0, (1 - alpha) - np.mean(correct_array[temp_index]))
            diff_violation_one = max(diff_violation_one, stratum_violation)
    return diff_violation, diff_violation_one, ccss_diff


@METRICS_REGISTRY_CLASSIFICATION.register()
def SSCV(prediction_sets, labels, alpha, stratified_size=[[0, 1], [2, 3], [4, 10], [11, 100], [101, 1000]]):
    """
    Size-stratified coverage violation (SSCV)
    """
    labels = labels.cpu()
    size_array = np.zeros(len(labels))
    correct_array = np.zeros(len(labels))
    for index, ele in enumerate(prediction_sets):
        size_array[index] = len(ele)
        correct_array[index] = 1 if labels[index] in ele else 0

    sscv = -1
    for stratum in stratified_size:
        temp_index = np.argwhere((size_array >= stratum[0]) & (size_array <= stratum[1]))
        if len(temp_index) > 0:
            stratum_violation = abs((1 - alpha) - np.mean(correct_array[temp_index]))
            sscv = max(sscv, stratum_violation)
    return sscv


```

```python
class Metrics:

    def __call__(self, metric) -> Any:
        if metric not in METRICS_REGISTRY_CLASSIFICATION.registered_names():
            raise NameError(f"The metric: {metric} is not defined in TorchCP.")
        return METRICS_REGISTRY_CLASSIFICATION.get(metric)


##########################################
# Custom metrics
##########################################


@METRICS_REGISTRY_CLASSIFICATION.register()
def multiclass_brier_score_loss(prediction_probs, labels):
    # Convert list of tensors to NumPy arrays, handling zero-dimensional tensors
    prediction_probs_np = np.concatenate(
        [tensor.cpu().numpy().reshape(1, -1) if tensor.ndim == 0 else tensor.cpu().numpy() for tensor in prediction_probs],
        axis=0
    )
    labels_np = np.concatenate(
        [tensor.cpu().numpy().reshape(1) if tensor.ndim == 0 else tensor.cpu().numpy() for tensor in labels],
        axis=0
    )

    # Calculate Brier score for each class and average them
    brier_conformal = np.mean([
        sklearn_brier_score_loss(labels_np == i, prediction_probs_np[:, i])
        for i in range(prediction_probs_np.shape[1])
    ])

    return brier_conformal



@METRICS_REGISTRY_CLASSIFICATION.register()
def log_loss(prediction_probs, labels):
    # Convert list of tensors to NumPy arrays, handling zero-dimensional tensors
    prediction_probs_np = np.concatenate(
        [tensor.cpu().numpy().reshape(1, -1) if tensor.ndim == 0 else tensor.cpu().numpy() for tensor in prediction_probs],
        axis=0
    )
    labels_np = np.concatenate(
        [tensor.cpu().numpy().reshape(1) if tensor.ndim == 0 else tensor.cpu().numpy() for tensor in labels],
        axis=0
    )

    # Calculate log loss
    return sklearn_log_loss(labels_np, prediction_probs_np)
```

```python
@METRICS_REGISTRY_CLASSIFICATION.register()
def unilabel_set_pct(prediction_sets, labels):
    labels = labels.cpu()
    nb_sets = 0
    for index, ele in enumerate(prediction_sets):
        if len(ele) == 1:
            nb_sets += 1
    return nb_sets / len(prediction_sets)
```

Listing 4: metrics.py

```python
import json
import pandas as pd
import numpy as np
from typing import Tuple

class ResultsDataHandler:
    def __init__(self, path: str) -> None:
        with open(path) as f:
            self.raw_data = json.load(f)
        self.df, self.hyperparam_df = self._process_data()

    def _process_data(self) -> Tuple[pd.DataFrame, pd.DataFrame]:
        data = self.raw_data
        rows = []

        for fun, alphas in data.items():
            for alpha, results in alphas.items():
                penalty = results.get('best_lambda', np.nan)  # Default to NaN if '
                    best_lambda' is not available

                row = {
                    'Function': fun,
                    'Alpha': alpha,
                    'Best_Temperature': results['best_temperature'],
                    'Coverage_Rate': results['test_results']['Coverage_rate'],
                    'Average_Size': results['test_results']['Average_size'],
                    'Unilable_share': results['test_results']['Unilable_share'],
                    'Brier_score': results['test_results']['Multiclass_brier_score'],
                    'Log_loss': results['test_results']['Log_loss']

