Minya University Faculty of Engineering CSE Dept.

Trajectory-based Intersection Detection and Classification

A project submitted to the department of Computers and Systems Engineering as a partial fulfillment for a B.Sc. degree

Submitted by

Abanoub Kased

Rana Rabee

Akram Fahmy

Hussein Mohammed

Marco Yacoub

Supervisors

Dr. Abdullah Hassan

Dr. Mohammed Elhenawy

Dr. Huthaifa Ashqar

“Education is not about thinning the herd. Education is about helping every student succeed.”

— Andrew Ng

ACKNOWLEDGMENTS

Our sincere gratitude goes out to Drs. Mohammed Elhenawy, Abdullah Hassan and Huthaifa Ashqar for their unwavering support, superb direction, and encouragement during our senior project.

Dr. Mohammed Elhenawy

Senior Research Fellow at Faculty of Health, Queensland University of Technology

Our primary supervisor, Dr. Mohammed Elhenawy, whose idea was to pursue this research, has our sincere gratitude. We are appreciative of his ongoing assistance, advice on scientific matters, and encouragement throughout the project.

Dr. Abdallah A.Hassan

Assistant Professor, Computers and Systems Engineering Department, Minia University

We would like to thank our supervisor, Dr. Abdullah Hassan, for providing this opportunity. We view it as a great honor to have studied under such a skilled instructor.

Dr. Huthaifa Ashqar

Assistant Professor, Arab American University

We would like to thank our supervisor, Dr. Huthaifa Ashqar, for sharing his expertise with us during the entire project. We're very grateful of his assistance.

Abstract

Accurate and up-to-date road network maps are required for connected and automated vehicles to keep up with the world's continually expanding road networks. Intersection maps in C-ITS play a crucial role in enhancing situational awareness for drivers, promoting safer driving behaviors, and facilitating coordinated actions to prevent conflicts and crashes. These maps provide geometric data, including lane configurations, road widths, intersection layouts, and traffic signal locations, which enable vehicles to make informed decisions and lane selection, and C-ITS systems can issue warnings to drivers about potential conflicts, such as red-light violations or stop sign violations. The traditional approaches for identifying road networks, i.e., satellite images and field surveys, require significant time and effort and are labor-intensive. With the wide usage of GPS-embedded devices, a massive amount of trajectory data is generated, which provides a new opportunity to update road maps with less effort. In recent years, deep learning techniques have emerged as powerful tools for automatic map creation from trajectory data. We propose a novel approach that leverages deep learning algorithms to extract meaningful spatial information and construct maps from trajectory data. The proposed method employs a convolutional neural network (CNN) architecture to learn spatial patterns and features from the trajectory data. An intersection classifier and intersection detector are built. An intersection detector is built to localize the intersection location from vehicle trajectories based on different object detection architectures. Intersection classifier built to identify intersection types from vehicle trajectories, this intersection classifier is based on five different convolutional deep neural network architectures. To evaluate the effectiveness of the proposed approach, extensive experiments are conducted using real-world trajectory datasets. The results demonstrate that the deep learning-based map creation method outperforms traditional techniques in terms of accuracy, efficiency, and generalizability. Furthermore, the proposed approach offers flexibility and scalability, as it can be applied to various types of trajectory data, including GPS traces, mobile phone data, and vehicle movement data. It enables generating maps at different

scales, ranging from localized urban maps to regional or global maps, depending on car trajectories data.

In conclusion, we present a novel deep learning-based approach for creating maps from trajectory data. The method harnesses the power of convolutional to effectively extract spatial and temporal information, leading to accurate and comprehensive map representations. The proposed approach has the potential to revolutionize the field of map generation, providing valuable insights for decision-making processes in transportation, urban planning, and other domains reliant on spatial data analysis.

Table of Contents

List of Figures

List of tables

Chapter 1: Introduction and Related Work

1. introduction

Intersection classification is essential for map generation as the rules for creating intersection maps depend on the type of intersection. By classifying intersections according to their geometry, it is possible to generate road networks that include appropriate junction types in the right locations. There are several benefits of using intersection classification for map generation [1-4]. First, improved traffic flow management; different types of intersections can have different traffic characteristics and requirements, such as traffic lights, roundabouts, or stop signs. By accurately reflecting these patterns in a generated map, it becomes possible to plan and manage traffic flow more effectively. By using intersection classification to generate maps with appropriate junction types, it becomes possible to prioritize safety in urban planning and road design. Third, accurately modeling real-world road networks requires a nuanced understanding of the types of intersections that are common in different regions and urban environments. By using intersection classification to generate maps, it becomes possible to create more realistic representations of real-world cities and towns. Fourth, by generating road networks with appropriate intersection types, it becomes possible to reduce congestion, improve traffic flow, and optimize transportation networks.

Intersections are a pivotal part of road networks, as most traffic fatalities take place at or close to intersections. The Federal Highway Administration reported 10,180 traffic fatalities involving an intersection in 2019 [5], so managing and updating them is the key to keeping the traffic in urban areas under control. Road intersections are places where two or more roads cross each other, and they are often the locations of crashes and collisions between vehicles, pedestrians, and cyclists.

Existing methods of intersection map inference/generation can be categorized into two categories based on the data sources used: (1) methods relying on image-based sources like remote sensing images (RS images), and aerial images[6-8], (2) methods relying on trajectories [9-14]

RS and aerial images are quickly advancing and provide much more information compared to trajectories. However, they are susceptible to errors caused by natural factors like rain and

clouds or by the city's topology, like trees or tall buildings, so it faces different challenges than trajectory data. With the rise of GPS-embedded devices, large quantities of trajectory data are being provided and updated daily, making using trajectory data a good choice for map generation. But with large amounts of data comes a price, which is the degraded quality of the data and the amount of noise in the data. Besides, most GPS devices have a low sampling rate which makes it difficult as the car might have passed multiple intersections before an update has taken place. Therefore, the classification of intersections needs a large set of observed trajectory data points to identify the intersection type. Consequently, the specific map inference model is chosen to estimate the intersection characteristics such as the center point, the stop lines, and the number of lanes.
Traditional map inference involves collecting data through surveys or remote sensing techniques, such as satellite imagery or light detection and imaging (LiDAR). While these methods can provide accurate information, they are often expensive and time-consuming. For example, a detailed topographical survey of an area could take weeks or even months to complete, and the cost of equipment and personnel can be significant. Crowdsourced data, on the other hand, offers a potential solution to this problem. By leveraging the vast amounts of data generated by mobile devices and other sources, researchers can potentially create and update maps more quickly and at a lower cost. One type of crowdsourced data that can be used for map inference is trajectory data.
Trajectories refer to the paths that individuals or objects take as they move through space. This data can be collected through GPS-enabled devices such as smartphones, fitness trackers, or vehicles equipped with tracking systems, including automated and connected vehicles (CAVs). By analyzing trajectories, researchers can gain insights into patterns of movement and activity within a particular area. This information can then be used to infer the locations of roads, buildings, and other features on a map. For example, if a large number of trajectories converge on a particular point, it may indicate the presence of a popular landmark or attraction. However, there are challenges to using crowdsourced data for map inference. For example, the quality and accuracy of the data can vary significantly, and there may be issues with privacy and data ownership. Additionally, crowdsourced data may not be available or limited to minor streets where traffic is light, leading to biases in the resulting maps. To overcome these challenges, more research is needed on how to effectively collect, process, and analyze crowdsourced data for map inference. This includes developing new algorithms and tools for data analysis, as well as addressing ethical and legal considerations related to data ownership and privacy. Ultimately, the use of crowdsourced data has the potential to revolutionize the way maps are

generated and updated, making them more accurate, timely, and accessible to a wider range of users

2. Related Work

Intersection classification is used to classify the types of intersections or road junctions that appear in a road network. The classification system is typically based on the number of ingress and egress approaches at each intersection, and the angles between them. The purpose of intersection classification is to identify the intersection type and consequently select the suitable map inference model to generate a realistic intersection map that accurately reflects the patterns and behaviors of real-world traffic flows. Previous research work focused only on the intersection map inference and assumed that the type of the intersection is known (i.e., four legs intersection, roundabout, etc.). This section briefly discusses the different categories of methods found in the literature that have been employed to generate maps at intersections.

A. Methods relying on image-based sources

As this is considered a map inference problem, Convolutional Neural Networks (CNNs) are often the tool used since they have proved their accuracy and reliability in this task, especially Deep Convolutional Neural Networks (DCNNs) [2]. The big number of layers allows room for more accuracy. Thus, several research efforts proposed the use of different DCNN architectures for the task of map generation with the increasing popularity of RS images and aerial images. He et al. [3] proposed an architecture called ResNet, with several variations discussed in their paper such as ResNet-18, ResNet-34, and ResNet-50, with the numbers corresponding to the number of layers. It was tested on the CIFAR-10 [4] dataset which is a dataset of colored images with ten labeled classes. Ronneberger et al. [1] proposed an architecture called U-Net which was used in image segmentation in the Biomedical field. The architecture consisted of a contracting path and an expansive path. The contracting path has the architecture of typical CNNs (two 3x3 convolutions followed by a ReLU), while the expansive path consists of an upsampling of the feature map followed by a 2x2 convolution that halves the number of feature channels. Zhang et al. [15] adapted both ResNet and U-Net architectures and proposed ResUNet to be optimal for the task of road extraction using aerial images of roads in Massachusetts [6]. ResUNet uses residual units as the basic building blocks and removed the cropping operation, it also has 15 Convolutional layers compared to 23 layers of U-Net and performs better in the task of road extraction (0.92 breakeven points with ResUNet compared to .905). LinkNet was proposed by Chaurasia et al. [16], it uses ResNet-18 as its pre-trained encoder, and it was used in semantic segmentation. Zhou et al. [7] used LinkNet architecture but with a different pre-trained encoder ResNet-34 and named it D-LinkNet, it was used on aerial images for road extraction and it achieved better performance than LinkNet for this task. The previous methods relied on only one source of data whether it was RS images or satellite images, however, both sources have their flaws as discussed earlier. Li et al. [8] proposed to combine trajectory data with RS images to get the best results and to overcome the challenges of using either of these sources alone. The U-Net architecture was utilized, and it showed far better results than RS images alone.

B. Methods relying on trajectories

The methods used in literature that fall under this category can be classified into three categories: trace merging, clustering, and rasterization. Trace merging is the process of adding new traces and merging them with the old traces like in Cao et al [9], and Niehöfer et al [10]. The method proposed by Cao is considered the standard trace merging method. Afterward, Niehöfer improved this method by removing or adjusting the existing road network after merging a new trace using its weights. Clustering is a method of grouping data points into similar groups, there are several methods of clustering trajectories like Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [11], K-means [12], Kernel Density Estimation (KDE) [13]. Edelkamp et al. [12] proposed the first K-means algorithm for generating road networks. Clustering and Trace merging methods do not perform well on noisy data [14]. Rasterization is the process of converting an image into raster data. In Guo et al. [17], rasterization with an Otsu algorithm was used. The rasterization method performs the best in comparison to the other two methods in the face of noise as it treats each trace individually [14].

C. Methods using extracted features

The two previous categories of methods only rely on the raw data and then try to make it more efficient or less noisy. Still, some researchers used trajectory data to extract input features, including speed, acceleration, heading, or density. Nguyen et al. proposed extracting Histogram-based features using GPS Trajectories [18]. The mapped space containing the trajectory was divided into equal-sized cell grids. Therefore, the GPS trajectory can be encoded into a matrix; its elements at location (i, j) are the number of GPS Points in the grid cell (i, j). Zhang et al. proposed converting the trajectories into images by counting the trajectory points inside each cell [19]. However, they suggested resampling

the trajectory at even intervals to avoid short intervals that lead to a long stay (i.e., many points inside the same cell). Elhenawy et al. proposed extracting features using Recurrence Quantification Analysis (RQA), which provided extensive temporal behavior of the obtained signal (e.g. speed and acceleration) [20]. Consequently, the extracted features using RQA were used to create images (called RQA images) that could then be used as inputs in a classification algorithm instead of using many numerical features. Daley et al. segmented 10 Hz trajectory data into a 1-second window that was transformed into a 3-channel image. The three channels represented the Gramian Angular Summation/Difference Field (GASF/GADF) and the Markov and Transition Field (MTF) [21]. Guo et al. [17] proposed using a grid of equal-sized cells and a Gaussian kernel to calculate the contribution of each GPS point in the 2D plan to the intensity of the cell. Li et al. used the grid idea to calculate the density graph/image and the speed graph/image, which are considered static and dynamic features, respectively [8]. Ruan et al. divided the map region of interest into tiles and each tile into a grid with $N \times M$ cells [22]. Then for each cell, two types of features are extracted, namely, the spatial and the transition features. The spatial features are extracted from the trajectory data inside the cell. Spatial features include the density of the GPS points, average speed, number of line segments inside the cell, and the normalized histogram of the regular direction of movement between two consecutive points. On the other hand, the transition features of a grid cell are useful for low-frequency trajectories. The transition features at cell i are two binary matrices; the first represents all the direct origins of trajectories landing in cell i, and the other matrix represents all the direct destinations from cell i. Prabowo et al. [23] divided the area of interest into a 1mx1m grid and used this grid to build a 3-channel image. The first channel is directly constructed such that a cell is assigned a one if it has at least one GPS point and zeroes otherwise. The other two channels need the computation of the speed vectors in the x and y directions by extrapolating the speed and heading of each GPS point in the trajectory. Then, each cell in the second and the third channels will have the aggregated and normalized speed projection in the x and y directions, respectively. Eftelioglu et al. proposed creating consecutive pairwise lines using the GPS points and using these lines to create the image [24].

In this study, the use of transfer learning and pre-trained DCNNs is proposed to classify/recognize the location type at which a subset of trajectory data is observed. Two approaches are investigated to convert the trajectories (i.e., time series) into an image that can be used as an input for the DCNNs. The first approach creates images by scattering the trajectory location information in the 2-D plan. The second approach converts the trajectory data into multi-channel images by dividing the 2-D plan into a grid of equal-area cells and extracting different features at each cell, including density and speed. For the sake of testing the robustness of the idea and getting good classification results, five different DCNNs (NasNet, ResNet, MobileNet, EfficientNet, and DenseNet) are trained and tested on three datasets from the following cities: Porto in Portugal, San Francisco in The United States, and Beijing in China.

In the first set of experiments (i.e., the baseline), simulated data is created and used in the first training round. Consequently, real data is used to fine-tune the models in the second training round. Then the trained models are tested using the real data only. In the second set of experiments, features are extracted from the dataset using two grid sizes 20x20 and 30x30, and only real data is used for training and testing the models. The extracted features are density, speed, acceleration, and heading. Moreover, different models are trained using different subsets of features, and their performances are compared. Finally, a bootstrap-based heuristic is used to construct a good ensemble using a subset of the trained models.

Chapter 2: Deep neural networks

1. Classification

Classification is the process of categorizing data into predefined classes or categories based on their features or attributes. It involves training a machine learning model to learn patterns and relationships in labeled data, enabling it to make predictions about the class or category of new, unseen data points. The model analyzes the features of the data points and maps them to the corresponding classes, allowing for effective classification of future instances. This process is widely used in various domains to automate the categorization and prediction tasks, enabling efficient decision-making and analysis of complex datasets.

Using classification enables:

- Categorization: Organizing and making sense of complex datasets by grouping data into different classes or categories based on their features or attributes.

- Pattern recognition: Learning patterns and relationships in the data, allowing the identification of similarities and differences among instances for more accurate decision-making.

- Prediction: Making predictions about the class or category of new, unseen instances based on trained classification models.

- Automation: Automating the process of assigning labels or categories to data points, saves time and effort, particularly with large datasets.

- Insight generation: Gaining insights into the characteristics and distributions of different classes within the data, identifying trends, outliers, and unique patterns.

- Efficient data organization: Structuring and organizing data based on similarities, improving data management and retrieval processes.

- Decision support: Using classification models as decision support tools, providing recommendations or predictions based on learned patterns for various applications.

- Customization and adaptation: Training classification models on specific datasets and tailoring them to project requirements for improved accuracy and relevance.

Classifiers in deep learning are commonly used for computer vision tasks like image classification and object detection. They leverage deep learning models, such as convolutional neural networks (CNNs), to achieve state-of-the-art performance in tasks like image recognition, object localization, and semantic segmentation. These classifiers learn from labeled datasets to recognize patterns, features, and characteristics in images, enabling accurate classification and object detection.

2. The five CNN

- **MobileNet-V3-Large**
- **EfficientNet-B0**
- **NASNetMobile**
- **ResNet-50**
- **DenseNet-121**

1. **NASNetMobile:** stands for Neural Architecture Search Network-Mobile. It is a convolutional neural network architecture that has been developed using neural architecture search techniques. It has approximately 4.5 million parameters, which determines the capacity and

complexity of the model. NASNetMobile is designed to be computationally efficient and suitable for mobile and embedded devices.

2. **ResNet-50:** is a widely used deep convolutional neural network architecture. It consists of 50 layers and is known for its residual connections, which enable the network to effectively train very deep models. ResNet-50 has approximately 23.5 million parameters, making it a larger and more powerful model compared to the other classifiers mentioned. It has been successfully used in various computer vision tasks, including image classification and object detection.

3. **MobileNet-V3-Large:** is an efficient convolutional neural network architecture that is designed to strike a balance between accuracy and computational efficiency. It has approximately 4.2 million parameters, making it lightweight and suitable for resource-constrained environments. MobileNet-V3-Large incorporates several techniques such as depthwise separable convolutions and linear bottlenecks to reduce the computational cost while maintaining good performance.

4. **EfficientNet-B0:** is part of a family of convolutional neural network architectures called EfficientNets. It is known for its efficiency in terms of both model size and computational resources. EfficientNet-B0 has approximately 4 million parameters and is designed to achieve good performance while being lightweight. The EfficientNet family uses a compound scaling method to balance the model's depth, width, and resolution, resulting in efficient and effective models.

5. **DenseNet-121:** is a compact convolutional neural network architecture that emphasizes densely connected layers. It has approximately 29 million parameters and is known for its ability to capture rich feature representations by connecting each layer to every other layer in a feed-forward manner.

DenseNet-121 is designed to encourage feature reuse and gradient flow throughout the network, enabling better information flow and improved performance.

These classifiers have been widely used in computer vision tasks, including image classification, object detection, and feature extraction. They offer different trade-offs between model complexity, computational efficiency, and accuracy, allowing researchers and practitioners to choose the one that best suits their requirements and constraints.

Table I Features of the used models

Model	Label	Number of Parameters	Number of Layers
MobileNet-V3-Large	MN	4.2 M	269
EfficientNet-B0	EN	4 M	238
NASNetMobile	NN	4.2 M	769
ResNet-50	RN	23.5 M	175
DenseNet-121	DN	29 M	427

3. Transfer learning

Transfer learning is a powerful technique in the field of deep learning that has revolutionized the way we approach complex tasks. It addresses the challenge of limited labeled data by leveraging knowledge gained from pre-training on a large-scale dataset to improve performance on a target task. The underlying concept behind transfer learning is that a model trained on a large and diverse dataset can learn general features and representations that apply to a wide range of tasks.

