A Data Collection Procedure

In this section, we describe the details of the collection of our dataset. We have provided the implementation in our code repository: https://github.com/VirtuosoResearch/ML4RoadSafety. We have also uploaded the entire dataset to an open repository: https://doi.org/10.7910/DVN/V71K5R. This section serves as the documentation for the collection process. For reference, we have collected the links to public data sources where we extracted our dataset below.

Table 5: Links to the data sources from which we extracted our dataset.

Traffic accident records	
Delaware Open Data	https://data.delaware.gov/Transportation/Public-Crash-Data-Map/3rrv-8pfj
Delaware DOT	https://deldot.gov/search/
Iowa DOT	https://icat.iowadot.gov/
Illinois DOT	https://gis-idot.opendata.arcgis.com/search?collection=Dataset&q=Crashes
Mass DOT	https://apps.impact.dot.state.ma.us/cdp/home
Maryland Open Data	https://opendata.maryland.gov/Public-Safety/Maryland-Statewide-Vehicle-Crashes/65du-s3qu
MN Crash	https://mncrash.state.mn.us/Pages/AdHocSearch.aspx
Montana DOT	https://www.mdt.mt.gov/publications/datastats/crashdata.aspx
Nevada DOT	https://ndot.maps.arcgis.com/apps/webappviewer/index.html?id=00d23dc547eb4382bef9beabe07eaefd
Road networks	
OSMnx Street Network Dataverse	https://dataverse.harvard.edu/dataset.xhtml?persistentId=doi:10.7910/DVN/CUWWYJ
OpenStreetMap Road Categories	https://wiki.openstreetmap.org/wiki/Highways#Classification
Google Map API	https://maps.google.com/
Weather reports	
Meteostat API	https://meteostat.net/en/
Traffic volume reports	
Delaware	https://deldot.gov/search/
Maryland	https://data-maryland.opendata.arcgis.com/datasets/mdot-sha-annual-average-daily-traffic-aadt-segments/explore?location=38.833256%2C-77.269751%2C8.30&showTable=true
Massasuchetts	https://mhd.public.ms2soft.com/tcds/tsearch.asp?loc=Mhd&mod= https://www.mass.gov/lists/massdot-historical-traffic-volume-data
Nevada	https://geohub-ndot.hub.arcgis.com/datasets/trina-stations/explore?location=38.490765%2C-116.969086%2C7.27&showTable=true

A.1 Traffic accident records

First, we construct the accident labels in our dataset. We collect accident records for each state from the state's Department of Transportation website. Each accident is associated with a report detailing the date and coordinates (i.e., latitude and longitude) of the accident. We collect accident data from eight states. The records are available in the sources listed in Table 5. Then, we map each accident to the closest road in the road network according to the coordinates of the accident. Specifically, we use the accident's location, denoted as c, and compare it with the coordinates of an edge's two endpoints, labeled as a and b. We map the accident to the edge with the smallest value of $D(a,b) - (D(a,c) + D(b,c))$ where $D(\cdot)$ is the Euclidean distance metric.

A.2 Road networks

We generate a road network for a state as a directed graph, based on the road network structure in the state extracted from OpenStreetMap. Nodes are defined as all the publicly available intersections on the roads in OpenStreetMap, and edges are the road segments between these nodes. Therefore, one road can have multiple edges depending on the number of intersections with other roads. There would be three edges if there are three intersections on a road.

The derived edges in the graph include road segments of various levels in a state, including every city, town, urbanized area, county, census tract, and Zillow neighborhood in the state. We obtain the above information from the OSMnx Street Networks Dataverse in OpenStreetMap (cf. Table 5).

A.3 Road network features

We describe four types of features associated with road networks in the following.

Graph structural features (Node-level, static). All nodes in a graph are associated with static structural features, including node in-degrees, out-degrees, and betweenness centrality scores. For node degrees, we encode node degree values as one-hot vectors with a dimension of the graph's max degree. We encode two vectors for every node's in-degree and out-degree. For betweenness centrality scores, we generate a real-value feature that measures the ratio of shortest paths between all node pairs that contain a certain node.

Historical weather information (Node-level, temporal). Besides, each node is also associated with daily weather information. We collect the following six features related to the weather conditions within a particular month: (i-iii) the maximum, minimum, and average of the temperature on the road's surface; (iv) the total precipitation, including rainfall or snowfall; (v) the average wind speed; (vi) the sea level air pressure. For every node, All the weather conditions are measured at the nearest meteorological station to the node according to its geometric coordinate. The weather information is extracted using Meteostat API (cf. Table 5). Since this data is temporal in nature and is available at every node, we believe that the weather features would be important for our task.