                }

                if fun in ['RAPS', 'SAPS']:
                    row['Lambda'] = results.get('best_lambda', np.nan)

                rows.append(row)

        processed_data = pd.DataFrame(rows)

        metrics = ['Coverage_Rate','Brier_score', 'Log_loss', 'Average_Size', '
            Unilable_share']
        hyperparams = ['Function', 'Alpha', 'Best_Temperature', 'Lambda']

        # Extract hyperparameters
        hyperparam_df = processed_data[hyperparams]

```

```
        # Pivot the DataFrame to get a suitable format
        df = processed_data.pivot_table(index='Alpha', columns='Function', values=metrics).reindex(columns=metrics, level=0)#.round(3)

        return df, hyperparam_df

    def get_styled_dataframe(self):
        """
        Get the styled DataFrame with highlights for average size and coverage rate.

        Returns:
            Tuple[pd.io.formats.style.Styler, pd.io.formats.style.Styler]: The styled DataFrames.
        """
        def highlight_min_avg_size(s):
            is_min = s == s.min()
            return ['background-color: coral' if v else '' for v in is_min]

        def highlight_max_coverage(s):
            is_max = s == s.max()
            return ['background-color: lightgreen' if v else '' for v in is_max]

        # Extract the 'Coverage_Rate' and 'Average_Size' from the pivoted DataFrame
        avg_size_df = pd.DataFrame(self.df['Average_Size'])
        coverage_df = pd.DataFrame(self.df['Coverage_Rate'])

        # Apply styling separately
        styled_avg_size_df = avg_size_df.style.apply(highlight_min_avg_size, axis=1)
        styled_coverage_df = pd.DataFrame(coverage_df).style.apply(highlight_max_coverage, axis=1)

        return styled_avg_size_df, styled_coverage_df

    def save_hyperparam_df(self, file_path: str) -> None:
        """
        Save the hyperparameter DataFrame to a CSV file.

        Args:
            file_path (str): The path to the CSV file.
        """
        self.hyperparam_df.to_csv(file_path, index=False)
        print(f"Hyperparameter DataFrame saved to {file_path}")

if __name__ =='__main__':
    res = ResultsDataHandler('results_with_temperature.json')

    # Get styled DataFrames
    styled_avg_size_df, styled_coverage_df = res.get_styled_dataframe()

    # Display the styled DataFrames
    print(styled_avg_size_df)
    print(styled_coverage_df)

    # Save hyperparameter DataFrame to CSV
    res.save_hyperparam_df('hyperparams.csv')
```

Listing 5: result_datahandler.py

```
# %%
# load packages
import numpy as np
import torch
import optuna
import json
#from sklearn.metrics import brier_score_loss, log_loss, accuracy_score,
    precision_score, recall_score, f1_score
from torchcp.classification.scores import THR, APS, SAPS, RAPS
from torchcp.classification.predictors import ClassWisePredictor
import pandas as pd

from typing import Callable, Optional
from model.DeepLOB import deeplob
from utils.torch_dfs import LobDataset
from utils.constants import DEVICE

# %%
batch_size = 64


dec_data = np.loadtxt('data/input/Train_Dst_NoAuction_DecPre_CF_7.txt')

dec_cal = dec_data[:, int(np.floor(dec_data.shape[1] * 0.8)):int(np.floor(dec_data.
    shape[1] * 0.975))]
dec_val = dec_data[:, int(np.floor(dec_data.shape[1] * 0.975)):]

dataset_cal = LobDataset(data=dec_cal, k=4, num_classes=3, T=100)
cal_loader = torch.utils.data.DataLoader(dataset=dataset_cal, batch_size=batch_size,
    shuffle=False)

print('Calibration Data Shape:', dataset_cal.x.shape, dataset_cal.y.shape)