The transfer learning process typically involves two main stages: pre-training and fine-tuning. In the pre-training stage, a deep learning model, such as a Convolutional

Neural Network (CNN) or a Transformer, is trained on a large dataset, often with millions of labeled examples. This pre-training step allows the model to learn generic features and representations that capture high-level patterns in the data. These learned representations can be thought of as a knowledge base that the model can draw upon when tackling new tasks.

Once the pre-training stage is complete, the model is fine-tuned on a smaller task-specific dataset. Fine-tuning involves initializing the model with the learned weights from pre-training and updating them using the target dataset. By fine-tuning, the model adapts its learned representations to the specific characteristics of the target task, allowing it to make more accurate predictions.

Transfer learning offers several key benefits. Firstly, it reduces the need for large amounts of labeled data. By leveraging the knowledge learned in the pre-training phase, the model can generalize well even with limited labeled examples, making it particularly useful in scenarios where obtaining large labeled datasets is challenging or expensive.

Furthermore, transfer learning improves the generalization capability of the model. The pre-trained models have learned features and representations from diverse datasets, enabling them to capture a wide range of patterns and variations. By leveraging this knowledge, transfer learning helps the model handle variations in lighting, viewpoints, object scales, and other factors, leading to improved performance across different tasks.

Transfer learning also speeds up the training process. Since the pre-trained model has already learned low-level features, subsequent layers can focus on learning more task-specific features during fine-tuning. This initialization helps the model converge faster, reducing the overall training time required for the target task.

There are different strategies for transfer learning. Full transfer learning involves using the entire pre-trained model for fine-tuning the target task. All layers of the model are updated during the training process. Partial transfer learning, on the other hand, involves freezing some layers while allowing others to be fine-tuned. Typically, lower layers responsible for low-level features are frozen, while higher layers capturing more task-specific features are fine-tuned. Another strategy is feature extraction, where the pre-trained model is used as a fixed feature extractor, and only the final classification layers are trained on the target dataset.

Transfer learning finds applications in various domains. In image classification, pre-trained CNN models, such as VGG, ResNet, or EfficientNet, have been successfully applied to different tasks, including identifying objects, recognizing facial expressions, or detecting diseases in medical images. In natural language processing, models like BERT or GPT have been fine-tuned on specific tasks, such as sentiment analysis or named entity recognition. Transfer learning has also shown promising results in computer vision tasks like object detection and semantic segmentation.

In conclusion, transfer learning is a powerful technique that allows us to leverage pre-trained models and their learned representations to tackle new tasks with limited labeled data. By transferring knowledge from the pre-training phase, models can benefit from generalized features, improved generalization, and faster convergence. Transfer learning has been instrumental in advancing deep learning applications and continues to drive progress in various domains, making it an indispensable tool in the deep learning toolbox.

Why Use Transfer Learning?

- dataset is limited
- Increase performance and accuracy
- Reduce time of training

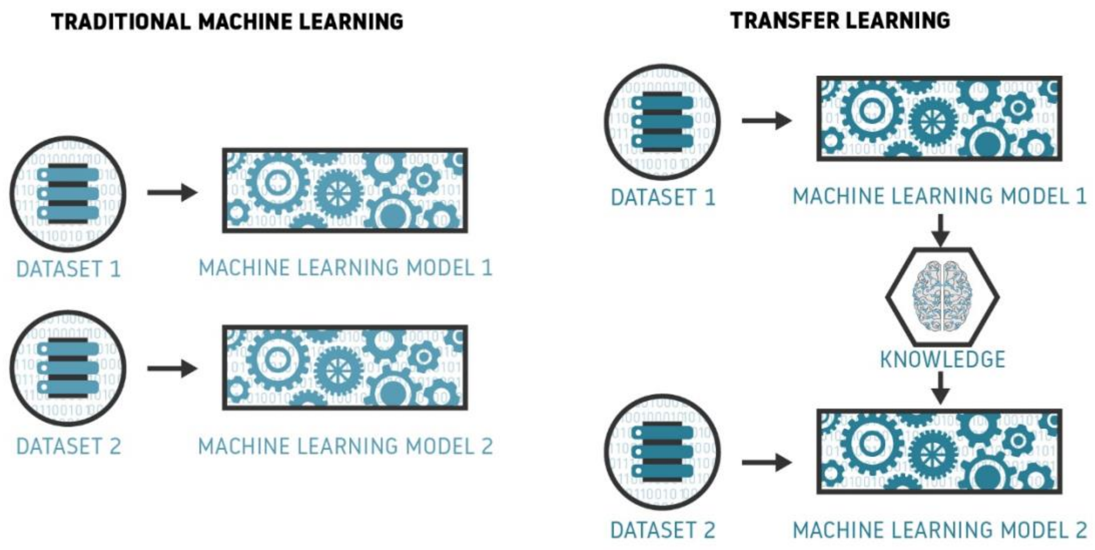


Figure 1 Traditional machine learning vs transfer learning

4. Fine-tuning

Fine-tuning is an indispensable and intricate process in the field of machine learning that plays a pivotal role in enhancing the performance, adaptability, and robustness of pre-trained models. This highly iterative and meticulous technique involves adjusting the various parameters, weights, and biases of a pre-existing model by subjecting it to task-specific or domain-specific data. By fine-tuning, the model can effectively leverage and build upon the knowledge it has acquired from a large-scale, general dataset while simultaneously tailoring its predictions and representations to meet the specific requirements of a given task or dataset.

One of the primary advantages of fine-tuning is its ability to handle scenarios where limited or specialized data is available. By exposing the model to a smaller, task-specific dataset, it can extract valuable patterns, nuanced information, and unique characteristics that are crucial for accurate predictions in the target domain. Through the fine-tuning process, the model gains the capacity to discern task-specific nuances, generalize better, and capture the intricacies and complexities of the data it is trained on.

Fine-tuning strikes a delicate balance between the knowledge captured by the pre-trained model and its ability to adapt and specialize for the target task. It allows the model to refine its representations, update its internal weights, and fine-tune its decision boundaries to align with the task-specific data distribution. By doing so, the model can make more accurate predictions, achieve higher performance, and produce superior quality results in the desired application.

Furthermore, fine-tuning empowers machine learning practitioners to address various challenges such as transfer learning, domain adaptation, and overcoming data scarcity. It serves as a powerful tool in fields such as natural language processing, computer vision, and reinforcement learning, enabling the application of pre-trained models to a wide range of real-world problems.

Chapter 3: Datasets and processing

1. Importance of datasets

Datasets play a critical role in deep learning, serving as the foundation for model training, evaluation, and ultimately, the success of a deep learning project. The quality, size, and diversity of datasets directly impact the performance and generalization capabilities of deep learning models. In the project mentioned, three datasets were utilized: the TDRIVE dataset, the Porto taxi trajectories dataset, and the San Francisco dataset. Let's explore the importance of datasets in deep learning in general.

1. Training Data: Deep learning models learn from large amounts of labeled data to extract meaningful features and patterns. The availability of high-quality training data is crucial for model performance. Datasets with a significant number of diverse samples allow models to learn robust representations and generalize well to unseen examples. In the project, the T-DRIVE, Porto taxi trajectories, and San Francisco datasets provide the necessary training data to train the deep learning models effectively.

2. Representation Learning: Deep learning models excel at automatically learning representations from raw data. Datasets that encompass a wide range of variations and scenarios enable models to capture the inherent complexity of real-world problems. By training on diverse datasets, models can learn features that are more generalizable and transferable, leading to better performance in different contexts.

3. Generalization: The ability of a deep learning model to generalize well to unseen data is a crucial factor in its effectiveness. Datasets that cover a broad range of scenarios, including different environments, conditions, and variations, help models learn robust representations

that are not overly specific to the training data. This allows the models to make accurate predictions on new, unseen examples. By utilizing multiple datasets, such as the T-DRIVE, Porto taxi trajectories, and San Francisco datasets, the project aims to improve the generalization capabilities of the deep learning models for classifying intersections.

4. Evaluation and Benchmarking: Datasets also play a vital role in evaluating and benchmarking deep learning models. By using standardized datasets, researchers and practitioners can compare the performance of different models on the same data, enabling fair and objective assessments. The chosen datasets in the project provide a means to evaluate the classification accuracy and effectiveness of the trained models in the context of road intersection analysis.

5. Real-World Relevance: The relevance of the dataset to the target problem domain is crucial for achieving practical and impactful results. Using datasets that represent real-world scenarios, such as the T-DRIVE dataset, Porto taxi trajectories dataset, and San Francisco dataset, ensures that the trained models capture the intricacies and challenges of the task at hand. This enhances the applicability and usefulness of the deep learning models in real-world road network analysis and traffic management.

In summary, datasets form the backbone of deep learning projects. They provide the necessary training examples, enable representation learning, improve generalization, facilitate evaluation and benchmarking, and ensure real-world relevance. The use of diverse, high-quality datasets, such as the T-DRIVE, Porto taxi trajectories, and San Francisco datasets in the mentioned project, empowers deep learning models to learn meaningful representations and accurately classify intersections based on trajectory data, contributing to advancements in transportation analysis and urban planning.

A. T-drive dataset

the trajectories data is collected from 10,357 taxis from Feb. 2 to Feb. 8, 2008, in Beijing. The total number of points in this dataset is about 15 million and the total distance of the trajectories reaches 9 million kilometers. The sampling rate average is about 177 seconds within 623 meters distance.

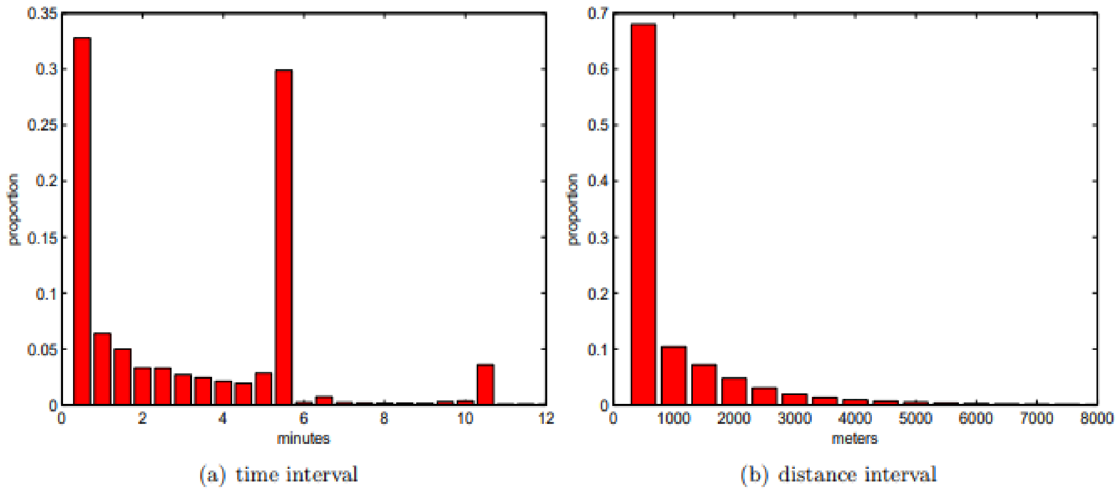
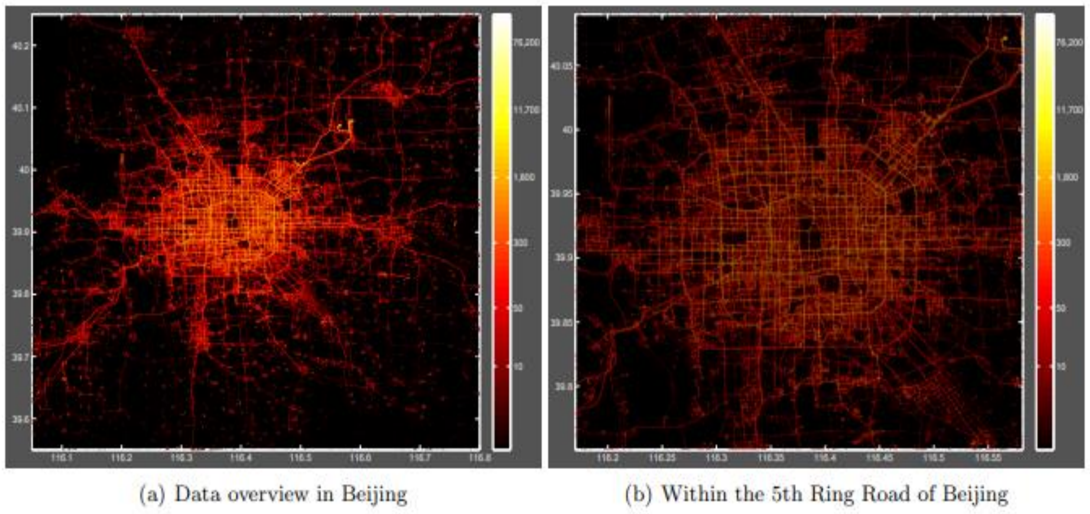


Figure 2 Histograms of time interval and distance between two consecutive points.



(a) Data overview in Beijing
(b) Within the 5th Ring Road of Beijing

Figure 3 Distribution of GPS points, where the color indicates the density of the points.

The T-Drive dataset, available on Kaggle, provides a detailed collection of information related to ride-hailing trips. It encompasses a wide range of variables that offer a comprehensive view of the ride-hailing industry and the behavior of drivers within this context.

The dataset likely includes key attributes such as trip duration, pick-up and drop-off locations, timestamps, and driver ratings. These variables allow for an analysis of trip patterns, time-based trends, and overall ride durations. By examining the pick-up and drop-off locations, one could potentially uncover insights about popular ride origins and destinations, helping to identify areas with high demand for ride-hailing services.

Moreover, the dataset may contain data on driver characteristics and behaviors. This could include information such as driver ratings, trip frequency, and possibly driver demographics. By examining these factors, researchers and analysts can explore the relationships between driver performance, customer satisfaction, and other variables that impact the overall ride-hailing experience.

Analyzing the T-Drive dataset can lead to several potential applications and research avenues. For instance, it could be used to develop predictive models that estimate trip durations or driver ratings based on specific variables. This would be valuable for both ride-hailing companies and customers who rely on accurate time estimates and quality service. Additionally, the dataset could be leveraged to optimize dispatch algorithms, enhance driver training programs, or identify areas for service expansion.

Furthermore, researchers interested in the ride-hailing industry could utilize the T-Drive dataset to conduct in-depth analyses. They could explore topics such as driver behavior patterns, the impact of external factors like weather on trip durations, or the effectiveness of surge pricing strategies. By examining such aspects, researchers can contribute to the development of data-driven insights and strategies that can enhance the overall efficiency and effectiveness of ride-hailing services.

Overall, the T-Drive dataset presents a valuable resource for studying and understanding various aspects of the ride-hailing industry. Its rich collection of variables enables researchers, analysts, and data enthusiasts to gain insights into driver behavior, trip characteristics, and the factors that contribute to a successful ride-hailing experience.

The T-Drive dataset available on Kaggle is a comprehensive collection of information related to ride-hailing trips. It provides detailed data on various aspects of the ride-hailing industry, focusing on driver behavior and trip characteristics.

The dataset includes a range of variables that offer insights into the dynamics of ride-hailing services.

Some of the key features of the dataset:

1. Trip details: This includes information such as trip duration, distance traveled, and the start and end timestamps of each trip. These variables allow for the analysis of trip patterns, duration distribution, and time-based trends.

2. Location information: The dataset likely contains data on the pick-up and drop-off locations for each trip. This enables the examination of popular ride origins and destinations, helping to identify areas with high demand for ride-hailing services.

3. Driver information: The dataset may include details about the drivers, such as their unique identifiers, ratings, and possibly demographic information. These variables provide insights into driver performance, customer satisfaction, and the factors that influence ratings.

4. Additional attributes: Depending on the dataset, there may be additional variables available, such as weather conditions during the trips, fare information, and customer feedback. These attributes can contribute to a more comprehensive analysis of the ride-hailing ecosystem.

The T-Drive dataset can be leveraged for various purposes. Researchers and data analysts can explore the dataset to uncover patterns, trends, and correlations within the ride-hailing industry. They can investigate the factors that impact trip durations, identify peak hours or busy locations, and assess the relationship between driver ratings and other variables.

Moreover, the dataset provides an opportunity to develop predictive models. Researchers can build algorithms to estimate trip durations, predict customer ratings based on trip characteristics, or optimize driver dispatch strategies for improved efficiency.

Additionally, the T-Drive dataset offers insights that can be beneficial for ride-hailing companies. They can use the dataset to evaluate driver performance, optimize pricing strategies, and enhance the overall customer experience.

By studying the T-Drive dataset, researchers, analysts, and industry professionals can gain a deeper understanding of the ride-hailing industry and make data-driven decisions to improve various aspects of the service.

B. ECML PKDD 15 dataset

The dataset available at the provided link corresponds to the "Porto Taxi Trajectories" dataset. It is hosted on Figshare, a reputable platform for sharing research data. This dataset offers a comprehensive collection of taxi trajectory data from the city of Porto, Portugal.

The Porto Taxi Trajectories dataset is a valuable resource for studying urban mobility patterns and analyzing the dynamics of taxi services within the city. It provides detailed information about taxi trips, including pick-up and drop-off locations, timestamps, and other relevant attributes. With this dataset, researchers and analysts can gain insights into travel patterns, traffic flows, and demand variations in different parts of the city.

By examining the trajectories of taxi trips, it becomes possible to analyze travel times, route choices, and congestion hotspots. This information can be used to identify traffic bottlenecks, optimize transportation infrastructure, and design effective urban planning strategies. Additionally, the dataset may contain information about the taxi drivers, such as their unique identifiers or other driver-related attributes, allowing for the analysis of driver behavior and performance.

The Porto Taxi Trajectories dataset has the potential to contribute to a wide range of research areas. It can be used to develop predictive models for travel time estimation, evaluate the effectiveness of transportation policies, or explore the impact of various factors, such as weather conditions or events, on taxi demand and mobility patterns. Furthermore, the dataset can serve as a benchmark for developing and testing algorithms and techniques related to transportation and urban mobility.

Overall, the Porto Taxi Trajectories dataset provides a valuable resource for researchers, urban planners, and transportation analysts interested in studying taxi services, traffic patterns, and urban mobility in the city of Porto. Its detailed and extensive nature allows for in-depth analysis and the generation of insights that can inform decision-making and contribute to the development of smarter, more efficient transportation systems.

The Porto Taxi Trajectories dataset, available on Figshare, offers a comprehensive collection of taxi trajectory data from the city of Porto, Portugal. It provides detailed information about taxi trips, including various attributes related to pick-up and drop-off locations, timestamps, and other relevant factors.

The dataset encompasses a vast amount of data that enables researchers and analysts to study and analyze urban mobility patterns within Porto. By examining the trajectories of taxi trips, it becomes possible to gain insights into travel times, route choices, and traffic flow dynamics. This information can be invaluable for understanding traffic patterns, identifying congestion-prone areas, and optimizing transportation infrastructure and urban planning.

In addition to trip-related details, the dataset may also include information about the taxi drivers, such as unique identifiers or other driver-specific attributes. This allows for the exploration of driver behavior, performance, and factors that influence their choices during trips.

The Porto Taxi Trajectories dataset offers numerous opportunities for research and analysis. Researchers can develop predictive models to estimate travel times and assess the impact of various factors, such as weather conditions or special events, on taxi demand and mobility patterns. It can also serve as a benchmark for evaluating the effectiveness of transportation policies and testing new algorithms and techniques in the field of transportation and urban mobility.