Road information (Edge-level, static). All edges are associated with static features describing length information and road categories. We encode the physical length of the road in meters as a real-value feature. We encode the road category as a 24-dimensional characteristic vector for each edge, with each entry as either 0 or 1, indicating whether the road is associated with the category. Each road can be assigned (multiple) categories from 24 road categories. These include a oneway, primary, primary link, secondary, secondary link, access ramp, bus stop, crossroad, disused, elevator, escape, living street, motorway, motorway link, residential, rest area, road, stairs, tertiary, tertiary link, trunk, trunk link, unsurfaced, and unclassified. We obtain the road information from OpenStreetMap.

Traffic volume (Edge-level, temporal). Each edge is also associated with a traffic volume feature that is measured yearly. We encode the traffic volume as a real-value feature that measures the average number of cars traveled on the road over a year. We extract the information from the Annual Average Daily Traffic reports published by the Department of Transportation of each state. These reports provide information on the traffic volume for a sample of streets within the state. By using the Google Maps API, we extract the corresponding coordinates for the street names and map them to the edges of the road network.

A.4 Summary of road network features

Here is a list of node-level features we have included in our dataset:

- Latitude,
- Longitude,
- Node indegree and outdegree,
- Betweenness centrality,
- Average surface temperature (tavg),
- Max surface temperature (tmax),
- Min surface temperature (tmin),

- Total precipitation (prcp),
- Avg wind speed (wspd),
- Sea level air pressure (pres).

Here are the edge-level features in our dataset:

- A binary label that indicates whether the road is one way or not,
- A multi-class label that indicates whether the road is highway, residential, etc.,
- Length of the road,
- Annual average daily traffic (AADT), if this information is available in the report.

B Experiment Details

In this section, we describe the details of our experiments that were left out in the main text. First, we describe additional implementation details. Then, we describe the omitted experimental results. These include the ablation study of graph structural features, transferability from traffic volume to accident prediction, various metrics for evaluating accident prediction, and observations of accident counts across seasons.

B.1 Implementation details

In our implementation, we set the dimension of node embeddings as 128, the number of layers as 2, and the hidden dimensionality as 256. We train our models using Adam as the optimizer. We use a learning rate of 0.001 for 100 epochs on all models. The hyper-parameters are determined by searching in the following ranges: The hidden dimensionality is tuned in a range of $\{128, 256, 512\}$. The number of layers is tuned in a range of $\{2, 3, 4\}$. The learning rate is tuned in a range of $\{0.01, 0.001, 0.0001\}$.

For each state, we evenly split the available period of accidents into training, validation, and test set. Table 6 summarizes the dataset splitting for each state. While our evaluation focuses on monthly predictions, our datasets can also be utilized for conducting analyses at daily or annual levels.

Table 6: Data splitting of accident records of eight states.

	Train (years)	Train (records)	Valid (years)	Valid (records)	Test (years)	Test (records)
DE	2009 - 2012	112,670	2013 - 2017	174,278	2018 - 2022	171,311
IA	2013 - 2016	213,019	2017 - 2019	171,455	2020 - 2022	156,065
IL	2012 - 2014	856,057	2015 - 2017	949,745	2018 - 2021	1,174,900
MA	2002 - 2008	1,265,895	2009 - 2015	933,786	2016 - 2022	1,096,885
MD	2015 - 2017	341,902	2018 - 2019	229,446	2020 - 2022	306,995
MN	2015 - 2017	148,361	2018 - 2019	154,150	2020 - 2022	188,858
MT	2016 - 2017	40,040	2018	20,677	2019 - 2020	39,222
NV	2016 - 2017	101,975	2018	48,854	2019 - 2020	86,509

B.2 Additional experimental results

Distilling graph structural features. We conduct a leave-one-out analysis of different categories of features for accident prediction. These include the graph-structural features. weather, and traffic volume information. We evaluate the performance of removing one type of feature at one time. Table 7 reports the results of this ablation study. We notice that removing the graph-structural features reduces the performance the most. Thus, we conclude that the graph-structural features are the most important features for accident prediction.

Tranferability from traffic volume to accident prediction. Next, we report the results of transferring a model from traffic volume to accident prediction. For one state, we first fit a model on the volume prediction task and then fine-tune the model on the accident prediction task. Table 7 reports the performance of the fine-tuned model on accident prediction tasks for four states with traffic volume information. The fine-tuned model outperforms STL by 0.6% on average over the four states.