dataset_val = LobDataset(data=dec_val, k=4, num_classes=3, T=100)
val_loader = torch.utils.data.DataLoader(dataset=dataset_val, batch_size=batch_size,
    shuffle=False)

print('Validation Data Shape:', dataset_val.x.shape, dataset_val.y.shape)

del dec_cal, dec_data, dataset_cal, dec_val, dataset_val

dec_test1 = np.loadtxt('data/input/Test_Dst_NoAuction_DecPre_CF_7.txt')
dec_test2 = np.loadtxt('data/input/Test_Dst_NoAuction_DecPre_CF_8.txt')
dec_test3 = np.loadtxt('data/input/Test_Dst_NoAuction_DecPre_CF_9.txt')
dec_test = np.hstack((dec_test1, dec_test2, dec_test3))

dataset_test = LobDataset(data=dec_test, k=4, num_classes=3, T=100)
test_loader = torch.utils.data.DataLoader(dataset=dataset_test, batch_size=batch_size,
     shuffle=False)


print('Test Data Shape:',dataset_test.x.shape, dataset_test.y.shape)

del dec_test, dec_test1, dec_test2, dec_test3, dataset_test

# %%
```

```python
model = deeplob(y_len = 3)
model_path = 'model/best_val_model_pytorch'

model = torch.load(model_path,  map_location=torch.device(DEVICE))
model.eval()

# %%
alphas = [0.1, 0.15, 0.2, 0.25]
score_fun = [APS, RAPS, SAPS]
res = {}

optuna.logging.set_verbosity(optuna.logging.WARNING)

def evaluate_predictor(fun: Callable, alpha: float, temperature: float, x: Optional[
    float] = None, loader=None):
    if loader is None:
        loader = val_loader  # Default to validation loader if none provided

    if x is not None:
        predictor = ClassWisePredictor(score_function=fun(x), model=model, temperature
            =temperature)
    else:
        predictor = ClassWisePredictor(score_function=fun(), model=model, temperature=
            temperature)

    predictor.calibrate(cal_loader, alpha)
    return predictor.evaluate(loader)

def objective(trial, fun: Callable, alpha: float):
    temperature = trial.suggest_float("temperature", 0.1, 10.0, log=True)

    if fun not in [APS]:
        x = trial.suggest_float("lambda", 0, 1)
        evaluation_results = evaluate_predictor(fun, alpha, temperature, x)
    else:
        evaluation_results = evaluate_predictor(fun, alpha, temperature)

    coverage_rate = evaluation_results['Coverage_rate']
    average_size = evaluation_results['Average_size']
    unilable_share = evaluation_results['Unilable_share']

    brier_score = evaluation_results['Multiclass_brier_score']
    log_loss = evaluation_results['Log_loss']

    if coverage_rate >= 1 - alpha:
        return average_size  # Direction is minimize so adjust sign accordingly
    else:
        return float('inf')  # Penalize trials that don't meet the coverage rate
            requirement

def process_score_function(fun: Callable):
    fun_name = fun.__name__
    print(fun_name)
    res[fun_name] = {}

    for alpha in alphas:
        print(f'Processing alpha: {alpha}')
```

```
        study = optuna.create_study(direction="minimize")
        study.optimize(lambda trial: objective(trial, fun, alpha), n_trials=50,
            show_progress_bar=True)

        best_temperature = study.best_params['temperature']
        if fun not in [APS]:
            best_lambda = study.best_params['lambda']
            # After finding the best hyperparameters, evaluate on the test set
            evaluation_results = evaluate_predictor(fun, alpha, best_temperature,
                best_lambda, loader=test_loader)
            res[fun_name][str(alpha)] = {
                "best_lambda": best_lambda,
                "best_temperature": best_temperature,
                "test_results": evaluation_results
            }
            print(f'alpha: {alpha}, best lambda: {best_lambda}, best temperature: {
                best_temperature}, test results: {evaluation_results}')
        else:
            # For APS, only tune temperature
            evaluation_results = evaluate_predictor(fun, alpha, best_temperature,
                loader=test_loader)
            res[fun_name][str(alpha)] = {
                "best_temperature": best_temperature,
                "test_results": evaluation_results
            }
            print(f'alpha: {alpha}, best temperature: {best_temperature}, test results
                : {evaluation_results}')

for fun in score_fun:
    process_score_function(fun)

with open('results_minsetsize.json', 'w') as json_file:
    json.dump(res, json_file, indent=4)

print("Results saved to results_minsetsize.json")
```

Listing 6: calibration.ipynb