By leveraging this dataset, researchers, urban planners, and transportation analysts can gain a deeper understanding of the taxi services and traffic dynamics within Porto. The dataset's rich and extensive nature provides a valuable resource for generating insights and informing evidence-based decision-making processes to improve transportation systems and enhance urban mobility in the city.

C. San Francisco dataset

The last dataset utilized in the project is the San Francisco dataset. This dataset, available on Kaggle, offers valuable insights into the road network and transportation patterns in the city of San Francisco. By incorporating this dataset into the project, deep learning models can learn from the specific characteristics and complexities of the San-Francisco Road system, enhancing their ability to classify intersections accurately. The San Francisco dataset provides a real-world context and relevance to the project, enabling the trained models to capture the unique challenges and dynamics of urban traffic in this vibrant city.

2. Converting Trajectory to Images

In this project, we harnessed the power of OpenStreetMap (OSM), a crowd-sourced mapping platform, to facilitate the identification of road intersections, extraction of trajectories within those intersections, and acquisition of visual images depicting the real-world scenes within each intersection. By leveraging the rich and comprehensive mapping data offered by OSM, we were able to obtain valuable insights into the movements, patterns, and characteristics of vehicles or objects within specific intersections.

OpenStreetMap served as an invaluable resource for this work, providing a detailed and up-to-date mapping database that captures intricate information about road networks, intersections, and geographic features. Leveraging this platform, we navigated through the vast mapping data to identify and isolate the intersections of interest, forming the foundation for subsequent analysis.

Once the intersections were identified, we employed the OSM data to extract trajectory information associated with each intersection. These trajectories provided a wealth of data on the movements and paths followed by vehicles or objects within the identified intersections. This data allowed us to gain a deeper understanding of the dynamics, traffic flow patterns, and behavior within each intersection.

In addition to trajectory extraction, we utilized the OpenStreetMap website to obtain visual images of the identified intersections. By accessing the street-level imagery available on OSM, we captured a realistic and comprehensive visual representation of the physical environment within each intersection. These images showcased the road layouts, traffic signs, buildings, and other relevant elements that contribute to the visual context and characteristics of the intersection.

The combination of trajectory data extracted from OSM and the visual images obtained from the platform provided a holistic dataset that integrated spatial information and visual cues. This comprehensive dataset became instrumental in training and evaluating deep learning models, enabling them to learn from both the trajectory data and the visual representation of the intersections. The integration of these two data sources enhanced the accuracy, realism, and contextual understanding of the models, enabling them to make informed decisions and accurate classifications based on the combined information.

Importance of Converting Trajectory to images:

The work of utilizing the OpenStreetMap (OSM) website to find intersections, extract trajectories, and obtain images of those intersections holds significant importance in several key aspects. Firstly, the ability to identify and extract trajectories within intersections provides valuable insights into the behavior, patterns, and dynamics of vehicles or objects within these critical road junctions. This information contributes to a deeper understanding of traffic flow, congestion, and potential risks within intersections, enabling effective traffic management and urban planning.

Secondly, the acquisition of visual images from OpenStreetMap enhances the contextual understanding of intersections. These images provide a realistic representation of the physical environment, including road layouts, lane markings, traffic signs, and surrounding infrastructure. By capturing the visual characteristics of intersections, this work enables researchers and practitioners to analyze and interpret the visual cues that influence driver behavior, traffic regulations, and overall intersection functionality.

Moreover, the integration of trajectory data and visual images from OpenStreetMap offers a comprehensive dataset that combines both spatial and visual information. This integrated dataset provides a more complete and nuanced understanding of intersections, allowing for a holistic analysis of the factors that contribute to their classification, efficiency, and safety. By leveraging this combined dataset, the development of accurate and robust deep learning models becomes feasible, which can have practical applications in various domains, including autonomous driving, traffic prediction, and transportation infrastructure optimization.

Furthermore, the use of OpenStreetMap in this work provides a valuable resource for researchers, as it offers a freely available and continuously updated mapping database. This accessibility fosters reproducibility and allows for broader collaborations within the scientific community. Additionally, OpenStreetMap's crowd-sourced nature ensures that the mapping data is constantly improving, incorporating real-time changes and community contributions, thereby enhancing the accuracy and reliability of the extracted trajectories and images.

Overall, the importance of using the OpenStreetMap website to find intersections, extract trajectories, and obtain images lies in its ability to provide valuable insights into the behavior, visual characteristics, and overall dynamics of intersections. This work contributes to enhancing traffic management strategies, optimizing urban planning efforts, and enabling the development of advanced machine learning models for intersection classification and analysis. Ultimately, this work has the potential to drive advancements in transportation systems, road safety, and urban infrastructure design.

3. Methods of Converting trajectory to images

The method employed to execute the work of utilizing the OpenStreetMap (OSM) website for finding intersections, extracting trajectories, and obtaining images involved a systematic and iterative process. The following steps were followed to accomplish this:

1. Data Retrieval: The first step involved accessing the OpenStreetMap website and retrieving the necessary mapping data. OpenStreetMap provides a rich database of road networks, intersections, and geographic features, which was accessed using appropriate APIs or data extraction techniques.

2. Intersection Identification: With the mapping data in hand, the intersections of interest were identified. This was achieved by analyzing the road network topology and identifying locations where multiple road segments converge. Through careful analysis and filtering, the target intersections were isolated, forming the basis for subsequent analysis.

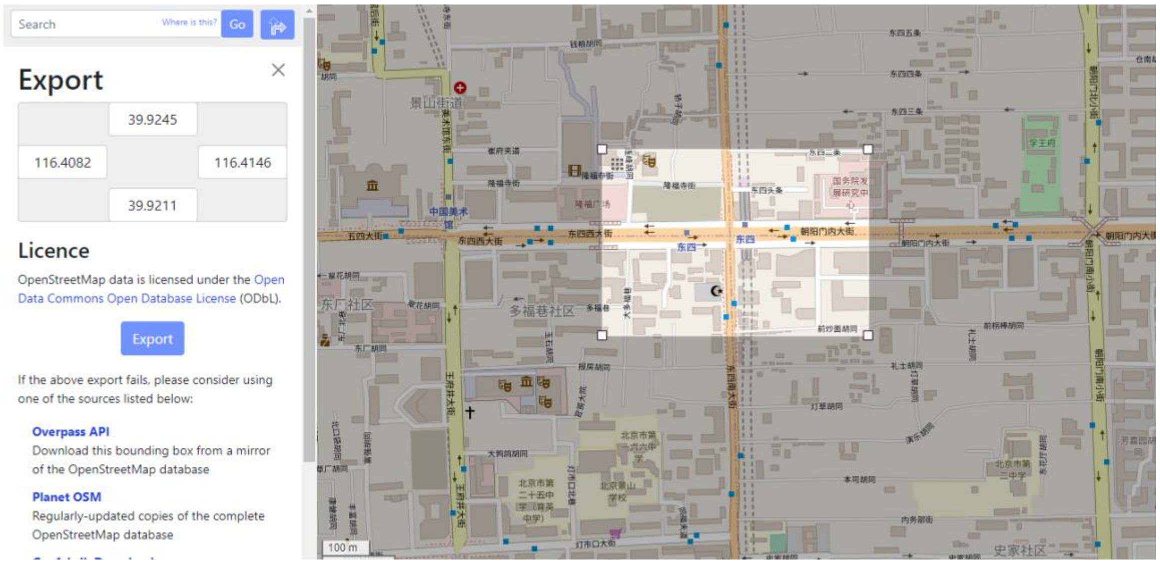
3. Trajectory Extraction: Once the intersections were identified, the next step was to extract trajectories associated with those intersections. This involved parsing and analyzing relevant data sources, such as GPS or vehicle tracking data, to identify trajectory points that intersected with the identified junctions. The extracted trajectory data provided valuable insights into the movements and patterns of vehicles or objects within each intersection.

4. Image Retrieval: In parallel with trajectory extraction, the project leveraged the OpenStreetMap website to obtain visual images of the identified intersections. This was accomplished by accessing the street-level imagery available on OSM or using appropriate APIs to retrieve the required images. The obtained images provided a visual representation of the physical environment within each intersection, capturing road layouts, lane markings, traffic signs, and other relevant features.

5. Integration and Analysis: The extracted trajectories and obtained images were then integrated into a comprehensive dataset. This involved aligning the trajectory data with the corresponding visual images, enabling the correlation of spatial information with visual cues. The integrated dataset facilitated subsequent analysis, such as deep learning model training or classification tasks, where the combined information was utilized to make informed decisions and accurate predictions regarding intersection characteristics.

Throughout the entire process, data preprocessing and cleaning techniques were applied to ensure data quality and consistency. Additionally, appropriate tools and programming languages, such as Python, were employed to handle data extraction, processing, and integration tasks efficiently.

In summary, the execution of the work involved retrieving mapping data from OpenStreetMap, identifying intersections, extracting trajectories, obtaining visual images, and integrating the trajectory data with the corresponding images. This method allowed for a comprehensive analysis of intersections, incorporating both spatial information and visual cues. By following this systematic approach, the project was able to leverage the power of OpenStreetMap and extract valuable insights into intersection behavior and characteristics, enabling further analysis and decision-making in the field of transportation and urban planning.



Export

39.9245

116.4082 116.4146

39.9211

Licence

OpenStreetMap data is licensed under the Open Data Commons Open Database License (ODbL).

If the above export fails, please consider using one of the sources listed below:

Overpass API
Download this bounding box from a mirror of the OpenStreetMap database

Planet OSM
Regularly-updated copies of the complete OpenStreetMap database

Figure 4 Using OpenStreetMap to get BoundBox

Chapter 4: intersections simulation

1. Introduction

The provided code is a comprehensive Python implementation of a simulation framework for traffic intersections, designed to replicate real-world traffic scenarios. It offers an array of functions that enable the creation of diverse road paths and intersections, encompassing a wide range of road sizes, radii, and round intersections.

Within the code, you'll find a collection of functions tailored to generate trajectories between source and destination points in a highly realistic manner. By meticulously calculating the coordinates along these trajectories, the code ensures the fidelity of the simulated paths. Moreover, the code incorporates provisions for introducing randomness and noise into the trajectories, mirroring the inherent variability and unpredictability of actual traffic movements.

In addition to the trajectory generation capabilities, the code includes specialized functions for constructing linear paths, half-circle paths, and quarter-circle paths. This versatility allows for the accurate representation of diverse road configurations encountered at traffic intersections. By seamlessly integrating these path types, the simulation provides a holistic view of the traffic dynamics within and around the simulated intersections.

To facilitate further analysis and visualization, the code also offers functions to generate an origin-destination (OD) matrix. This matrix serves as a valuable representation of the traffic flow distribution among various source and destination points. By leveraging the OD matrix, the simulation dynamically determines the probabilities associated with selecting specific source and destination points for each simulated trip, resulting in a nuanced and data-driven traffic simulation.

For enhanced understanding and interpretation, the code includes comprehensive plotting functionalities. These capabilities enable the generation of visual representations of the simulated trajectories, empowering researchers and traffic analysts to delve deeper into traffic patterns and behaviors. By saving the plots as images, the code facilitates the generation of detailed reports and visual aids for in-depth analysis and presentation.

In summary, this Python code presents a robust and highly customizable simulation framework for the study and analysis of traffic intersections. By simulating realistic traffic scenarios and incorporating advanced features such as trajectory generation, OD matrix calculation, and visualization tools, it provides a powerful platform for evaluating traffic management strategies and algorithms. Researchers and practitioners can leverage this code to gain valuable insights into traffic flow dynamics, inform urban planning initiatives, and devise effective traffic control measures.

2. Flowchart of simulation

The more the simulated examples are close to real road intersection examples, the more variability will be. To create a simulated example, several factors are considered including sampling rate, independent noise for X and Y coordinates achieved by assuming that the data is normally distributed with a different mean and variance for each direction, the density of trips on each road is different than other roads in the same road intersections, and an OD matrix is used with a random number of trips for source-destination. An example of an intersection resulting from the simulation is shown in figure 1. The flow chart for creating the simulated data is presented in figure 2.

The simulation description steps as shown in figure 1:

1) select OD Matrix size based on intersection type (cross intersection, T intersection, roundabout intersection, no intersection)
2) fill OD Matrix with random data
3) select a path between source and destination based on the probability of each source and destination, source and destination can`t be the same as we suppose that there is no return in the same path
4) based on intersection type we create the path between the source and destination
5) we introduce many parameters to simulate the real trajectory
6) Like sampling time, the sampling rate of the data, length of the path, the radius of the round if present, the sampling rate of car trips, noise size in each direction (x, y), mean of the data, standard deviation of the data in each direction (x, y), the distance between two besides the road, number of points represent the roundabout circle.
7) if it`s a roundabout intersection we create a linear path for the source and destination and then we add a roundabout circle and finally, we filter data based on the parameters that we want
8) if it`s a cross intersection we create a linear path for the source and destination and filter data based on the parameters that we want
9) if it`s a T intersection we create a linear path for the source and destination and filter data based on the parameters that we want
10) if it`s no intersection we create a linear path for the source and destination and filter data based on the parameters that we want

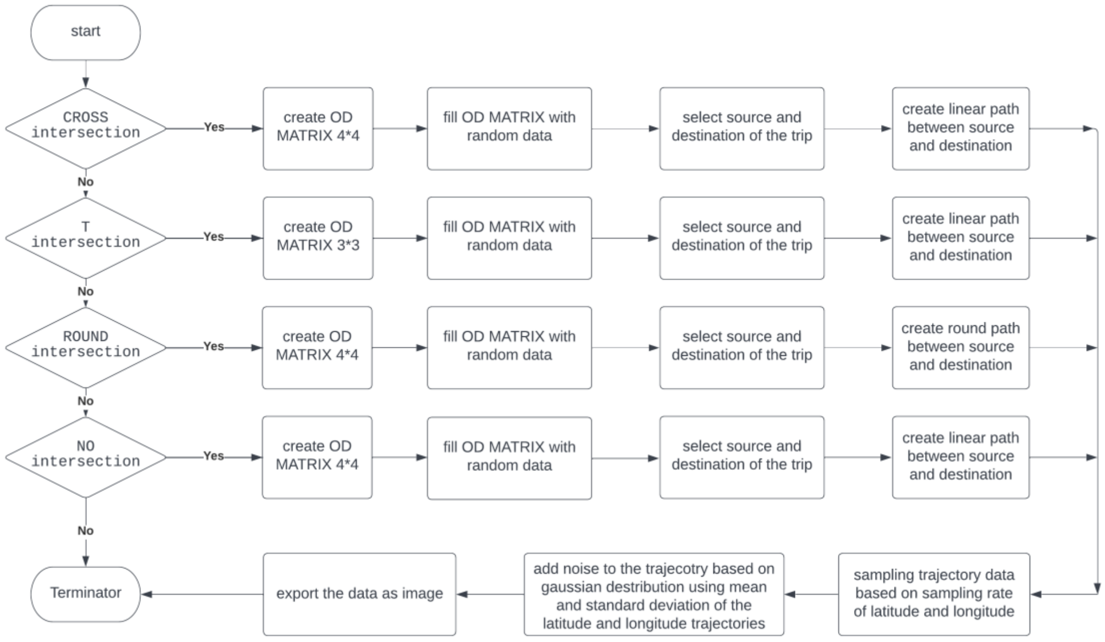

start
CROSS intersection
Yes
No
create OD MATRIX 4*4
fill OD MATRIX with random data
select source and destination of the trip
create linear path between source and destination
T intersection
create OD MATRIX 3*3
ROUND intersection
create round path between source and destination
NO intersection
Terminator
export the data as image
add noise to the trajecotry based on gaussian destribution using mean and standard deviation of the latitude and longitude trajectories
sampling trajectory data based on sampling rate of latitude and longitude

Figure 5 flowchart of intersection simulation.

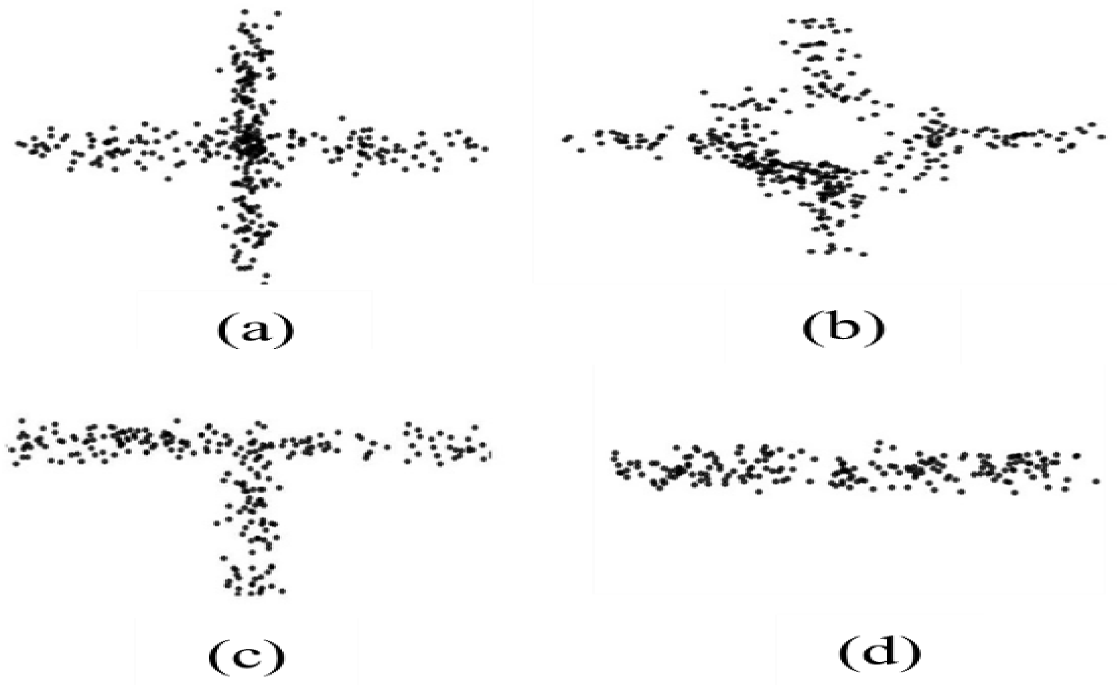


Figure 6 An image of a simulated example of (a) Cross-intersection; (b) Roundabout; (c) T-intersection; (d) Non-intersection

Chapter 5: Generalization test and results

1. Baseline experiment

The five classifiers selected for this study are tested on three different datasets including T-Drive, Cab-spotting, and ECML-PKDD 15. In each dataset, the same 10,000 images of simulated data are used for the first phase of the training and then the model is fine-tuned using a real-world training dataset. Consequently, 500 images of each real-world dataset are used for testing.

Based on the results shown in **Error! Reference source not found.**, The results show that DenseNet has the best F1 score across all three datasets, followed by ResNet, EfficientNet, and MobileNet. NasNet has the lowest F1 scores.

There are a few possible explanations for these results.

One possibility is that DenseNet has the optimal combination of parameters and layers. DenseNet has a high number of parameters, which allows it to learn complex patterns in the data. However, it also has a low number of layers, which prevents it from overfitting the data. ResNet and EfficientNet also have a high number of parameters, but they have a larger number of layers than DenseNet.