Table 7: We ablate the influence of graph-structural structure, weather, and traffic volume information. We report the test AUROC of accident classification by removing each feature type in the network. We also include the results of transferring a model from traffic volume to accident prediction.

	DE	MA	MD	NV
Using all features	87.61±0.10	81.80±0.12	87.51±0.02	91.62±0.99
w/o graph structural features	81.25±0.51	79.63±0.18	79.77±0.94	82.56±0.09
w/o weather information	87.27±0.32	80.71±0.26	80.22±0.45	90.38±0.88
w/o road information	82.99±0.54	81.65±0.77	80.63±0.52	84.24±0.41
w/o traffic volume information	87.15±0.49	80.94±0.31	86.17±1.15	91.58±0.99
Transfer from traffic volume prediction	87.78±0.07	82.18±0.14	87.77±0.24	92.07±0.57

Table 8: We report the precision and recall scores on the test split on eight states, using node embedding methods, graph neural networks, and multitask and transfer learning methods. We run the experiment over three different random seeds and report the averaged result and standard deviations.

Precision	DE	IA	IL	MA	MD	MN	MT	NV
Training Size	93,184	187,046	646,739	540,682	283,226	124,435	34,475	73,164
Positive Rate	0.23	0.07	0.14	0.10	0.15	0.05	0.05	0.12
MLP	4.99±0.1	1.78±0.0	2.54±0.2	3.52±0.6	3.47±0.1	1.87±0.0	0.87±0.0	3.30±0.0
Node2Vec	12.76±0.2	3.16±0.3	3.52±0.5	3.28±0.8	5.64±0.4	2.38±0.3	3.50±0.2	10.74±0.0
DeepWalk	14.13±0.5	2.90±0.5	3.37±0.6	3.30±0.1	4.88±0.1	2.64±0.3	2.62±0.4	7.74±0.0
GCN	11.09±0.7	2.36±0.1	7.54±0.7	4.05±0.4	5.60±0.1	4.60±0.1	5.27±0.2	9.17±0.1
GraphSAGE	18.56±0.9	4.13±0.2	8.54±0.3	4.71±0.2	7.04±0.1	6.26±0.4	4.11±0.5	8.62±0.0
GIN	13.95±0.6	3.63±0.2	9.84±0.8	4.45±0.8	6.50±0.5	4.08±0.7	5.72±0.7	10.27±0.0
AGCRN	11.36±0.6	3.50±0.2	7.60±1.1	4.30±0.3	6.61±0.2	5.66±0.2	3.12±0.1	7.74±0.1
STGCN	12.33±0.2	5.20±0.2	7.14±1.7	4.54±0.9	8.91±0.1	4.65±0.1	4.04±0.9	9.72±0.9
Graph Wavenet	15.56±0.5	3.83±0.3	7.76±0.7	4.22±1.6	7.21±0.3	6.62±0.5	3.36±0.1	7.84±0.1
DCRNN	11.33±0.5	3.87±0.4	6.16±0.1	4.67±0.1	4.75±0.4	3.62±0.1	3.66±0.4	10.06±0.1
STL w/ GraphSAGE	18.56±0.9	4.13±0.2	8.54±0.3	4.71±0.2	7.04±0.1	6.26±0.4	4.11±0.5	8.62±0.0
MTL w/ GraphSAGE	14.34±0.1	3.52±0.4	**13.62±0.7**	5.11±0.1	8.66±0.0	4.44±0.2	**5.85±0.3**	16.80±0.0
MTL-FT w/ GraphSAGE	**18.61±0.1**	**4.66±0.0**	13.61±0.5	**5.20±0.1**	**8.76±0.1**	**6.66±0.3**	5.72±0.0	**16.85±0.4**
TL w/ GraphSAGE	14.10±0.4	-	-	5.07±0.4	8.51±0.4	-	-	9.5±0.1
Recall	DE	IA	IL	MA	MD	MN	MT	NV
MLP	60.4±3.7	83.8±2.8	81.1±2.5	79.9±1.4	67.2±2.0	70.3±1.2	72.0±1.6	74.3±0.4
Node2Vec	**83.8±1.3**	**89.5±1.6**	78.2±0.6	80.1±3.1	80.0±2.3	**80.0±2.2**	73.6±0.9	83.9±0.5
DeepWalk	83.2±2.9	80.7±4.8	81.0±1.1	85.7±3.1	**86.2±3.6**	78.4±2.9	74.0±3.3	**88.9±0.0**
GCN	75.2±3.0	74.0±2.0	**84.1±0.8**	82.5±2.1	79.1±2.6	70.7±1.8	63.6±3.8	85.4±0.8
GraphSAGE	60.9±2.7	58.5±2.0	72.7±2.4	51.0±1.3	68.7±1.1	54.5±2.2	59.6±2.9	84.7±1.1
GIN	65.8±3.7	75.0±2.4	78.0±4.3	48.2±2.0	78.3±3.2	74.8±2.4	65.6±1.8	88.6±0.9
AGCRN	71.5±1.6	71.7±1.9	73.4±1.0	58.2±1.6	75.4±1.5	73.1±1.5	70.7±1.3	82.6±3.1
STGCN	82.9±0.2	78.2±4.7	75.3±1.2	60.7±1.3	77.6±1.0	75.8±0.3	72.6±1.1	83.0±0.0
Graph Wavenet	73.5±3.0	67.7±0.7	78.3±0.2	77.7±1.1	73.0±0.2	72.9±3.2	66.1±1.0	79.3±0.1
DCRNN	75.4±2.6	71.1±2.2	80.5±1.7	81.5±2.0	83.0±1.4	63.6±1.3	66.3±1.0	83.4±0.8
STL w/ GraphSAGE	60.9±2.7	58.5±2.0	72.7±2.4	51.0±1.3	68.7±1.1	54.5±2.2	59.6±2.9	84.7±1.1
MTL w/ GraphSAGE	63.8±1.8	78.6±1.0	75.6±1.3	75.5±0.9	78.1±1.2	77.6±0.7	74.2±0.9	82.3±0.5
MTL-FT w/ GraphSAGE	64.5±0.7	78.2±2.5	76.7±1.2	73.7±1.1	77.4±2.1	73.2±0.8	**74.3±3.6**	84.9±1.2
TL w/ GraphSAGE	66.7±1.4	-	-	**92.7±1.3**	81.7±1.6	-	-	84.1±3.3