```
# %%
# load packages
import numpy as np
import torch
from sklearn.metrics import brier_score_loss, log_loss, accuracy_score,
    precision_score, recall_score, f1_score
from torchcp.classification.scores import  SAPS
from torchcp.classification.predictors import ClassWisePredictor
import pandas as pd

from model.DeepLOB import deeplob
from utils.torch_dfs import LobDataset
from utils.constants import DEVICE
from helpers.result_datahandler import ResultsDataHandler

# %%
batch_size = 64


```

```
dec_data = np.loadtxt('data/input/Train_Dst_NoAuction_DecPre_CF_7.txt')

dec_cal = dec_data[:, int(np.floor(dec_data.shape[1] * 0.8)):int(np.floor(dec_data.
    shape[1] * 0.975))]
dataset_cal = LobDataset(data=dec_cal, k=4, num_classes=3, T=100)
cal_loader = torch.utils.data.DataLoader(dataset=dataset_cal, batch_size=batch_size,
    shuffle=False)

print('Calibration Data Shape:', dataset_cal.x.shape, dataset_cal.y.shape)

del dec_cal, dec_data, dataset_cal

dec_test1 = np.loadtxt('data/input/Test_Dst_NoAuction_DecPre_CF_7.txt')
dec_test2 = np.loadtxt('data/input/Test_Dst_NoAuction_DecPre_CF_8.txt')
dec_test3 = np.loadtxt('data/input/Test_Dst_NoAuction_DecPre_CF_9.txt')
dec_test = np.hstack((dec_test1, dec_test2, dec_test3))

dataset_test = LobDataset(data=dec_test, k=4, num_classes=3, T=100)
test_loader = torch.utils.data.DataLoader(dataset=dataset_test, batch_size=batch_size,
     shuffle=False)


print('Test Data Shape:',dataset_test.x.shape, dataset_test.y.shape)

del dec_test, dec_test1, dec_test2, dec_test3, dataset_test

# %%
model = deeplob(y_len = 3)
model_path = 'model/best_val_model_pytorch'

model = torch.load(model_path,  map_location=torch.device(DEVICE))

# %%
res_minbrier = ResultsDataHandler('data/results/results_minbrier.json')
res_minbrier.save_hyperparam_df('data/hyperparameters/optimal_brier.csv')
res_minbrier.df.round(3)


# %%
res_logloss = ResultsDataHandler('data/results/results_minlogloss.json')
res_logloss.save_hyperparam_df('data/hyperparameters/optimal_logloss.csv')
res_logloss.df.round(3)

# %%
res_maxuni = ResultsDataHandler('data/results/results_maxunilabel.json')
res_maxuni.save_hyperparam_df('data/hyperparameters/optimal_unilabel.csv')
res_maxuni.df.round(3)

# %%
res_minset = ResultsDataHandler('data/results/results_minsetsize.json')
res_minset.save_hyperparam_df('data/hyperparameters/optimal_minsetsize.csv')
res_minset.df.round(3)

# %%
alpha = 0.1
fun = 'SAPS'
params = res_maxuni.hyperparam_df
```

```python

best_model_params = params[(params['Alpha'] == str(alpha))&(params['Function'] == fun)
    ]

best_predictor = ClassWisePredictor(score_function=SAPS(best_model_params['Lambda'].
    iloc[0]), model=model, temperature=best_model_params['Best_Temperature'].iloc[0])
best_predictor.calibrate(cal_loader, alpha)

# %%
best_predictor.evaluate(test_loader)

# %%
# Placeholder lists for true labels and probabilities
true_labels = []
base_model_probabilities = []
conformal_model_probabilities = []

# Placeholder lists for predictions
base_model_pred_flattened = []
conformal_pred_labels = []

# Additional placeholders for filtering by size 1 prediction sets
filtered_true_labels = []
filtered_conformal_pred_labels = []
total_conformal_sets = 0
size_1_conformal_sets = 0