This may be why they have slightly lower F1 scores than DenseNet. NasNet had the lowest F1 scores. This is likely because they have a lower number of parameters than the other classifiers.

This means that they are not able to learn complex patterns in the data, which leads to lower accuracy. NasNet also has a large number of layers, which may contribute to its low F1 score. The large number of layers may make it more difficult for NasNet to generalize new data.

Overall, the results shown in **Error! Reference source not found.** suggest that DenseNet is the best classifier for this task. However, ResNet and EfficientNet are also good choices. MobileNet and NasNet should be avoided, as they have lower F1 scores.

Here are some additional details about the results:

- The ECML-PKDD dataset is the most challenging, and DenseNet achieves the highest F1 score on this dataset. This suggests that DenseNet is particularly well-suited for challenging tasks.
- The cab-spotting dataset is the least challenging, and all of the classifiers achieve high F1 scores on this dataset. This suggests that the differences in performance between the classifiers are not as pronounced in this dataset.
- The T-drive 15 dataset is a middle-ground between the T-drive and cab-spotting datasets, and the results show that DenseNet and ResNet achieve the highest F1 scores on this dataset.

Here is a more detailed explanation of the role of parameters and layers in the performance of these classifiers:

- Parameters are the weights and biases of the neural network. The more parameters a neural network has, the more complex patterns it can learn. However, more parameters can also lead to overfitting, which is when the neural network learns the training data too well and does not generalize well to new data.
- Layers are the building blocks of a neural network. Each layer performs a specific operation on the data, such as convolution, pooling, or activation. The more layers a neural network has, the deeper it is. Deeper neural networks can learn more complex patterns, but they are also more prone to overfitting.

The results shown in **Error! Reference source not found.** suggest that there is an optimal balance between parameters and layers for image classification tasks.

DenseNet, ResNet, and EfficientNet all have a high number of parameters, but they have different numbers of layers. DenseNet has the fewest layers, followed by ResNet and EfficientNet.

This suggests that the optimal number of layers for image classification tasks is relatively small. The results also suggest that the number of parameters is more important than the number of layers for image classification tasks.

DenseNet has the highest number of parameters, followed by ResNet and EfficientNet. This suggests that DenseNet can learn more complex patterns in the data than the other classifiers, which leads to better accuracy.

Overall, the results shown in Figure 7 suggest that DenseNet is the best classifier for this task. However, ResNet and EfficientNet are also good choices. MobileNet and NasNet should be avoided, as they have lower F1 scores.

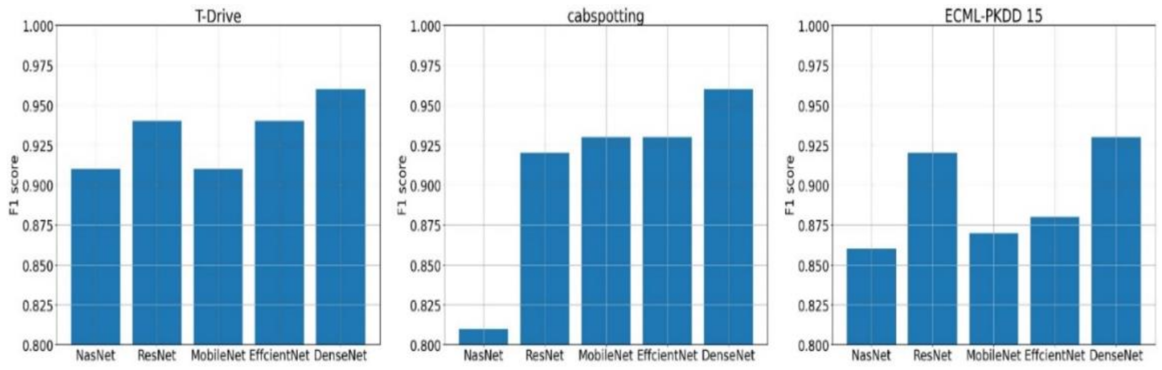


Figure 7 F1 scores for the five classifiers applied to the three datasets

2. Cross-Testing Models

Table II Cross-testing method

The model	Test on T-Drive	Test on San-Fran	Test on Porto
Train on T-Drive (ABC)	D set of T-Drive	All real of San-Fran	All real of Porto
Train on T-Drive (ABD)	C set of T-Drive	All real of San-Fran	All real of Porto
Train on T-Drive (ACD)	B set of T-Drive	All real of San-Fran	All real of Porto
Train on T-Drive (BCD)	A set of T-Drive	All real of San-Fran	All real of Porto
Train on San-Fran (ABC)	All real of T-Drive	D set of San-Fran	All real of Porto
Train on San-Fran (ABD)	All real of T-Drive	C set of San-Fran	All real of Porto
Train on San-Fran (ACD)	All real of T-Drive	B set of San-Fran	All real of Porto
Train on San-Fran (BCD)	All real of T-Drive	A set of San-Fran	All real of Porto
Train on Porto (ABC)	All real of T-Drive	All real of San-Fran	D set of Porto
Train on Porto (ABD)	All real of T-Drive	All real of San-Fran	C set of Porto
Train on Porto (ACD)	All real of T-Drive	All real of San-Fran	B set of Porto
Train on Porto (BCD)	All real of T-Drive	All real of San-Fran	A set of Porto

A. T-drive results

The main goal of this step is to ensure that the framework can be generalized by testing it on a separate set of data (i.e., test set) to ensure that it can make accurate predictions on new, unseen data. The trained models on each dataset are saved individually and then tested on the datasets they were not initially trained on.

To visualize the results, the Kruskal test is used to compare the distribution of performance metrics of the model when tested using two different datasets and assuming a cut-off p-value of 0.05.

The Kruskal test is used because the results are not uniformly scattered. Figure 8 shows the results on T-drive.

From Figure 8, The results show that DenseNet has the best F1 score on the T-drive dataset, followed by ResNet and EfficientNet.

However, DenseNet is the only classifier that generalizes well to the other two datasets. ResNet is rejected on both the cab-spotting and ECML-PKDD 15 datasets, while EfficientNet is accepted on the cab-spotting dataset but rejected on the ECML-PKDD 15 dataset.

NasNet and MobileNet are rejected on both the cab-spotting and ECML-PKDD 15 datasets.

There are a few possible explanations for these results.

One possibility is that DenseNet can learn more generalizable features than the other classifiers. This is likely since DenseNet has a higher number of parameters, which allows it to learn more complex patterns in the data.

Another possibility is that the other classifiers are overfitting the training data. This is likely since the other classifiers have a larger number of layers, which makes them more prone to overfitting.

The results also suggest that the ECML-PKDD 15 dataset is more challenging than the other two datasets. This is likely since the ECML-PKDD 15 dataset has a larger amount of noise. This noise can make it more difficult for the classifiers to learn generalizable features.

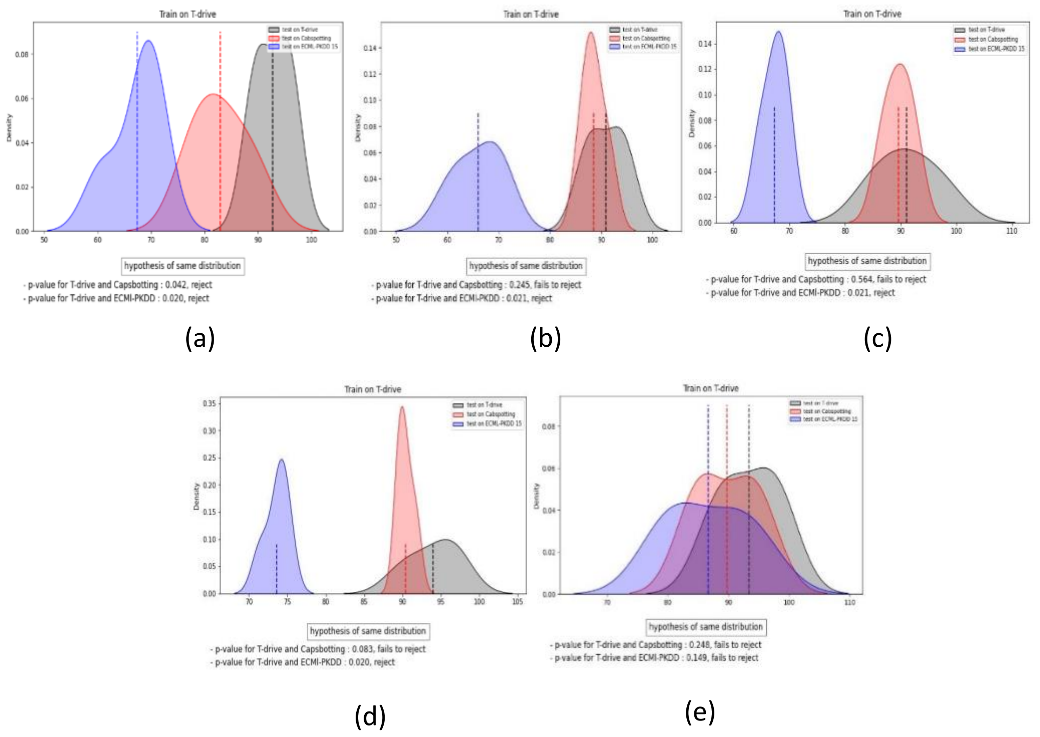


Figure 8 Kruskal Test on T-drive. (a) ResNet; (b) NasNet; (c) MobileNet; (d) EfficientNet; (e) DenseNet.

B. cab-spotting results

Error! Reference source not found. shows the results of cab-spotting. From **Error! Reference source not found.**, it can be seen that DenseNet still generalizes well on the other two datasets.

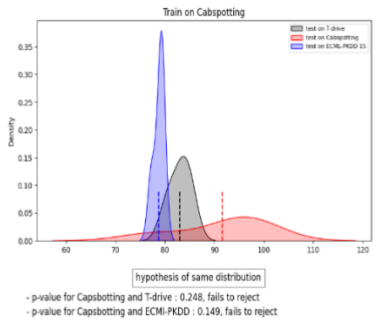
It can also be seen that even though MobileNet and EfficientNet are the second-best performing models, MobileNet is rejected from both datasets while EfficientNet failed to reject from both.

It can also be seen that even though ResNet has a lower score than MobileNet, it failed to reject both. And while NasNet performs the worst here with an F1 score of 81%, it still fails to reject from the T-drive dataset.

We can conclude from that

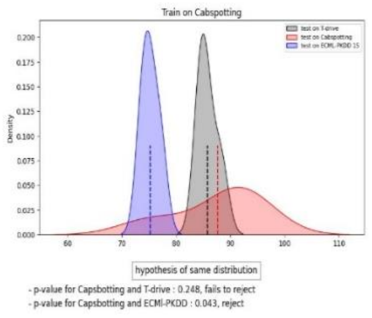
- ResNet can generalize well if trained on cab-spotting
- Although NasNet had the lowest f1 score, when trained on cab-spotting it can generalize well for the t-drive dataset
- Although MobileNet had a good score it could not generalize well on either of the datasets
- EfficientNet had the same score as MobileNet, but it could generalize well on both datasets
- DenseNet has the highest score and could generalize well on both datasets

We can conclude from these points that a high f1 score doesn’t determine if we would be able to generalize well on other datasets



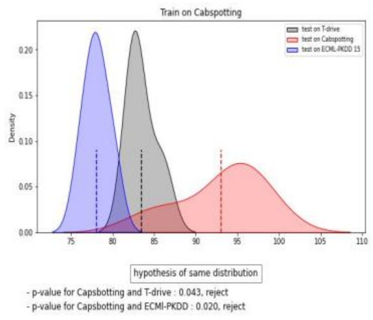
- p-value for Capsbotting and T-drive : 0.248, fails to reject
- p-value for Capsbotting and ECMI-PKDD : 0.149, fails to reject

(a)



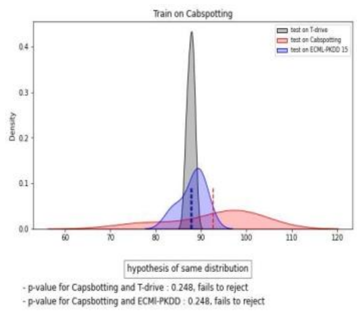
- p-value for Capsbotting and T-drive : 0.248, fails to reject
- p-value for Capsbotting and ECMI-PKDD : 0.043, reject

(b)



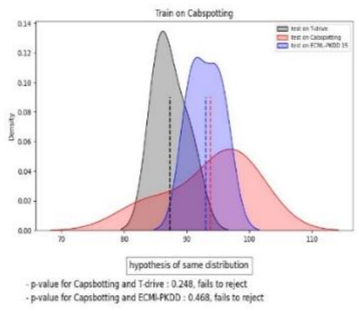
- p-value for Capsbotting and T-drive : 0.043, reject
- p-value for Capsbotting and ECMI-PKDD : 0.020, reject

(c)



- p-value for Capsbotting and T-drive : 0.248, fails to reject
- p-value for Capsbotting and ECMI-PKDD : 0.248, fails to reject

(d)



- p-value for Capsbotting and T-drive : 0.248, fails to reject
- p-value for Capsbotting and ECMI-PKDD : 0.468, fails to reject

(e)

Figure 9 Kruskal Test on cab-spotting. (a) ResNet; (b) NasNet; (c) MobileNet; (d) EfficientNet; (e) DenseNet.

C. ECML-PDKK 15 results

Error! Reference source not found. shows the results on ECML-PKDD 15. From **Error! Reference source not found.**, it can be seen that ResNet, which has the second-best score, is also rejected from both datasets, while EfficientNet and NasNet are not rejected from both datasets. Even though NasNet has the lowest score out of all the five classifiers. MobileNet is also rejected from T-drive similar to DenseNet.

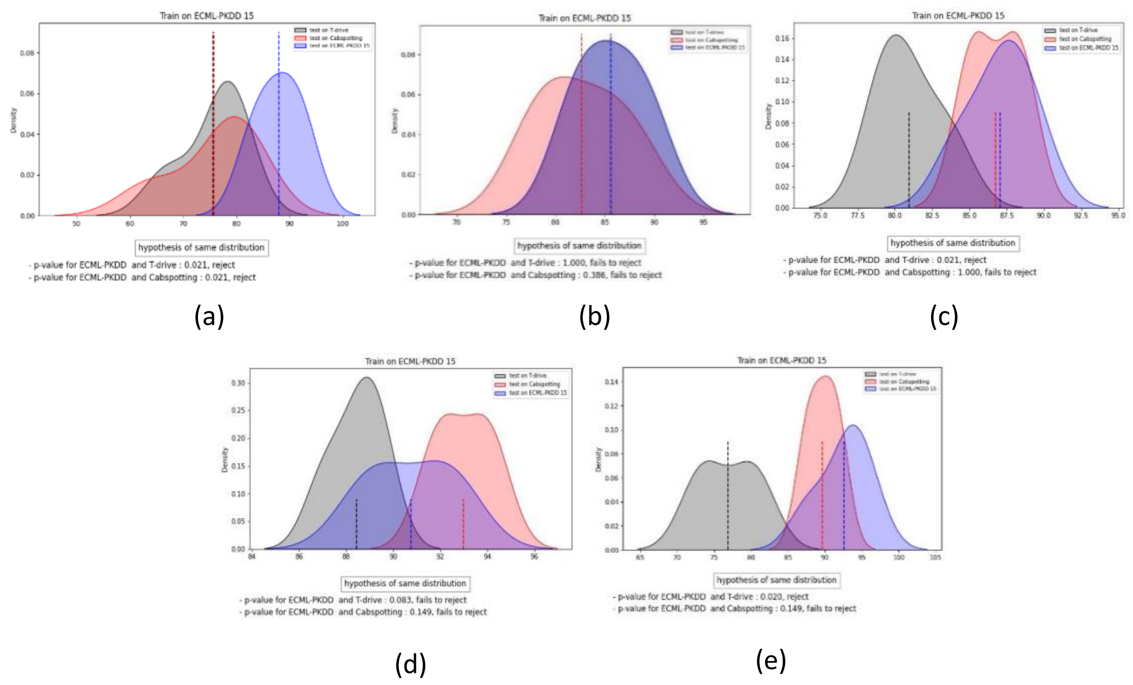


Figure 10 Kruskal Test on ECML-PKDD 15. (a) ResNet; (b) NasNet; (c) MobileNet; (d) EfficientNet; (e) DenseNet

Chapter 6: vehicle trajectories features

1. trajectory data as images

to take advantage of the trajectory data and increase the accuracy, density, speed, heading, and acceleration features are extracted. Extracting features from trajectories involved quantifying the movement patterns of vehicles in terms of various attributes such as speed, heading, acceleration, and density. These features can be derived from the raw trajectory data. This can be achieved by dividing the space into a grid of equal cells and using the trajectory data points inside each cell to estimate the feature we are interested in.

Multi-channel images are visual representations of the trajectory features, where each channel corresponds to a particular feature and its values are mapped to pixel intensities or color values. The images are typically created by stacking multiple channels on top of each other to form a composite image. For example, a multi-channel image for a set of trajectories of a vehicle might have the following channels:

- Speed channel, which represents the speed of each car at each point in time, with faster speeds shown in brighter colors
- Heading channel, which represents the direction that each car is traveling, with different colors for different directions
- Acceleration channel, which represents the rate of change of speed of each car, with positive acceleration shown in one color and negative acceleration shown in a different color

By overlaying these channels, a multi-channel image can provide a more comprehensive view of the movement patterns and interactions between cars, which can be useful in understanding and analyzing traffic flow dynamics. No specific maximum number of channels can be used in generating multi-channel images from trajectory data, as the number of channels depends on the number of features extracted and visualized. In practice, the number of channels used in these images is usually determined by the number of relevant features and the amount of information that needs to be conveyed. However, it is important to keep in mind that as the number of channels increases, the complexity of the image also increases, which can make it more difficult to interpret and analyze. Therefore, it is important to strike a balance between the number of channels used and the clarity and interpretability of the resulting image. Hence, the maximum number of channels used in this work is determined to be three channels. As such, a fourteen combination of the four features, (i.e. $4_{C_1} + 4_{C_2} + 4_{C_3}$), can create images for training. With five different DNN algorithms, there are about 70 models that can classify intersections.

2. PTRAIL: A Parallel trajectory data preprocessing library

PTRAIL is an advanced library for preprocessing mobility data, specifically focused on trajectory data. It offers various features such as data filtering, feature generation, and trajectory interpolation.

A. The main features of PTRAIL:

- PTRAIL utilizes parallel computation using Python Pandas and NumPy, resulting in high-speed performance compared to other available libraries.
- By utilizing all available cores in the computer, PTRAIL fully harnesses the computational power of the machine it runs on.
- PTRAIL introduces a custom DataFrame designed on top of Python Pandas, providing an efficient representation and storage solution for trajectory data.
- The library offers a range of temporal and spatial features, computed primarily using parallel computation, ensuring fast and accurate calculations.
- The library provides multiple functionalities for filtering and cleaning noise in trajectory data.
- innovative aspect of PTRAIL is its provision of four distinct trajectory interpolation techniques.