Detailed results of classification metrics. Next, we report the detailed results of classifying whether an accident occurred on a particular road. Table 8 reports the recall and precision scores of the predictions. First, we observe that for all baselines, the recall scores are higher than the precision scores. Using the graph neural networks can predict whether an accident occurred on a road with 10% precision and 85% recall on average over eight states. Second, we also observe that multitask learning outperforms STL relatively by 20% and 21% in terms of precision and precision, respectively. Third, combining traffic volume with accident prediction also improves the STL relatively by 4% and 15% in terms of precision and recall scores.

We observe that while using MLP on node embeddings achieves higher recall than graph neural networks, graph neural networks achieve higher precision scores. This indicates that MLP models tend to be more over-confident when classifying the likelihood of an accident occurring on a road. Given the low precision score, we report the AUROC score in the main text, which provides a summary of the recall score across all decision thresholds.

Table 9: Ablation study of different hyper-parameters, including the number of layers, the hidden dimensionality, the learning rate, and training epochs. We report the AUROC scores on the validation split on the Delaware (DE) state dataset using GraphSAGE and DCRNN. We run the same experiment over three random seeds and report the average result.

	GraphSAGE			DCRNN		
Number of layers	2	3	4	2	3	4
	85.2±0.1	84.9±0.3	84.4±0.4	**67.8**±1.2	67.2±0.8	67.3±0.5
Hidden dimensionality	128	256	512	128	256	512
	84.5±0.4	**85.2**±0.1	84.5±0.5	66.9±0.7	**67.8**±1.2	66.9±1.1
Learning rate	$1e^{-2}$	$1e^{-3}$	$1e^{-4}$	$1e^{-2}$	$1e^{-3}$	$1e^{-4}$
	85.0±0.7	**85.2**±0.1	84.0±0.5	66.8±1.0	**67.8**±1.2	66.5±0.9
Epochs	50	100	200	50	100	200
	84.0±0.2	**85.2**±0.1	**85.2**±0.3	66.4±0.7	**67.8**±1.2	**67.8**±1.0

Table 10: We report the results of applying graph contrastive learning on our datasets using GraphSAGE and DCRNN. We report the AUROC scores on the test split across four states. We run the same experiment over three random seeds and report the average result.