# Populate the true_labels and predictions
for inputs, targets in test_loader:
    # Move to GPU
    inputs, targets = inputs.to(DEVICE, dtype=torch.float), targets.to(DEVICE, dtype=
        torch.int64)

    # Collect true labels
    true_labels.extend(targets.tolist())

    # Base Model Predictions
    base_probs = model(inputs).detach()
    base_model_probabilities.extend(base_probs.cpu().numpy())
    _, predictions = torch.max(base_probs, 1)
    base_model_pred_flattened.extend(predictions.tolist())

    # Conformal Model Predictions
    pred_sets = best_predictor.predict(inputs)
    conformal_probs = best_predictor.predict_probabilities(inputs)
    conformal_model_probabilities.extend(conformal_probs.cpu().numpy())

    # Select the prediction with the highest probability or fallback to 1
    for i, (pred_set, probs) in enumerate(zip(pred_sets, conformal_probs)):
        total_conformal_sets += 1

        if len(pred_set) == 1:
            # Size-1 prediction set
            size_1_conformal_sets += 1
            conformal_pred_labels.append(pred_set[0])
            filtered_true_labels.append(targets[i].item())
            filtered_conformal_pred_labels.append(pred_set[0])
```

```
        else:
            # Move probs to CPU and convert to NumPy before applying argmax
            if len(pred_set) > 0:
                highest_prob_label = pred_set[np.argmax(probs[pred_set].cpu().numpy())]
                conformal_pred_labels.append(highest_prob_label)
            else:
                conformal_pred_labels.append(1)  # Fallback to 1 if no valid prediction

# Convert to numpy arrays
true_labels = np.array(true_labels)
base_model_probabilities = np.array(base_model_probabilities)
conformal_model_probabilities = np.array(conformal_model_probabilities)
base_model_pred_flattened = np.array(base_model_pred_flattened)
conformal_pred_labels = np.array(conformal_pred_labels)
filtered_true_labels = np.array(filtered_true_labels)
filtered_conformal_pred_labels = np.array(filtered_conformal_pred_labels)

# Calculate Brier score for each class and average them for the base model
brier_base = np.mean([
    brier_score_loss(true_labels == i, base_model_probabilities[:, i])
    for i in range(base_model_probabilities.shape[1])
])

# Calculate Brier score for each class and average them for the conformal model
brier_conformal = np.mean([
    brier_score_loss(true_labels == i, conformal_model_probabilities[:, i])
    for i in range(conformal_model_probabilities.shape[1])
])

# Calculate log loss for base model
log_loss_base = log_loss(true_labels, base_model_probabilities)

# Calculate log loss for conformal model
log_loss_conformal = log_loss(true_labels, conformal_model_probabilities)

# Calculate model-level accuracy
accuracy_base_model = accuracy_score(true_labels, base_model_pred_flattened)
accuracy_conformal_model = accuracy_score(true_labels, conformal_pred_labels)

# Calculate model-level accuracy for filtered conformal model
accuracy_filtered_conformal_model = accuracy_score(filtered_true_labels, filtered_conformal_pred_labels) if len(filtered_true_labels) > 0 else np.nan

# Percentage of size-1 sets
percentage_size_1 = (size_1_conformal_sets / total_conformal_sets) * 100

# Function to calculate metrics for a given label
def calculate_metrics(true, pred, label):
    accuracy = accuracy_score(true == label, pred == label)
    precision = precision_score(true, pred, labels=[label], average='macro', zero_division=0)
    recall = recall_score(true, pred, labels=[label], average='macro', zero_division=0)
    f1 = f1_score(true, pred, labels=[label], average='macro', zero_division=0)
    return accuracy, precision, recall, f1
```