B. The PTRAIL pipeline:

- In the PTRAIL pipeline, trajectory points are first inputted into the PTRAILDataFrame. Once a valid file is loaded, all functionalities become accessible. Each trajectory loaded as a PTRAILDataFrame is vectorized, and the applied functionalities are executed in parallel.
- Users can apply filters to remove noise and/or perform data smoothing through interpolation.
- Finally, higher-quality data sets can be used to extract features, and the data can be exported to common formats for compatibility with other libraries.

The PTRAIL pipeline is shown in Figure 1.

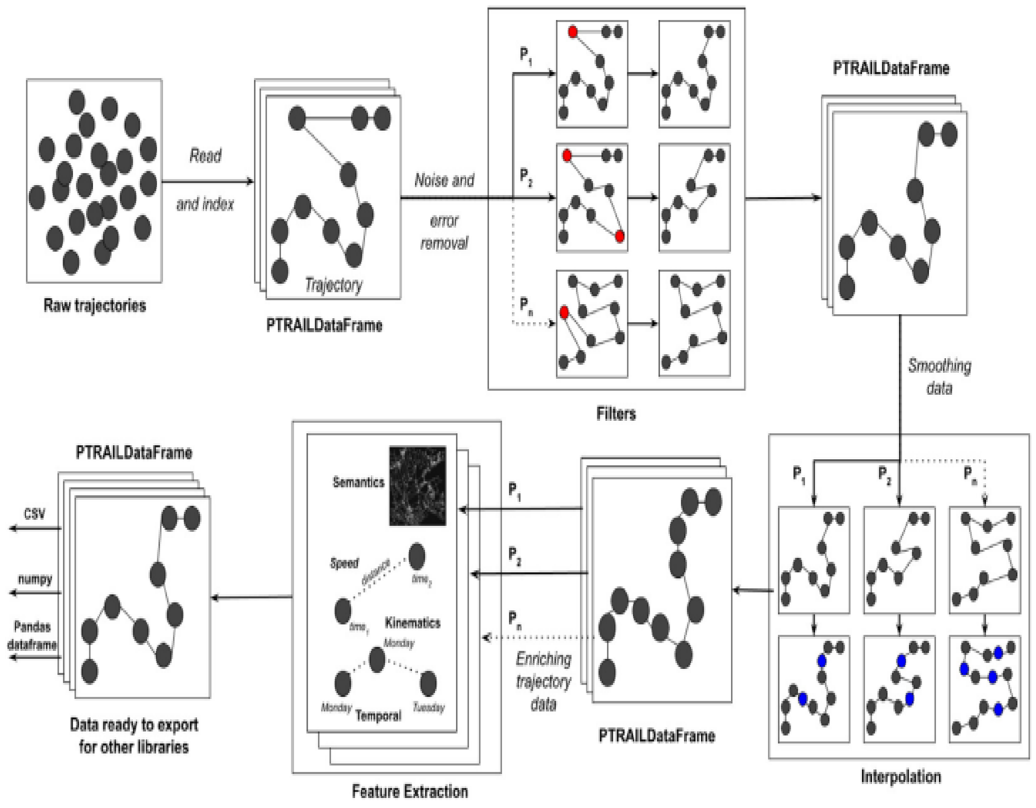


Figure 11 PTRAIL pipeline

3. Acceleration, speed, and heading extraction

In addition to the density, three other features are extracted including acceleration, speed, and heading.

The PTRAIL library [33] is used, which uses trajectory data to extract features using the NumPy library and parallel computing which makes it relatively faster in computations.

Extracting features is performed through the following steps.

- First, the type of data frame is converted to a PTRAIL Data Frame using the ***ptrail.core.TrajectoryDF*** module.

- Second, using the **ptrail.features.kinematic_features.KinematicFeatures** module, three features are extracted by applying three corresponding methods; **create_acceleration_column** for acceleration, **create_speed_column** for speed, and **create_bearing_column** for heading.

- Third, trajectories are calculated from a bounding box which typically involves analyzing the movement of an object within a given space. A bounding box is a rectangular shape that is used to enclose a region of interest (i.e., intersection). In the context of trajectory calculation, a bounding box is used to define the position of the vehicle within the defined region.

- Finally, The **ptrail.preprocessing.filters** module is used to apply filters to the dataset of trajectories before any analysis or visualization. Filters can help to remove noise, reduce data redundancy, and improve the quality of the data. These filters can help to improve the accuracy and reliability of the trajectory data and to make it easier to analyze and visualize the data.

4. Density feature extraction

To extract the density feature using a Gaussian kernel, a grid of cells that covers the spatial extent of the trajectory data is defined. For each trajectory point, its contribution to the density of each cell is calculated using a Gaussian kernel.

The Gaussian kernel has a bell-shaped curve that assigns higher weights to nearby points and lower weights to farther points. The formula for a 2D Gaussian kernel is:

$$K(X,Y) = \frac{1}{\sqrt{2 * \pi * \sigma}} * e^{\frac{-((x_1 - x_0)^2 + (y_1 - y_0)^2)}{2 * \sigma^2}}$$

where σ is the standard deviation of the Gaussian kernel, x, and y are the distances from the trajectory point to the center of the cell.

Then, the sum of contributions of all trajectory points to each cell is calculated to obtain the cell density. Cell densities are normalized by dividing by the total number of trajectory points.

5. vehicle trajectories features` as input to the models

To use the extracted features as inputs to the model, they need to be represented as images. The features are placed in the RGB channels of the photo, which means there can only be a maximum of three combinations in a single image.

As mentioned earlier, four features are extracted, with fourteen possible unique feature combinations including density (D), speed (S), acceleration (A), heading (H), density-speed (DS), density-acceleration (DA), density-heading (DH), speed-acceleration (SA), speed-heading (SH), acceleration-heading (AH), density-speed-acceleration (DSA), density-speed-heading (DSH), density-acceleration-heading (DAH), and speed-acceleration-heading (SAH).

The following steps are used to create the grid, illustrated in Figure 1:

1) Determining the minimum and maximum of the longitude and latitude,

2) Choosing the appropriate grid size (e.g., 20x20 and 30x30),

3) Calculating the longitude and latitude step using the following equation: $step = \frac{|\max - min|}{grid\ length * 2}$,

4) Creating a 3D matrix with the first two dimensions equal to the grid dimensions and the third dimension equal to two which corresponds to the longitude and latitude of the grid cell center, the matrix is initialized with zeros,

5) Calculating the center points of the grid cell using the following equations:
$matrix[x, y, 0] = \max longitude - (1 + 2 * x) * step\ longitude$
$matrix[x, y, 1] = \min latitude + (1 + 2 * y) * step\ latitude$,

6) Initializing four grids with the same dimensions of the grid for each feature with zeros,

7) Determining the weights of the Gaussian Kernel, where the distance between the two points is less than or equal to three times the sigma value as follows:
$Distance = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}$,

8) Extracting the required features, where the distance between every two points does not exceed the distance limit,

9) Completing the features grid with the result of weights of the Gaussian kernel matrix by calculating the dot product with the feature matrix,

10) Transposing and flipping the matrices of features, so they look identical to the original intersection photo,

11) Applying the min-max scaling on the matrix so it can fit in the range of the RGB channel

12) Placing the feature grids in the RGB channel of the images

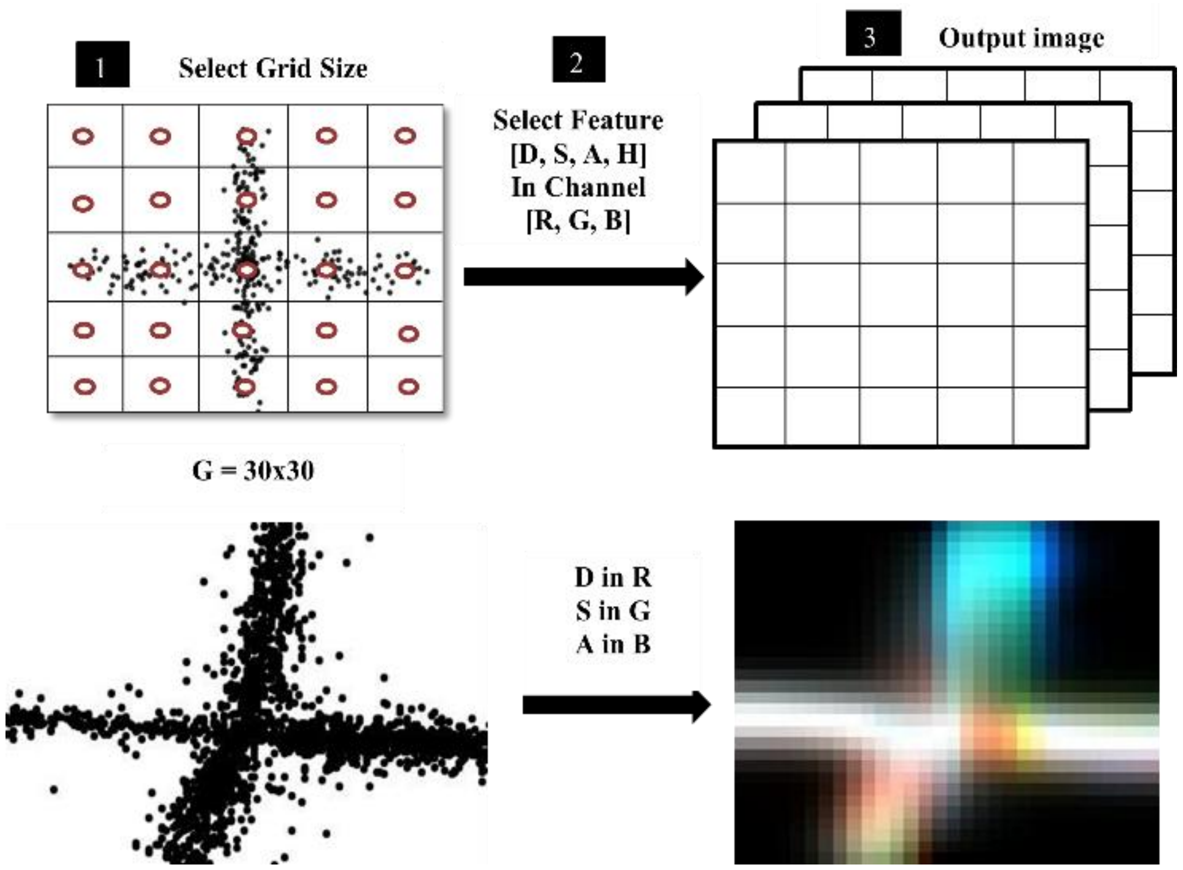


Figure 12 Illustrating the method of generating features.

Chapter 7: Models Ensemble

1. Ensemble learning

Sometimes one model is not enough to achieve high accuracy as the prediction of the model is based on what pattern the models see during the training process, so by using ensemble learning we can increase the overall accuracy.

Ensemble learning is a machine learning approach that combines multiple models in the prediction process to achieve better predictive performance. The reason why we need to use ensemble learning:

- High variance: model is very sensitive to the provided inputs
- Low accuracy
- Features noise and bias: the model relies on too few features while making a prediction

Aggregating predictions

When we use an ensemble learning algorithm to adapt the prediction of the combination of multiple models it is required an aggregation method.

The Aggregating techniques:

- Max Voting: For classification problems, the ultimate prediction is determined through a voting mechanism. The model with the highest number of votes among the ensemble is selected as the final prediction.

- Averaging: Employed in regression problems, this technique involves calculating the average of the predictions generated by each model in the ensemble. The resulting average is considered as the final prediction. Additionally, in classification tasks, probabilities can also be averaged to derive the final classification.

- Weighted Average: In certain cases, it becomes necessary to assign specific weights to individual models or algorithms within the ensemble. This approach allows us to emphasize the predictions of certain models by assigning higher weights, ultimately influencing the final prediction. By applying weighted averaging, we can incorporate the expertise and strengths of each model in a controlled manner to improve the overall prediction accuracy.

Ensemble learning has three main methods:

- bagging
- boosting
- stacking

A. BAGGING (Bootstrap Aggregation)

This method involves using a single machine learning algorithm and training each model on a different sample of the same training data set, and then the prediction made of this combined model is done by voting or averaging.

Bagging reduces variance and minimizes overfitting because every model is trained on a unique sample of the dataset.

The technique used to create the sample is bootstrap sampling. Bootstrap sampling is a method in which a certain number of equally sized subsets of datasets are extracted with replacement.

replacement means that if a row is selected, it is returned to the training dataset for reselection in the same training dataset, which that for a specific training dataset, a row of data may be chosen 0 or 1 or many times.

In this way, multiple different training datasets may be prepared to use as unique data set to train different models.

Prediction from the models trained on the different datasets is better than model one model fit to the original dataset which will achieve better accuracy.

We can summarize the method of bagging as follows:

- Bootstrapping uses multiple sets of the original dataset with replacement.
- all subsets have equal size and is used to train models.
- Aggregate the prediction of the models using voting or averaging.

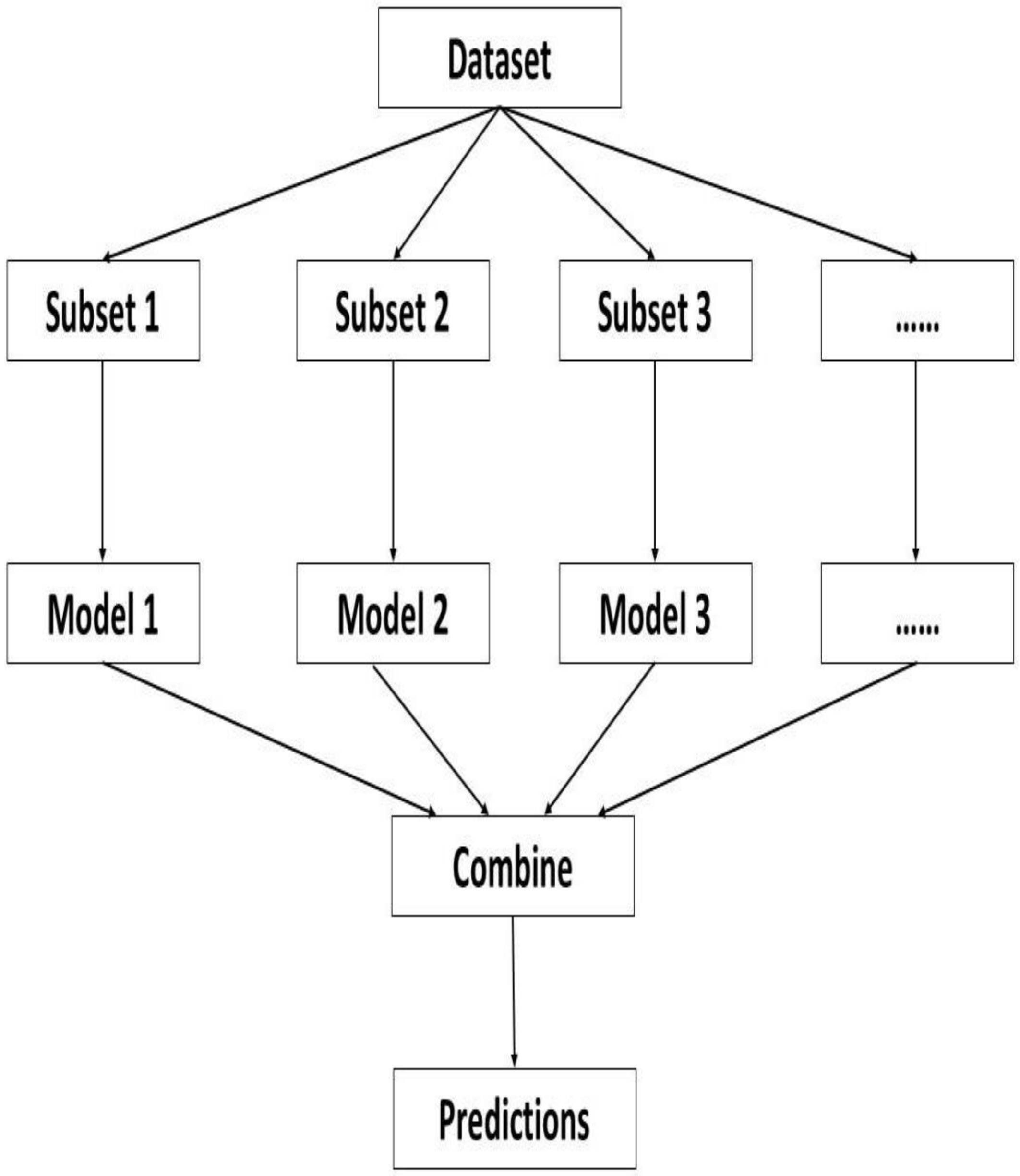


Figure 13 Bagging Method.

popular ensemble algorithms are based on the bagging method:

- Bagged Decision Tree
- Random Forest
- Extra Tree

B. Boosting

Boosting, an ensemble technique, aims to modify the training data to prioritize instances that were previously misclassified by models trained on the dataset.

The primary characteristic of boosting ensembles lies in their ability to rectify prediction errors. Models are sequentially trained and added to the ensemble, with each subsequent model attempting to improve upon the predictions of its predecessor. This sequential process continues, with each model attempting to correct the errors of the previous model and refine the overall ensemble's performance.

Within this ensemble method, weak learners are employed. In the context of boosting, weak learners typically refer to basic decision trees that make only a limited number of decisions. These learners are deemed weak due to their simplicity and limited predictive capabilities.

The predictions of these weak learners are aggregated through voting or averaging, with each learner's contribution weighted proportionally to its performance or competence. The ultimate objective is to construct a "strong learner" by combining multiple purpose-built weak learners.

It is important to note that the boosting method does not alter the training dataset itself. Instead, the learning algorithm adjusts its focus on specific examples based on whether they were correctly or incorrectly predicted by previously added ensemble members.

Popular ensemble algorithms based on Boosting method:

1) AdaBoost, a type of ensemble algorithm, employs multiple weak learners to construct its models. The remarkable adaptability of AdaBoost has established it as one of the pioneering binary classifiers, demonstrating early success in the field.
2) Gradient boosting, an exceptional methodology known for its superior predictive capabilities, encompasses a range of powerful algorithms. Prominent examples include Xgboost, LightGBM, and CatBoost.