	DE	MA	MD	NV
GraphSAGE	87.6±0.1	81.8±0.1	87.5±0.0	91.6±0.9
GraphSAGE w/ GCL	86.7±0.2	82.4±0.8	85.9±0.4	91.8±0.4
DCRNN	81.2±1.2	70.5±0.1	84.5±0.3	90.5±0.7
DCRNN w/ GCL	86.6±0.3	82.5±0.7	87.8±0.9	91.7±0.4
STGCN	85.4±0.1	81.9±0.3	88.7±0.1	91.5±0.3
STGCN w/ GCL	86.0±0.1	81.7±0.1	89.7±0.5	92.4±0.1

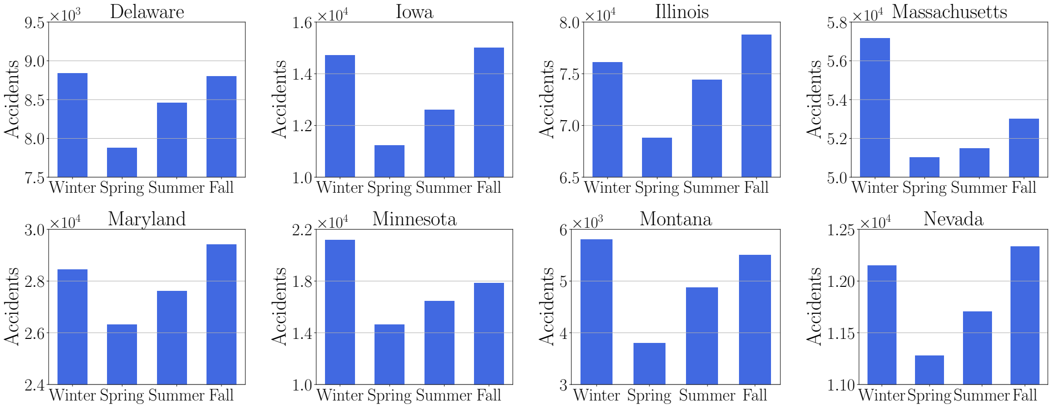
Detailed results of hyper-parameter tuning. We ablate the common hyper-parameters used in our experiments, including the number of layers, the hidden dimensionality, the learning rate, and the number of epochs. In each ablation study, we vary one hyper-parameter and keep the others unchanged. The fixed hyper-parameters are used as follows: the number of layers of 2, a hidden dimensionality of 256, a learning rate of $1e^{-3}$, and 100 epochs.

Table 9 shows the validation AUROC scores, varying hyper-parameters for GraphSAGE and DCRNN on the Delaware state dataset. We notice that using the number of layers as 2, hidden dimensionality as 256, and learning rate as $1e^{-3}$ yields the best results for both baselines. The validation performance stops improving after training the model up to 100 epochs. We also find that these hyper-parameters are useful for other models. Thus, we adopt these settings as the default parameters in the experiments.

Applying graph contrastive learning. We conduct a preliminary study of applying graph contrastive learning using GraphSAGE and DCRNN as the base model across four states. We compare them with supervised learning of the baselines. Table 10 shows the results. We find that graph contrastive learning can improve the test performance of the baselines in some states.

Seasonal patterns of road accidents. We study how the number of accidents evolves within a year and explore its potential association with seasonal patterns. We hypothesize that the accidents may be affected by the weather and show seasonal trends. To examine this, we aggregate accident counts within four seasons in a year for each state. Specifically, we aggregate accidents occurring from December to February for winter, from March to May for spring, from June to August for summer, and from September to November for fall. Figure 2 shows trends of accident numbers across the seasons for the four states. We notice a significant disparity in accident counts between winter and fall compared to spring and summer. The disparity suggests that accidents may indeed be influenced by seasonal factors, including severe weather conditions and road hazards that are more prevalent during the colder months.

Figure 5: Seasonal trend of accident counts within a year. We observe higher accident counts during Winter and Fall compared to Spring and Summer.



Number of parameters used by each model: This information complements our discussion in Section 3.2:

- GraphSAGE: 115K,
- AGCRN: 219K,
- STGCN: 638K,
- Graph WaveNet: 1079K,
- DCRNN: 1388K.