```python

# Collect metrics in dictionaries
metrics_base = {'Label': [], 'Accuracy': [], 'Precision': [], 'Recall': [], 'F1-Score': []}
metrics_conformal = {'Label': [], 'Accuracy': [], 'Precision': [], 'Recall': [], 'F1-Score': []}
metrics_filtered_conformal = {'Label': [], 'Accuracy': [], 'Precision': [], 'Recall': [], 'F1-Score': []}

for label in [0, 1, 2]:
    # Metrics for Base Model
    accuracy, precision, recall, f1 = calculate_metrics(true_labels, base_model_pred_flattened, label)
    metrics_base['Label'].append(label)
    metrics_base['Accuracy'].append(accuracy)
    metrics_base['Precision'].append(precision)
    metrics_base['Recall'].append(recall)
    metrics_base['F1-Score'].append(f1)

    # Metrics for Unfiltered Conformal Model
    accuracy, precision, recall, f1 = calculate_metrics(true_labels, conformal_pred_labels, label)
    metrics_conformal['Label'].append(label)
    metrics_conformal['Accuracy'].append(accuracy)
    metrics_conformal['Precision'].append(precision)
    metrics_conformal['Recall'].append(recall)
    metrics_conformal['F1-Score'].append(f1)

    # Metrics for Filtered Conformal Model
    if len(filtered_true_labels) > 0:  # Ensure there are filtered predictions
        accuracy, precision, recall, f1 = calculate_metrics(filtered_true_labels, filtered_conformal_pred_labels, label)
    else:
        accuracy = precision = recall = f1 = np.nan

    metrics_filtered_conformal['Label'].append(label)
    metrics_filtered_conformal['Accuracy'].append(accuracy)
    metrics_filtered_conformal['Precision'].append(precision)
    metrics_filtered_conformal['Recall'].append(recall)
    metrics_filtered_conformal['F1-Score'].append(f1)

# Convert dictionaries to DataFrames for display
df_metrics_base = pd.DataFrame(metrics_base)
df_metrics_conformal = pd.DataFrame(metrics_conformal)
df_metrics_filtered_conformal = pd.DataFrame(metrics_filtered_conformal)

# Print the percentage of size-1 sets
print(f"Percentage of prediction sets of size 1: {percentage_size_1:.2f}, Alpha: {alpha}%")
print('\n')
# Print Brier score and Log loss
print(f"Brier Score (Base Model): {brier_base:.4f}")
print(f"Brier Score (Conformal Model): {brier_conformal:.4f}")
print('\n')
print(f"Log Loss (Base Model): {log_loss_base:.4f}")
print(f"Log Loss (Conformal Model): {log_loss_conformal:.4f}")
print('\n')
```

```python
# Print model-level accuracy
print(f"Model-Level Accuracy (Base Model): {accuracy_base_model:.4f}")
print(f"Model-Level Accuracy (Conformal Model): {accuracy_conformal_model:.4f}")
print(f"Model-Level Accuracy (Filtered Conformal Model): {
    accuracy_filtered_conformal_model:.4f}")

# Display the metrics for the Base Model
print("\nBase Model Metrics:")
print(df_metrics_base.to_string(index=False))

# Display the metrics for the Unfiltered Conformal Model
print("\nConformal Model Metrics (Unfiltered):")
print(df_metrics_conformal.to_string(index=False))