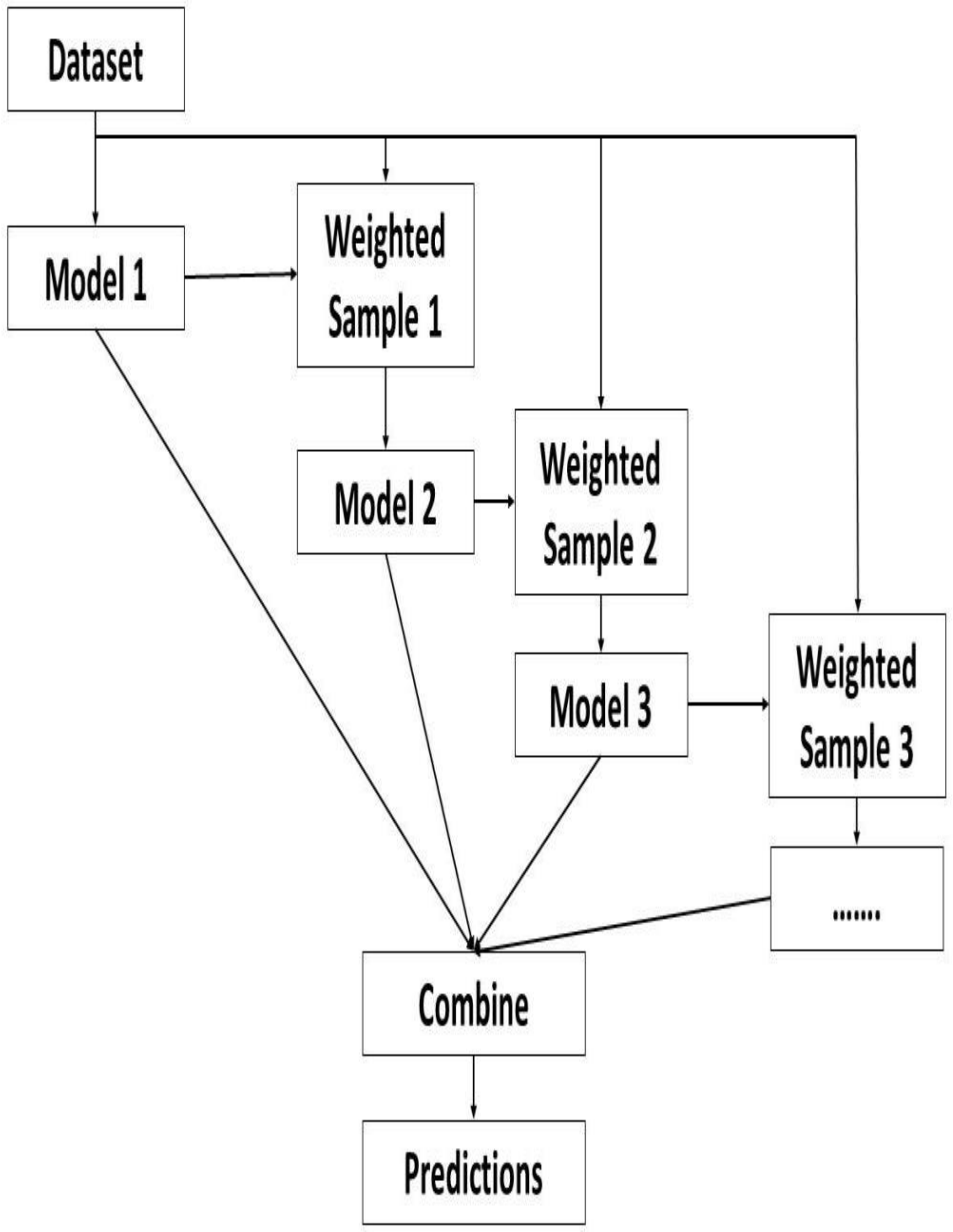


Figure 14 Boosting method.

C. Stacking

stacking focuses on building a diverse set of members by employing different types of models on the training data, and then utilizing a model to combine their predictions. This approach, akin to boosting models, yields more resilient predictors, enhancing overall model performance. Stacking entails the process of discovering methods to construct a stronger model from an ensemble of weak learners.

In stacking, specific terminology is used to describe its components. The individual members of the ensemble are commonly referred to as level-0 models, while the model responsible for combining the predictions is known as the level-1 model. Although the two-level hierarchy of models is the most prevalent approach, it is worth noting that stacking can involve the use of multiple layers of models for more intricate architectures.

Stacking builds a meta-learner model, which takes the base model predictions as input features and the corresponding ground truth labels as the target variable. The meta-model is trained to learn how to combine the predictions of the base models to make the final prediction.

The main idea behind stacking is that the meta-model can learn to leverage the strengths of the base models and effectively combine their predictions to achieve better performance than any individual base model alone. By using different types of models or models trained with different algorithms as base models, stacking can capture diverse perspectives and improve overall prediction accuracy.

We can summarize the method of Stacking as follows:

- Prepare the training data.
- Train multiple diverse base models.
- Use the base models to make predictions on unseen data.
- Build a meta-model that learns to combine the base model predictions.
- Use the meta-model to make the final prediction on new, unseen data.

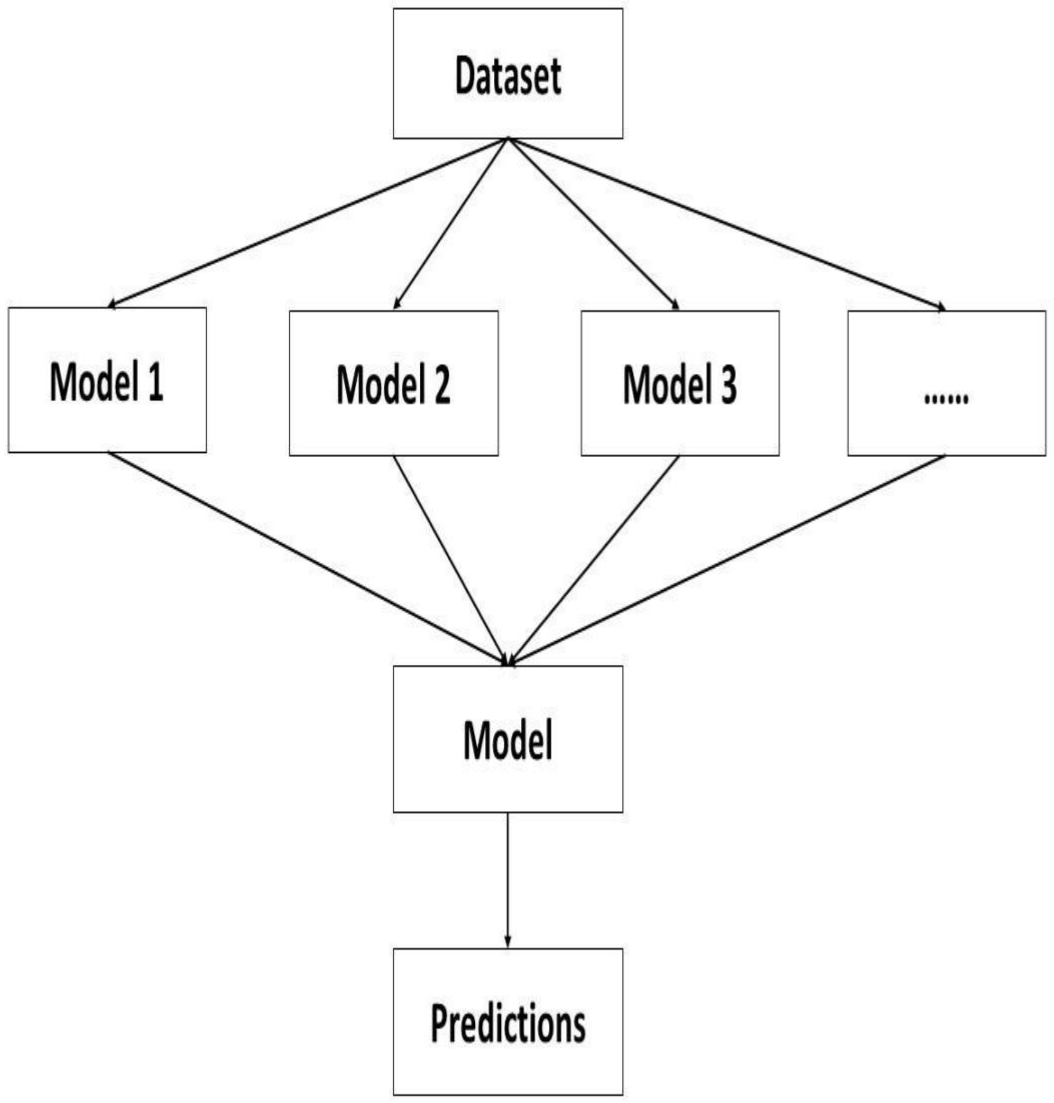


Figure 15 Stacking method.

Popular ensemble algorithms based on the Stacking method:

- Stacked Models (canonical stacking)
- Blending
- Super Ensemble

2. bootstrapping-based heuristic

bootstrapping-based heuristic is employed to choose the best-performing models.

Bootstrapping refers to a family of statistical methods that involve resampling from a dataset to estimate the uncertainty and variability of a statistical model or an estimator.

In bootstrapping, multiple resamples are created from the original dataset by randomly sampling observations with replacement.

Each resample is used to fit the statistical model or estimate the parameter of interest. This process is repeated many times, generating multiple bootstrap models or estimates.

The variability and uncertainty of the original model or estimator can be estimated by analyzing the distribution of the bootstrap models or estimates. This can provide insight into the stability and robustness of the original model or estimator and can help identify potential sources of bias or overfitting.

Bootstrapping can be used in various statistical applications, including regression analysis, hypothesis testing, and machine learning.

3. Ensemble trajectory-features models

Ensemble models can be applied to a wide range of machine learning tasks, including classification (i.e., as in this work), regression, and clustering. They are particularly effective in situations where individual models may have limited accuracy or reliability due to overfitting, noisy data, or bias.

In this step, an ensemble is built using the models trained with fourteen possible feature combinations and five different models. Therefore, 70 different models are trained independently. To select a good subset of models that yield an accurate ensemble, a stepwise regression approach (i.e. a greedy approach) is used.

Stepwise regression needs the models to be ordered using some metric. Using the accuracies of the stand-alone models in Figure 4 to order them for the stepwise regression is not a good choice because it does not reflect how well a model performs in an ensemble. Therefore, the marginal accuracies of the models are initialized using the algorithm described in the next subsection.

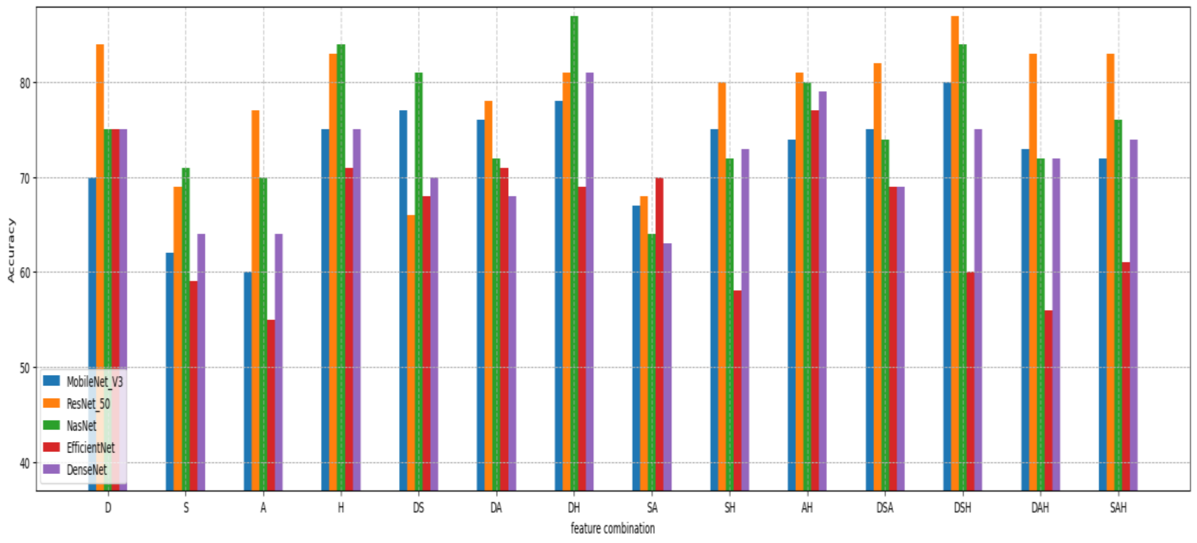


Figure 16 Classification accuracy of the five models with the 30x30 grid from the ECML PKDD 15 dataset.

Estimating the marginal accuracy of a model in an ensemble

This algorithm estimates the models' marginal accuracies based on randomly forming ensembles. The whole experiment is repeated (NE) times. Each time a randomly chosen subset of models is used. The number of models in each subset can be any random number from two to a maximum length (NF). This estimation method consists of the following four steps:

- The first step is ensemble generation. NE ensembles are generated randomly during this step, but model duplication is not allowed within the same subset.
- The second step is evaluation, where each ensemble is evaluated using the testing dataset. After the evaluation, each ensemble has its classification percentage.
- The third step is updating the marginal accuracy of individual models. The marginal accuracy of each model

is the average of the ensemble's classification accuracy, which contains this model.

- The final step is ordering the models in descending order according to their marginal accuracy.

To calculate the marginal accuracy of every model

- initialize a vector with the accuracies of the models and a counter equal to 1 then run a loop for a thousand iterations.

- In each iteration, a number of models between 2 and 10 is chosen randomly, and the chosen models are ensembled together, the resulting accuracy is added to every model accuracy in the initialized vector.

- Within the loop, the randomly chosen combination with the highest accuracy is saved.

- The models are ranked depending on the highest marginal accuracies and performed stepwise on the ranked models, starting with the 1st ranked model then ensemble with it the 2nd rated model, and continuing to ensemble the 70 models together.

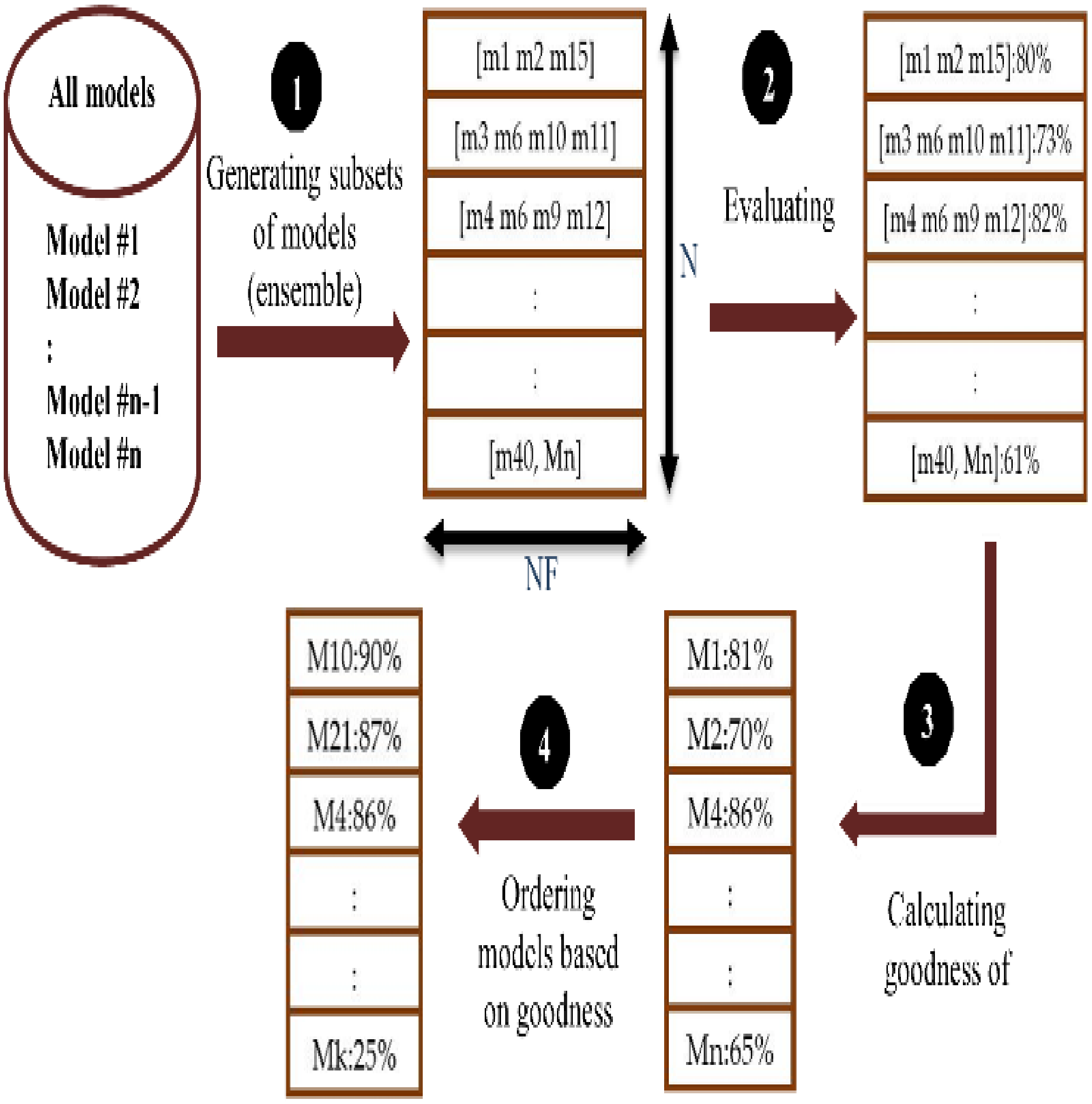


Figure 17 Illustration of the model marginal accuracy estimation.

Ensemble Model Results

Figure 6 shows the marginal accuracy results in descending order. ResNet-Heading (RN-H) has the highest accuracy while EfficientNet-Acceleration (EN-A) has the lowest accuracy.

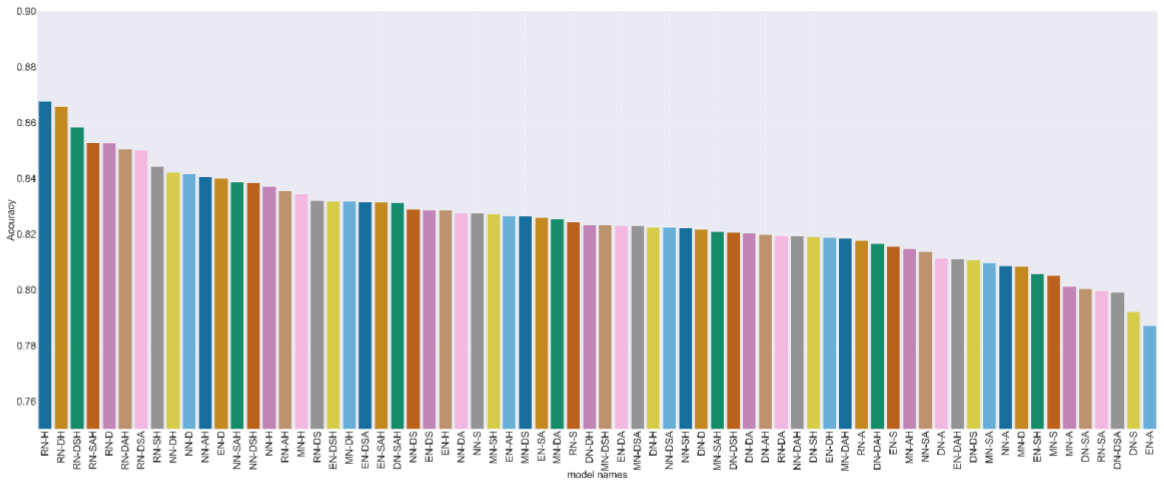


Figure 18 Classification accuracy results of the five models with the ensemble on the ECML PKDD 15 dataset.

Figure 7 shows the frequency of each of the chosen models, RN-SA and DN-D are the models chosen most (about 55 times), while some models are not chosen a lot such as EfficientNet-Heading (EN-H) (less than 30 times).