B.3 Examples for Using the ML4RoadSafety Package

We provide examples for accessing the data and training models using our dataset. Our package uses the same data format as the existing graph learning library, i.e., PyTorch Geometric, which is fully compatible with PyTorch. As shown in Code Snippet 1, our package provides access to our dataset with a single line of code, where the user only needs to specify the name of a state, such as Massachusetts. The package will automatically download, store, and return the dataset object. Then, the user can use a function to obtain the accident records and network features for a particular month.

```
>>> from ml_for_road_safety import TrafficAccidentDataset
# Creating the dataset as PyTorch Geometric dataset object
>>> dataset = TrafficAccidentDataset(state_name = "MA")
# Loading the accident records and traffic network features of a particular month
>>> data = dataset.load_monthly_data(year = 2022, month = 1)
# Pytorch Tensors storing the list of edges with accidents and accident numbers
>>> accidents, accident_counts = data["accidents"], data["accident_counts"]
# Pytorch Tensors of node features, edge list, and edge features
>>> x, edge_index, edge_attr = data["x"], data["edge_index"], data["edge_attr"]
```

Code Snippet 1: ML4RoadSafety Data Loader

Our package provides a trainer class that implements training a graph neural network on one state in our dataset. The trainer class contains the logic for both training and evaluation. As shown in Code Snippet 2, the user can create a trainer object by specifying a model, a dataset, and a corresponding evaluation metric. Then, the user can use a function to launch the training and obtain evaluation results after the training process.

```
>>> from ml_for_road_safety import Trainer, Evaluator, TrafficAccidentDataset
# Creating the dataset
>>> dataset = TrafficAccidentDataset(state_name = "MA")
# Get an evaluator for accident prediction, e.g., the classification task.
>>> evaluator = Evaluator(type = "classification")
```

```
# Initialize a trainer with a GNN model, a dataset, and an evaluator
>>> trainer = Trainer(model, dataset, evaluator, ...)
# Conduct training and evaluation inside the trainer
>>> log = trainer.train()
```

Code Snippet 2: ML4RoadSafety Trainer

B.3.1 Implementation of Multitask and Transfer Learning

In multitask learning, we combine multiple datasets and optimize the average loss of the combined data. We implement this as follows. First, one trainer is created for every dataset. Then, we can optimize the average loss by iterating through every task trainer in one epoch. The logic is shown in Code Snippet 3,

```
# Create a trainer for every task
>>> self.task_to_trainers = {}
>>> for task_name in tasks:
>>>     self.task_to_trainers[task_name] = Trainer(model, dataset, evaluator, ...)
# Optimize the average loss by iterating over all task trainers in each epoch.
>>> for epoch in range(1, 1 + epochs):
>>>     for task_name in task_list:
#           Each task trainer optimizes the loss of the task itself
>>>         task_trainer = self.task_to_trainers[task_name]
>>>         task_trainer.train_epoch()
```

Code Snippet 3: Implementation of Multitask Learning

To make this easy to use, we have wrapped the logic into a multitask trainer. As shown in Code Snippet 4, the user can create a multitask trainer by specifying a model and providing a list of datasets. After that, the user can train a multitask model using a single function.

```
>>> from ml_for_road_safety import MultitaskTrainer
# Specify the tasks that are combined in multitask learning
>>> task_list = ["MA_accident_classification", "MD_accident_classification", ...]
>>> task_datasets = {}; task_evaluators = {}
>>> for task_name in task_list:
>>>     state_name, data_type, task_type = task_name.split("_")
>>>     task_datasets[task_name] = TrafficAccidentDataset(state_name = state_name)
>>>     task_evaluators[task_name] = Evaluator(type=task_type)
# Initialize a trainer with a GNN model, multiple datasets, and multiple evaluators
>>> trainer = MultitaskTrainer(model, tasks = task_list,
    task_to_datasets=task_datasets, task_to_evaluators=task_evaluators, ...)
# Conduct multitask learning and evaluation for every task
>>> trainer.train()
```

Code Snippet 4: ML4RoadSafety Multitask Trainer

We use transfer learning to transfer knowledge from traffic volume information to traffic accident prediction. We implement this using our package by training traffic volume and traffic accident prediction simultaneously in a multitask model. As shown in Code Snippet 5, the user can create a multitask trainer to train a model on the accident and volume prediction tasks from one state.

```
>>> from ml_for_road_safety import MultitaskTrainer
# Specify the accident and volume prediction tasks from one state
>>> task_list = ["MA_accident_classification", "MA_volume_regression"]
>>> task_datasets = {}; task_evaluators = {}
>>> for task_name in task_list:
>>>     state_name, data_type, task_type = task_name.split("_")
>>>     task_datasets[task_name] = TrafficAccidentDataset(state_name = state_name)
>>>     task_evaluators[task_name] = Evaluator(type=task_type)
# Initialize a trainer with a GNN model and two tasks of accident and volume
    prediction.
```

```
>>> trainer = MultitaskTrainer(model, tasks = task_list,
    task_to_datasets=task_datasets, task_to_evaluators=task_evaluators, ...)
# Conduct multitask learning and evaluation for both tasks
>>> trainer.train()
```