# Display the metrics for the Filtered Conformal Model
print("\nFiltered Conformal Model Metrics (Only considering size-1 sets):")
print(df_metrics_filtered_conformal.to_string(index=False))
```

Listing 7: evaluation.ipynb

References

Muhammad Amjad and Xiaoyu Zhou. Applying conformal prediction to forecasting energy prices with machine learning. *International Journal of Forecasting*, 38(2):324–340, 2022.

Anastasios N Angelopoulos, Stephen Bates, Clara Fannjiang, and Michael I Jordan. Learn then test: Calibrating predictive algorithms to achieve risk control. *NeurIPS*, 35:12352–12364, 2022.

Paul Bilokon, Manfred Mudelsee, and Volodymyr Bilokon. Transformers vs lstms: A comparative study for financial time series prediction. *Quantitative Finance*, 23(4):665–682, 2023.

Glenn W Brier. Verification of forecasts expressed in terms of probability. *Monthly weather review*, 78 (1):1–3, 1950.

Maxime Cauchois, Lorenzo Rosasco, and Chiyuan Zhou. A probabilistic interpretation of adaptive prediction sets with conformal prediction. *Journal of Machine Learning Research*, 22:1–30, 2021.

Lawrence R Glosten and Paul R Milgrom. Bid, ask and transaction prices in a specialist market with heterogeneously informed traders. *Journal of financial economics*, 14(1):71–100, 1985.

Irving J Good. Rational decisions. *Journal of the Royal Statistical Society: Series B (Methodological)*, 14(1):107–114, 1952.

Martin D Gould, Mason A Porter, Stacy Williams, Mark McDonald, Daniel J Fenn, and Sam D Howison. Limit order books. *Quantitative Finance*, 13(11):1709–1742, 2013.

Albert Gu, Ashish Vaswani, Anwar Khan, and Hao Zhou. Mamba: A new deep learning architecture for financial time series forecasting. *IEEE Transactions on Neural Networks and Learning Systems*, 34(5):1325–1336, 2023.

Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q Weinberger. On calibration of modern neural networks. *International Conference on Machine Learning*, 70:1321–1330, 2017.

Li Huang, Guangming Cui, Chunxiang Lin, and Shaolong Liu. Attnlob: A dual-stream self-attention-based model for predicting stock prices using limit order book data. *IEEE Transactions on Neural Networks and Learning Systems*, 32(5):1700–1709, 2021.

Albert S Kyle. Continuous auctions and insider trading. *Econometrica: Journal of the Econometric Society*, pages 1315–1335, 1985.

Ernst Luetkebohmert, Ulrich Schmock, and Peter Welz. Market making with conformal prediction: Forecast intervals for net positions. *Journal of Financial Markets*, 50:123–145, 2021.

Allan H Murphy. A new vector partition of the probability score. *Journal of applied Meteorology*, 12 (4):595–600, 1973.

Avraam Ntakaris, Matteo Magris, Juho Kanniainen, Moncef Gabbouj, and Alexandros Iosifidis. Benchmark dataset for mid-price forecasting of limit order book data with machine learning methods. *Journal of Forecasting*, 37(8):852–866, 2018.

OpenAI. Gpt-4. https://openai.com/gpt-4, 2024. Language model used for writing assistance.

Maureen O'Hara. Limit order book as a market for liquidity. *Journal of Finance*, 60(1):665–729, 2005.

David Martin Powers. Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. *Journal of Machine Learning Technologies*, 2(1):37–63, 2011.

Carlos Riquelme, Cristian Esteban, and Chelsea Finn. Torchcp: A conformal prediction library for pytorch. *Journal of Machine Learning Research*, 23:1–6, 2022.

Yaniv Romano, Matteo Sesia, and Emmanuel J Candès. Classification with a reject option using adaptive prediction sets. *Journal of the American Statistical Association*, 115(532):1890–1903, 2020.

Weijie Shi, Assaf Schuster, and Matan Gavish. Class-wise predictor for conformal prediction. *Journal of Machine Learning Research*, 14:2991–3007, 2013.

Justin Sirignano and Rama Cont. Deep learning for limit order books. *Quantitative Finance*, 19(4): 549–570, 2019.

Avraam Tsantekidis, Nikolaos Passalis, Anastasios Tefas, Juho Kanniainen, Moncef Gabbouj, and Alexandros Iosifidis. Forecasting stock prices from the limit order book using convolutional neural networks. *IEEE Transactions on Signal Processing*, 68:344–358, 2020.

Vladimir Vovk, Alexander Gammerman, and Glenn Shafer. *Algorithmic learning in a random world*. Springer, 2005.

Ling Ye, Zhenyu Ma, Yuan Ma, Xunyu Wang, Jia He, Xiao Yang, and Tie-Yan Liu. Lob-net: A deep neural network for modeling high-frequency limit order book dynamics. *Journal of Financial Data Science*, 2(4):100–116, 2020.

Li Zhang and Ming Li. On the applicability of mamba models for financial time series prediction. *Journal of Financial Data Science*, 4(3):245–265, 2023.

Zihao Zhang, Stefan Zohren, and Stephen Roberts. Deeplob: Deep convolutional neural networks for limit order books. *IEEE Transactions on Signal Processing*, 67(11):3001–3012, 2019.