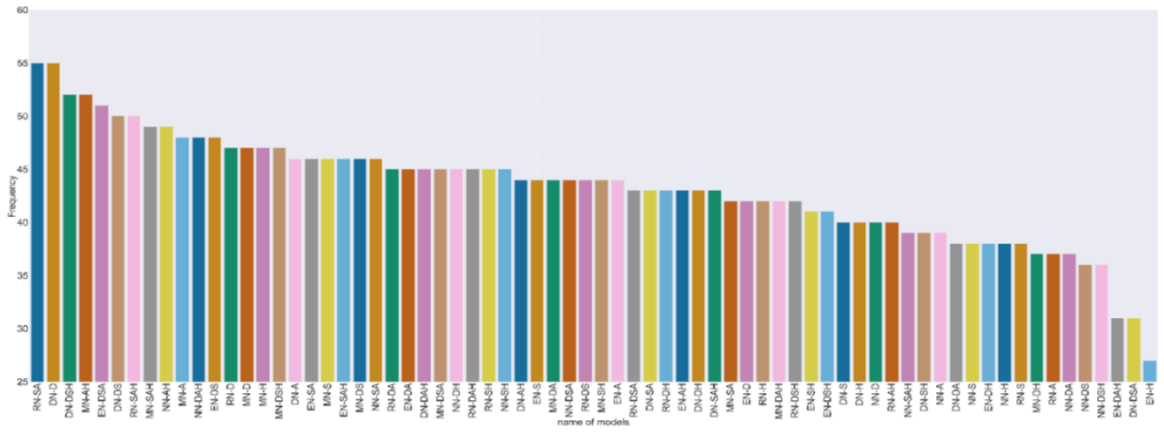


Figure 19 Frequency of the chosen models ensemble applied on the ECML PKDD 15 dataset.

*It is found that **MobileNet-SAH, NasNet-AH, MobileNet-SH, DenseNet-DA, EfficientNet-DA, ResNet-DAH, ResNet-DSH, EfficientNet-DSA** outperformed the other models and can be generalized with about **91%** classification accuracy.*

figure 8 shows The achieved accuracy levels range between **84-91%**, after performing the stepwise of combining the ordered models based on their marginal accuracy.

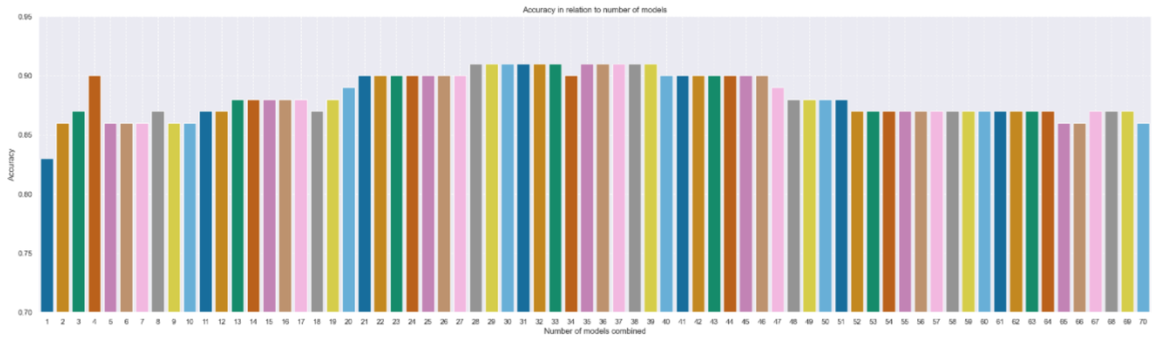


Figure 20 the result of accuracy after performing stepwise ordered models

Chapter 8: Object Detection

1. Introduction to Object Detection

Object detection is a crucial task in the field of computer vision, enabling machines to identify and locate objects within images or videos. In this chapter, we will provide an overview of object detection, its significance, applications, and the challenges it entails.

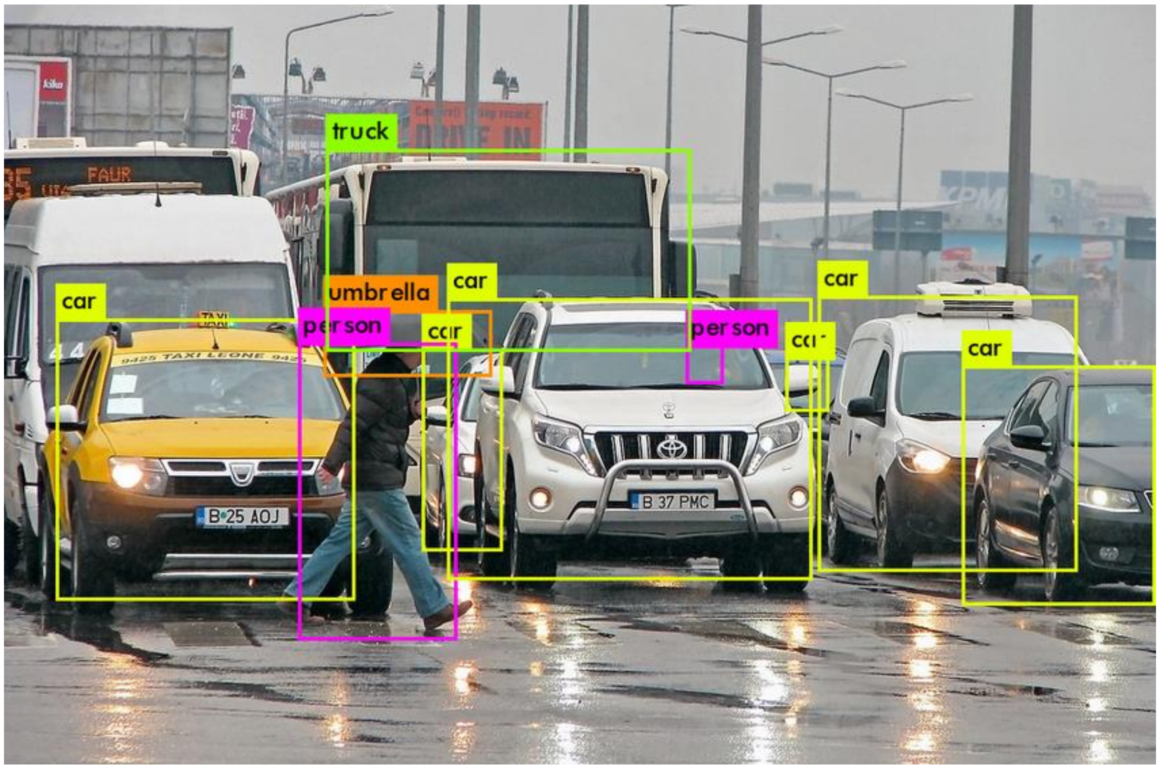


Figure 21 Example of object detection

A. What is Object Detection?

Object detection is the process of identifying and localizing objects of interest in digital images or videos. Unlike image classification, which classifies an entire image into predefined categories, object detection goes a step further by detecting and precisely delineating the locations of individual objects within an image.

B. Significance of Object Detection

Object detection plays a vital role in numerous domains, revolutionizing various industries and enabling groundbreaking applications. Some key areas where object detection has had a significant impact include:

- Autonomous Driving: Object detection is critical for self-driving cars to perceive and understand the surrounding environment, detecting pedestrians, vehicles, traffic signs, and obstacles.

- Surveillance and Security: Object detection is widely used in surveillance systems for detecting intruders, tracking suspicious activities, and enhancing public safety.

- Robotics: Robots employ object detection to navigate and interact with their surroundings, recognizing and manipulating objects in real-world scenarios.

- Medical Imaging: Object detection aids in medical diagnosis and treatment by identifying abnormalities in medical images, such as tumors, anatomical structures, or anomalies.

- Augmented Reality: Object detection allows for the precise alignment of virtual objects with real-world scenes, enhancing user experiences in augmented reality applications.

C. Challenges in Object Detection

Object detection poses several challenges due to the complexity and variability of real-world scenarios. Some of the key challenges include:

- Scale Variation: Objects can vary significantly in size, making it challenging to detect objects at different scales within an image.

- Occlusion: Objects may be partially occluded by other objects or occluded by the environment, making their detection more difficult.

- Background Clutter: Objects of interest can blend with complex backgrounds, leading to false positives or missed detections.

- Object Categories and Diversity: Objects belong to diverse categories with varying shapes, appearances, and poses, making it necessary to handle a wide range of object classes.

- Real-Time Performance: Many applications require real-time object detection, necessitating efficient algorithms and architectures to achieve high-speed processing.

D. Object detection techniques

Object detection techniques refer to the methods and algorithms used to identify and locate objects within images or videos. Here are some commonly employed object detection techniques:

1. Viola-Jones Algorithm: The Viola-Jones algorithm is a classic object detection approach that utilizes Haar-like features and the AdaBoost algorithm to detect objects

efficiently. It is known for its real-time performance and has been widely used in applications like face detection.

2. Histogram of Oriented Gradients (HOG): The HOG technique computes the gradients and orientations of image patches and represents them as feature vectors. These feature vectors are then used to train a classifier, such as Support Vector Machines (SVM), to detect objects based on their shape and appearance.

3. Selective Search: Selective Search generates a set of region proposals by grouping similar image regions based on various cues, such as color, texture, and size. These region proposals are then fed into a classifier to identify objects within the proposed regions.

4. R-CNN (Region-based Convolutional Neural Networks): R-CNN combines region proposal methods, such as Selective Search, with convolutional neural networks (CNNs). It generates region proposals and extracts features using CNNs, which are then classified using SVMs or softmax classifiers.

5. Fast R-CNN: Fast R-CNN improves upon R-CNN by sharing computation across different region proposals, enabling faster processing. It introduces a region of interest (RoI) pooling layer that extracts features from the entire image and applies them to each region proposal.

6. Faster R-CNN: Faster R-CNN further enhances the speed and accuracy of object detection by introducing a Region Proposal Network (RPN). The RPN generates region proposals directly from convolutional feature maps, eliminating the need for external proposal methods.

7. YOLO (You Only Look Once): YOLO is a one-stage object detection algorithm that treats object detection as a regression problem. It divides the input image into a grid

and predicts bounding boxes and class probabilities directly from each grid cell. YOLO achieves real-time object detection by making predictions in a single pass through the network.

8. SSD (Single Shot MultiBox Detector): SSD is another one-stage object detection approach that predicts objects at multiple scales within an image. It utilizes a set of default bounding boxes with different aspect ratios and applies convolutional filters at different feature maps to detect objects of various sizes.

9. RetinaNet: RetinaNet addresses the problem of class imbalance in object detection by introducing a focal loss that assigns higher weights to challenging examples during training. It utilizes a feature pyramid network (FPN) and a classification subnet to detect objects at different scales and levels of abstraction.

10. EfficientDet: EfficientDet is an efficient and scalable object detection architecture that combines the ideas of EfficientNet (efficient convolutional networks) and the BiFPN (bidirectional feature pyramid network). It achieves a good balance between accuracy and computational efficiency across different resource constraints.

These techniques represent a range of approaches with varying trade-offs in terms of speed, accuracy, and complexity. They have played a significant role in advancing the field of object detection and have been the foundation for many state-of-the-art models and frameworks.

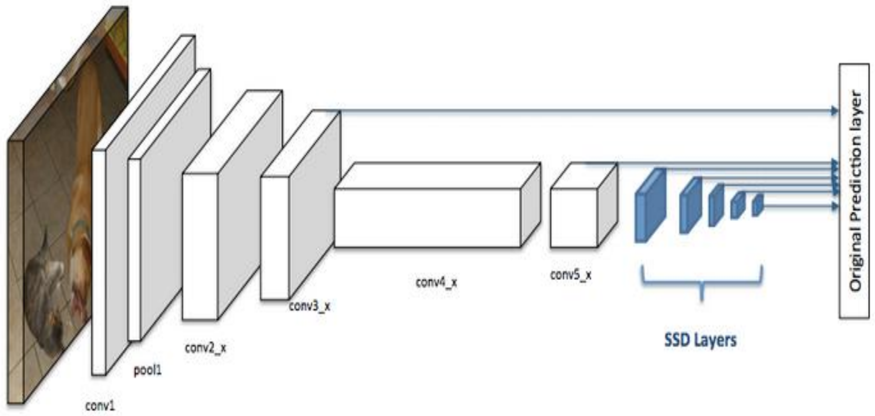


Figure 22 SSD models

2. Intersections Detection

Intersection detection is essential for map generation as the rules for creating intersection maps depend on the type of intersection, as shown in Figure x.

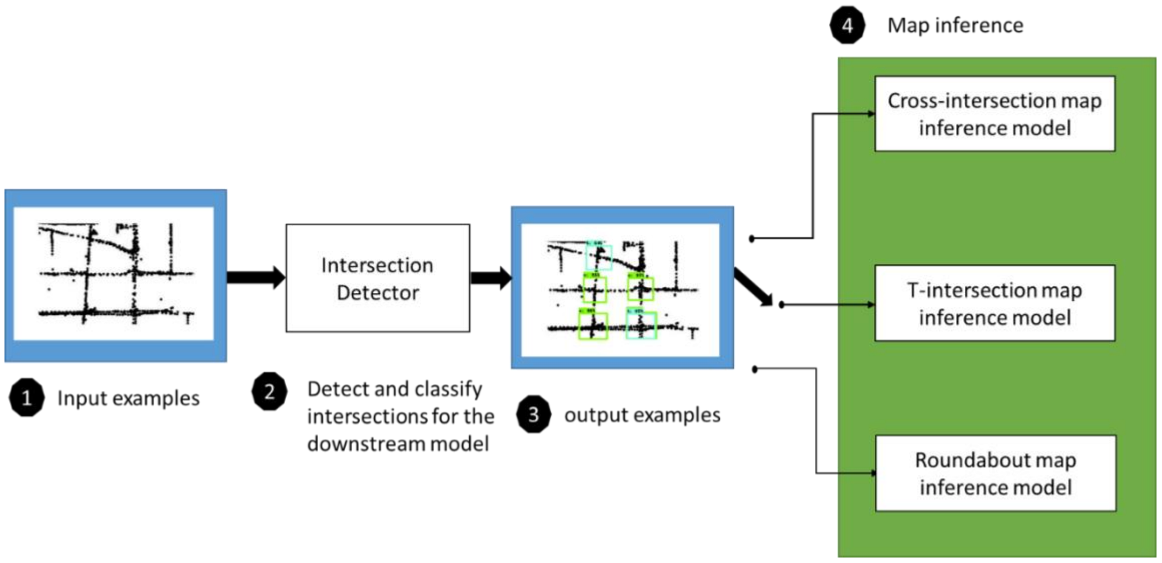

4 Map inference
Cross-intersection map inference model
T-intersection map inference model
Roundabout map inference model
Intersection Detector
1 Input examples
2 Detect and classify intersections for the downstream model
3 output examples

Figure 23 Illustration of the need for object detectors for intersection map inference

By detecting intersections according to their geometry, it is possible to generate road networks that include appropriate junction types in the right locations. Detection models have two tasks to perform; localization in which the model tries to accurately locate the object, and classification in which the model tries to classify the object. In this study, we used detection models to classify intersections in two different datasets using Deep Neural Network.

We used two different datasets and applied the CNN model. The T-Drive dataset [1, 2] has a total distance of the trajectories that reaches nine million kilometers, an average sampling interval of about 177 seconds, and a distance of about 623 meters. ECML-PKDD 15 dataset [3] contains the trajectories of taxis of about 1.7 million instances and each data sampled corresponds to one complete trip.

First, we converted the longitude and latitude features into images to use them as a training dataset for the CNN model. We extracted random areas from the map within the area of interest. We then created the bounding box and labeled it for each intersection. The extracted dataset consists of about 2000 images 20% of which were used for testing.

The program used to make the dataset is LabelImg and here are some of the benefits of using LabelImg:

- It is open-source and free to use.
- It is easy to install and use.
- It can be used to label images in a variety of formats.
- It supports a variety of annotation formats, including PASCAL VOC, YOLO, and CreateML.
- It is a powerful tool that can be used for a variety of computer vision tasks.

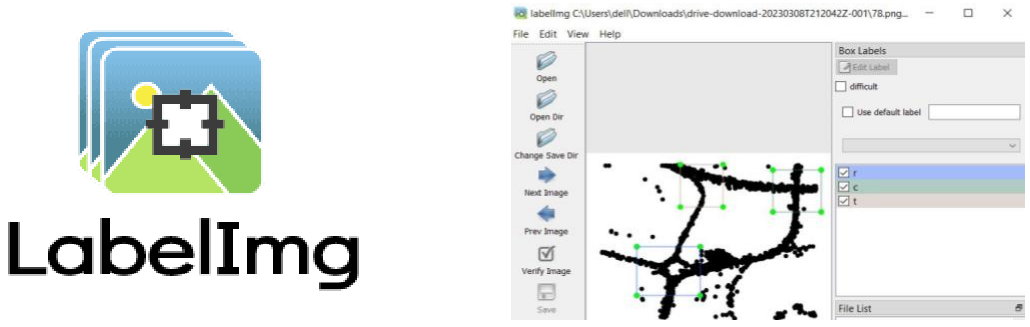


Figure 24 Using LabelImg to make the dataset

We identified three classes, namely, T-intersection, cross-intersection, and round-intersection. We decided to give attention to location error and reduce that error to the lowest degree so we used the Single-Shot Detection (SSD) models that could provide low location loss.

The cross-intersection class was more common than the other two classes of intersections in the datasets, so the numbers of intersections might be slightly skewed. Results showed that the confidence score ranges for the intersections were 42-94%, 51-98%, and 66-94% for the T-intersection, cross-intersection, and round-intersection, respectively. The mean average precisions (mAP) for each class were about 15.86%, 18.96%, and 8.19% for the T-intersection, cross-intersection, and round-intersection, respectively. The mAP results are relatively low, but the results can be considered promising for a dataset of fewer than 2000 images and only tested on one type of model. For future recommendations, we believe that trying different CNN models like Faster-CNN or any of the R-CNN family and expanding the dataset will result in better prediction estimates.

what is the R-CNN family?

Region-based Convolutional Neural Networks combine region proposal methods with CNNs for object detection, using bounding box proposals, feature extraction with CNNs, classification with SVMs or softmax classifiers, and refinement with regression models. It improved accuracy but suffered from computational inefficiency. Variants like Fast R-CNN and Faster R-CNN addressed these issues, sharing computations and introducing Region Proposal Networks (RPNs). R-CNN family models are widely used and have advanced object detection.