Code Snippet 5: Implementation of Transfer Learning

B.3.2 Applying Graph Contrastive Learning on Our Dataset

Our package can be easily extended to incorporate graph contrastive learning methods in traffic accident prediction on our datasets. For example, we can implement graph contrastive learning [67] on our datasets with a few lines of code. As shown in Code Snippet 6, one can define a trainer for contrastive learning by modifying the training loss in the base trainer class. Then, the user can use the trainer to conduct contrastive learning on our dataset.

```
>>> from ml_for_road_safety import Trainer
# Define a trainer for contrastive learning inherited from the base Trainer class
>>> class GraphContrastiveLearningTrainer(Trainer):
# Modify the training loss in the training logic
>>>     def train_epoch(self):
#           Define the contrastive loss
>>>         ...
>>>         loss = info_nce(outputs_1, outputs_2)
>>>         ...
# Initialize a contrastive learning trainer
>>> trainer = GraphContrastiveLearningTrainer(model, dataset, evaluator, ...)
# Conduct training and evaluation inside the trainer
>>> log = trainer.train()
```

Code Snippet 6: Implementation of graph contrastive learning

B.3.3 Implementation of Spatiotemporal Graph Neural Networks

Next, our package supports the evaluation of spatiotemporal graph neural networks. For example, as shown in Code Snippet 7, the user can create a spatiotemporal model using our package, such as STGCN, by specifying the corresponding model name. Our package includes the implementation of spatiotemporal models from an open-sourced repository, `pytorch-geometric-temporal`. Then, the user can create a trainer object that directly trains the model on a given dataset.

```
>>> from ml_for_road_safety import Trainer, GNN
# Create a spatiotemporal model, e.g., STGCN, from an online implementation
>>> model = GNN(encoder = "stgcn", ...)
# Initialize a trainer with the model and specify use_time_series as True
>>> trainer = Trainer(model, dataset, evaluator, use_time_series=True)
# Conduct training and evaluation inside the trainer
>>> log = trainer.train()
```

Code Snippet 7: Training a spatiotemporal model

B.3.4 Incorporating Advanced Multitask and Transfer Learning Techniques

Lastly, our package can be easily extended to incorporate advanced multitask and transfer learning techniques. We describe two examples in the following.

For multitask learning, we consider two task grouping methods, which identify tasks that would benefit from training together and train them in one neural network as a group. These methods include task affinity grouping (TAG) [13] and approximating higher-order task groupings (HOA) [53]. Our package can be extended to incorporate these methods with a few lines of code. As shown in Code Snippet 8, the user can obtain the task grouping by using the task grouping methods. Then, the user can use the multitask trainer to train a multitask model on each group of tasks.

```
>>> from ml_for_road_safety import MultitaskTrainer
# Generate task groupings from previous task grouping methods, such as TAG or HOA
>>> task_list = ["DE_accident_classification", "IL_accident_classification",
    "MA_accident_classification", "MD_accident_classification", ...]
>>> task_groups = group_tasks(method = "hoa", task_list)
# Generated task grouping is a list of grouped tasks
>>> task_groups = [
       ["DE_accident_classification", "IL_accident_classification", ...],
       ["MA_accident_classification", "MD_accident_classification", ...],
       ["IA_accident_classification", "NV_accident_classification", ...]
   ]
>>> for group_task_list in task_groups:
>>>     task_datasets = {}; task_evaluators = {}
>>>     for task_name in group_task_list:
>>>         state_name, data_type, task_type = task_name.split("_")
>>>         task_datasets[task_name] = TrafficAccidentDataset(state_name)
>>>         task_evaluators[task_name] = Evaluator(task_type)
#       Initialize a trainer with the combined datasets of a group
>>>     trainer = MultitaskTrainer(model, tasks = group_task_list,
        task_to_datasets=task_datasets, task_to_evaluators=task_evaluators, ...)
#       Conduct multitask learning on one group of tasks
>>>     trainer.train()
```

Code Snippet 8: Training multitask learning models on groups of tasks

For transfer learning, we consider two regularization methods for fine-tuning a model trained on a source task to a target task. These methods include soft penalty (SP) and sharpness-aware minimization (SAM). Soft penalty regularizes the fine-tuned model distances to the initial model weights. Sharpness-aware minimization simultaneously minimizes loss value and loss sharpness with regard to the model weights. For example, as shown in Code Snippet 9, one can define a trainer to add the soft penalty loss by modifying the training loss in the base trainer class. Then, the user can use the trainer to fine-tune a model with a soft penalty on the fine-tuned distances.

```
>>> from ml_for_road_safety import Trainer
# Define a trainer for soft penalty inherited from the base Trainer class
>>> class SoftPenaltyTrainer(Trainer):
# Modify the training loss in the training logic
>>>     def train_epoch(self):
#           Combine the soft penalty loss with the cross-entropy loss
>>>         ...
>>>         loss = cross_entropy_loss + \
                 lambda*add_soft_penalty(model, initial_state_dict)
>>>         ...
# Initialize a soft penalty trainer
>>> trainer = SoftPenaltyTrainer(model, dataset, evaluator, initial_state_dict, ...)
# Conduct training and evaluation inside the trainer
>>> log = trainer.train()
```

Code Snippet 9: Implementation of fine-tuning with soft penalties

We find that using these methods improves the test performance over simple methods by 0.6% on average over four states. More comprehensive evaluations of related methods are left for future work.

C Additional Related Works and Discussions

C.1 Spatiotemporal graph neural networks

Previous works have proposed spatiotemporal graph neural networks to capture spatial and temporal dependencies for time series analysis on graph-structured data. DCRNN [39] captures the spatial dependency using bidirectional random walks on the graph and the temporal dependency using the encoder-decoder architecture with scheduled sampling. Instead of applying regular convolutional and recurrent units, STGCN [68] builds the model with complete convolutional structures, which enable much faster training speed with fewer parameters. Bai et al. [1] propose AGCRN with two adaptive modules for enhancing GNNs, including one module to capture node-specific patterns and another to infer the inter-dependencies among different traffic series automatically. Wu et al. [62] develop Graph WaveNet with an adaptive dependency matrix to capture the hidden spatial dependency in the data and a dilated 1D convolution component to handle very long sequences. We refer interested readers to a comprehensive survey [58] on spatio-temporal models.

C.2 Graph contrastive learning

The use of contrastive learning on graph-structured data has been extensively explored in recent work [67]. A refined optimization algorithm is introduced in You et al. [66]. For spatiotemporal graph learning, Qu et al. [50] formulate a contrastive self-supervision method to predict fine-grained urban flows considering all spatial and temporal contrastive patterns. Zhang et al. [71] build a heterogeneous graph neural architecture to capture the multi-view region dependencies concerning POI semantics, mobility flow patterns, and geographical distances.

C.3 Explainability in graph neural networks

Yuan et al. [69] provide unified and taxonomic explanations regarding the importance of a node/an edge/a subgraph in a graph neural network, etc. In our leave-one-out analysis, because we are interested in which categories of information (graph structural vs. weather vs. traffic volume) are most useful, our analysis can be viewed as a first-order explanation of the importance of graph structural features. Further applying the methods of Yuan et al. [69] to explain our findings is a research question for future studies.

C.4 Multitask and transfer learning

Our methods for conducting multitask learning are based on recent theoretical developments regarding negative transfer in multitask learning, thus calling for more robust procedures [64]. In particular, Li et al. [36] propose surrogate modeling to approximate multitask predictions. Li et al. [35] introduce a boosting procedure as an ensemble method for multitask learning on graph-structured data. Our methods for transfer learning are based on recent developments for robust fine-tuning [37, 33]. Ju et al. [32] develop a spectrally-normalized generalization bound for graph neural networks and design a noise-stability optimization algorithm for improved fine-tuning.

C.5 Development of our dataset

Currently, we have the road network features and weather information for all the states in the US. The bottleneck is the traffic volume and the accidents. For the eight states in our current dataset, both types of data are published by the Department of Transportation on the respective state's website. See the links to each state's government website in Table 5.

For the other states, we have checked their Department of Transportation websites, and we could not find detailed data, including accidents and traffic volume (like the eight states we currently have). Once the data is updated, we would be happy to update our dataset as well.

For a few states, for example, California and New York, the traffic volume data and accident information are both available for a few counties through their transportation departments, such as Los Angeles and New York City. For New York City, we have collected 2.02 million accident records from 2012 to 2023, including the latitude and longitude of each accident. For California, we have 0.4 million Motor Vehicle Crashes from 2016 to 2021. However, these do not have the latitude and longitude information, so we cannot match a record to a particular edge/node of the network.