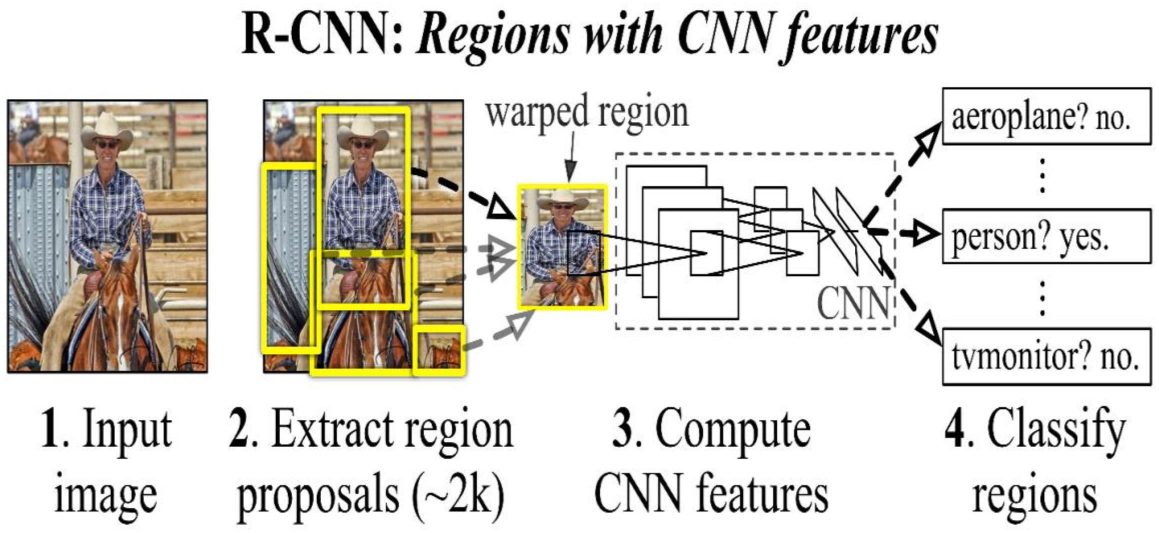


Figure 25 R-CNN

3. Object detection models

In our endeavor to achieve the best results for our project, we have trained three distinct models using a carefully curated dataset. Our primary objective was to explore a diverse range of model architectures, enabling us to leverage their unique characteristics and capabilities.

The first model, the EfficientDet D1 640x640, is a cutting-edge object detection model known for its efficiency and accuracy. By utilizing the compound scaling method and the EfficientNet backbone architecture, this model strikes a balance between computational efficiency and detection accuracy. With an input resolution of 640x640 pixels, it excels at identifying objects within images and localizing them with precision.

A. EfficientDet D1 640x640

EfficientDet is a family of object detection models that are known for their efficiency and high performance. The "D1" in EfficientDet D1 refers to the model's size variant, where D1 represents a specific configuration of the model architecture. The "640x640" refers to the input resolution of the model.

EfficientDet models are based on a compound scaling method that aims to balance the model's accuracy and efficiency. They incorporate the EfficientNet architecture as a backbone network, which is designed to optimize both accuracy and computational efficiency by scaling the depth, width, and resolution of the network.

The EfficientDet D1 variant specifically refers to a model that has been configured to strike a balance between accuracy and efficiency, with D1 representing a medium-sized variant within the EfficientDet family. The "640x640" input resolution indicates that the model expects input images to be of size 640 pixels in width and 640 pixels in height.

The choice of input resolution in object detection models like EfficientDet can impact both the accuracy and the computational efficiency of the model. Higher resolutions generally allow for better localization and detection of smaller objects but come with increased computational costs. Lower resolutions reduce computational requirements but may lead to reduced accuracy, particularly for smaller objects.

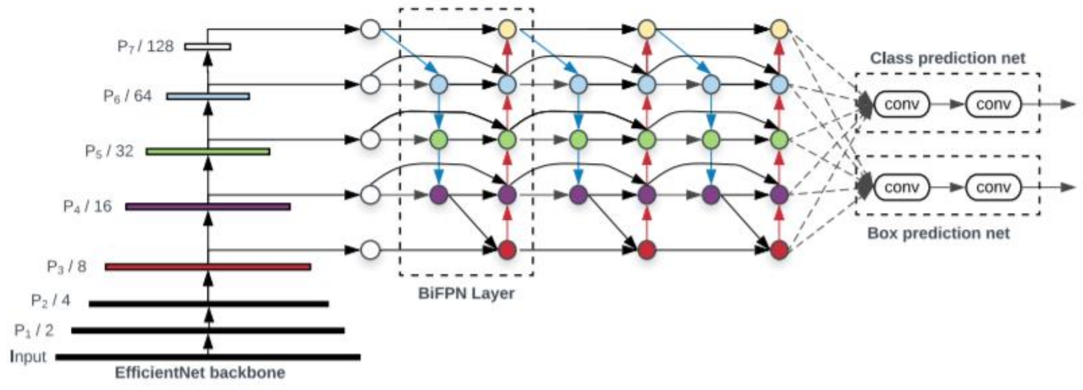


Figure 26 EfficientDet architecture

EfficientDet D1 640x640, therefore, refers to an EfficientDet model variant that has been specifically designed to achieve a balance between accuracy and efficiency, with an input resolution of 640x640 pixels. It provides a trade-off between model size, accuracy, and the ability to detect objects of varying sizes within images.

B. YOLOv4

One of the models we employed is YOLOv4 (You Only Look Once version 4), a renowned object detection framework. YOLOv4 is known for its exceptional speed and accuracy in real-time object detection tasks. By utilizing a single neural network to simultaneously predict bounding boxes and class probabilities.

YOLOv4 is an advanced object detection framework that has gained significant popularity and achieved state-of-the-art performance. It is renowned for its exceptional speed and accuracy in real-time object detection tasks, making it a powerful tool in computer vision applications.

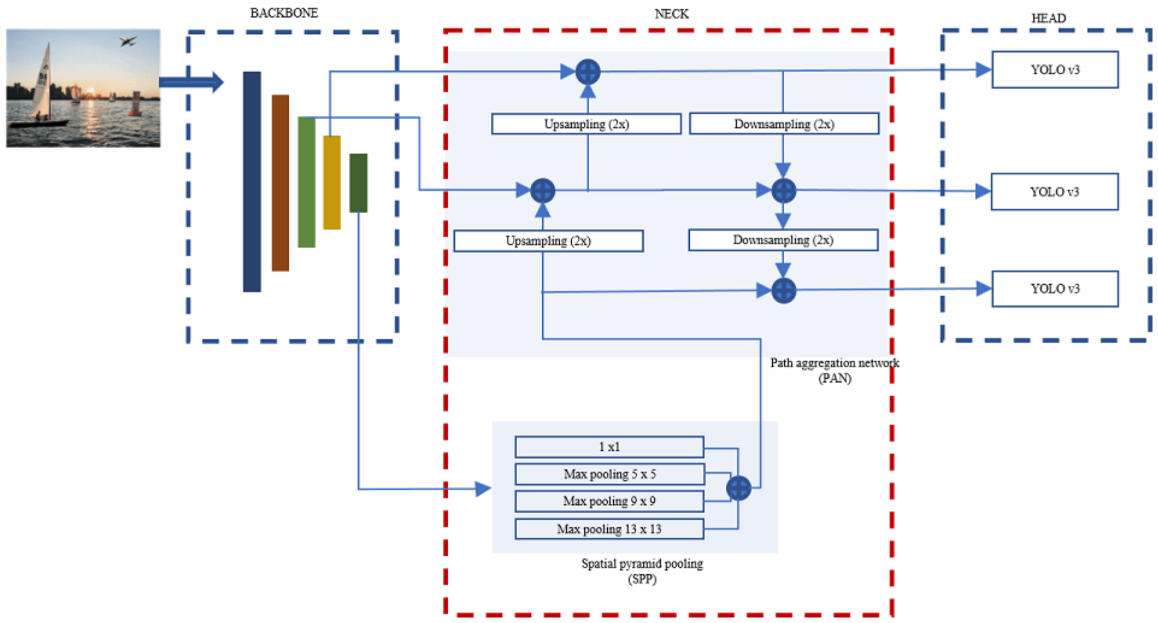


Figure 27 YOLOv4 architecture

YOLOv4 builds upon the success of its predecessors and introduces several enhancements and innovations to further improve object detection performance. The model combines deep neural networks and advanced techniques to achieve accurate and efficient object detection.

One key improvement in YOLOv4 is the integration of a modified backbone architecture called CSPDarknet53. This modified backbone enhances feature extraction by employing Cross-Stage Partial connections, reducing computational complexity while maintaining accuracy. This allows YOLOv4 to achieve better performance in detecting objects of varying sizes and scales.

To enhance the precision of object detection, YOLOv4 incorporates multiple detection heads at different scales. These detection heads extract features at different levels of granularity, allowing the model to detect objects of different sizes with greater accuracy. Additionally, YOLOv4 uses anchor boxes of various aspect ratios to further improve the detection of objects with diverse shapes.

YOLOv4 also introduces novel techniques to address the challenges of occlusions and small object detection. By employing the Mish activation function and advanced data augmentation strategies like mosaic data augmentation and CIOU loss, YOLOv4 improves the model's ability to handle complex scenarios and achieve more accurate object localization.

The third model we incorporated into our model portfolio is Faster R-CNN (Region Convolutional Neural Network) with a ResNet50 V1 backbone and an input resolution of 640x640 pixels. Faster R-CNN is a widely-used object detection framework known for its accuracy and robustness.

C. Faster R-CNN ResNet50 V1 640x640

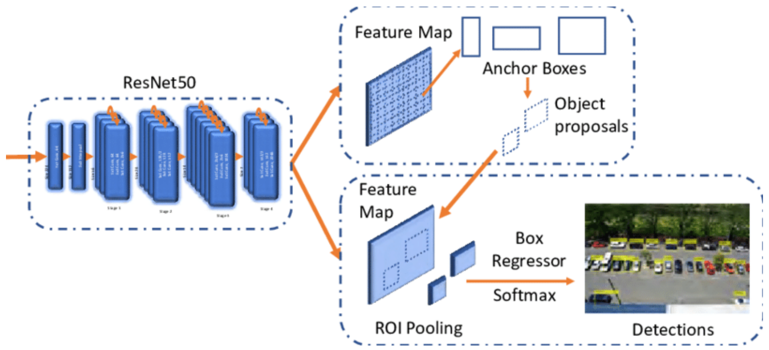


Figure 28 R-CNN ResNet 50 architecture

The Faster R-CNN architecture is composed of two main components: a region proposal network (RPN) and a subsequent network for object classification and bounding box regression. The RPN generates region proposals that are likely to contain objects, while the classification and regression network refines these proposals to accurately localize and classify objects within the image.

By utilizing the ResNet50 V1 backbone, the model benefits from the deep residual network's ability to extract hierarchical features from the input image. ResNet50 V1 consists of 50 convolutional layers and residual connections, allowing the model to capture both low-level and high-level visual features, leading to improved object detection performance.

With an input resolution of 640x640 pixels, the model can effectively handle images of larger sizes, enabling the detection and localization of objects with greater detail. This resolution choice strikes a balance between computational efficiency and the ability to capture fine-grained features within the image.

Faster R-CNN with ResNet50 V1 640x640 offers several advantages, including precise object localization, accurate object classification, and robustness against occlusions and background clutter. It excels in scenarios where both high accuracy and computational efficiency are required, making it a popular choice for a wide range of applications, such as autonomous driving, video surveillance, and object recognition.

By including Faster R-CNN with ResNet50 V1 640x640 in our model selection, we aim to leverage its exceptional object detection capabilities. The model's ability to accurately locate and classify objects within images will contribute to our comprehensive analysis and facilitate valuable insights from our dataset.

Our careful selection of diverse model architectures reflects our commitment to exhaustively exploring various methodologies and capitalizing on their respective strengths. By embracing this multifaceted approach, we aspire to attain the best possible results and unlock valuable insights from our dataset, ultimately advancing the goals of our project.

4. Object detection models Results

A. EfficientDet D1 640x640

The mAP results are relatively low but for a dataset of fewer than 3000 images

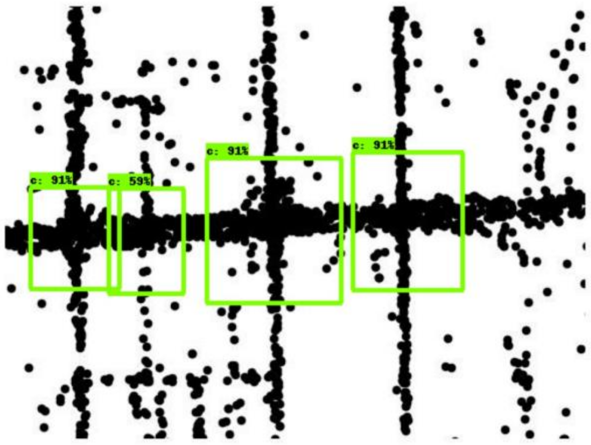
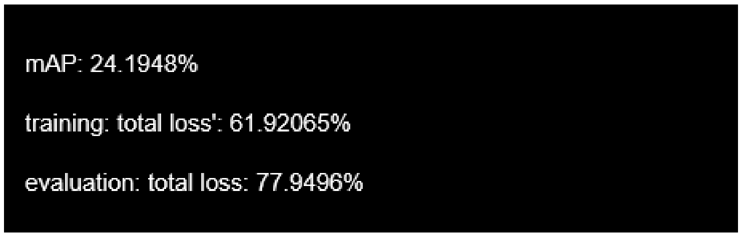


Figure 29 Example results of EfficientDet



mAP: 24.1948%

training: total loss': 61.92065%

evaluation: total loss: 77.9496%

B. Yolov4

mAP: 24%

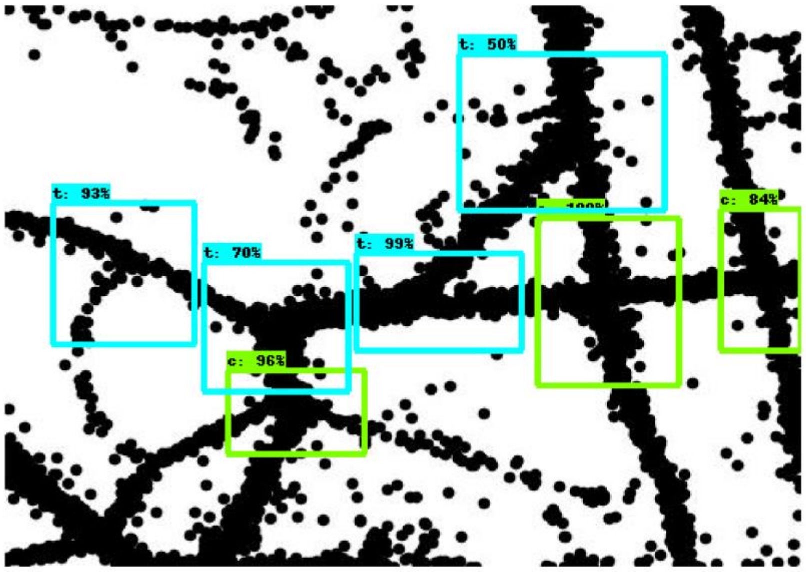
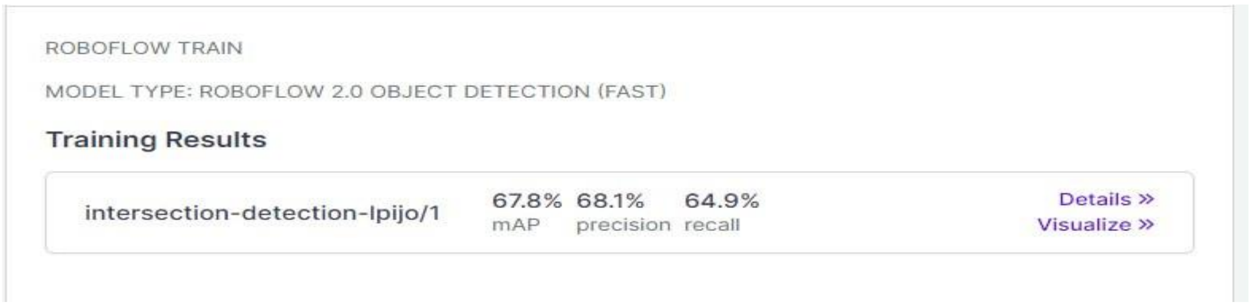


Figure 30 Example results of YOLOv4



ROBOFLOW TRAIN

MODEL TYPE: ROBOFLOW 2.0 OBJECT DETECTION (FAST)

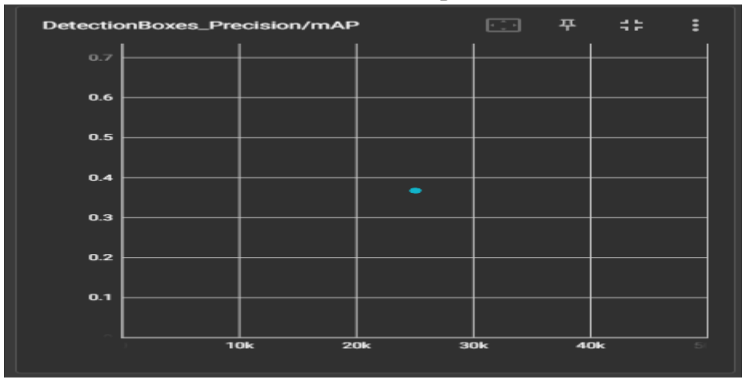
Training Results

intersection-detection-lpijo/1 67.8% mAP 68.1% precision 64.9% recall

C. Faster R-CNN ResNet50 V1 640x640

Mean average precision = 36%

Total loss for training = 0.08



Chapter 9: Code

1. simulated data

A. Required modules

```
#required modules
from itertools import count
import os
from tkinter import image_names
import numpy as np
from random import randrange, uniform
from matplotlib import pyplot as plt
import math
import matplotlib.image as mpimg
import random
```

B. Create round path

```

def create_round_path(self,source,destination,points,radius,road_size):
        L=[source,destination]
        if (source+destination)==4:
            #verical round intersection
            points=int(points/2)

 x,y =self.create_half_circle_1_3(radius,points,source,road_size)

        elif (source+destination)==2:
            #horizantal round intersection
            points=int(points/2)

 x,y =self.create_half_circle_0_2(radius,points,source,road_size)

        elif (source+destination)==5:
            #quarter circle between path 2,3
            points=int(points/4)

 x,y =self.create_quart_circle_2_3(radius,points,source,road_size)

        elif (source+destination)==1:
            #quarter circle between path 0,1
            points=int(points/4)

 x,y= self.create_quart_circle_0_1(radius,points,source,road_size)

        elif (source+destination)==3 and ( L==[0,3] or L[::-1]==[0,3]):
            #quarter circle between path 0,3
            points=int(points/4)

x,y= self.create_quart_circle_0_3(radius,points,source,road_size)

        elif (source+destination)==3 and ( L==[1,2] or L[::-1]==[1,2]):
            #quarter circle between path 1,2
            points=int(points/4)

 x,y= self.create_quart_circle_1_2(radius,points,source,road_size)
        return x,y
```

C. Create path

```
 Def
create_path(self,path_number,step,rang,radius,S_OR_D,road_size,is_round):
        if path_number==0:
            x,y=self.create_linear_path_0(step,rang,radius,S_OR_D,road_siz
e,is_round)
            return x,y
        elif path_number==1:
            x,y=self.create_linear_path_1(step,rang,radius,S_OR_D,road_siz
e,is_round)
            return x,y
        elif path_number==2:
            x,y=self.create_linear_path_2(step,rang,radius,S_OR_D,road_siz
e,is_round)
            return x,y
        elif path_number==3:
            x,y=self.create_linear_path_3(step,rang,radius,S_OR_D,road_siz
e,is_round)
            return x,y
```

D. Create OD matrix

```
        def create_od_matrix(self,raw_size):

        OD_MATRIX=[randrange(10000) for i in range(raw_size*raw_size)]
        if raw_size==3:
            #no return
            OD_MATRIX[0],OD_MATRIX[4],OD_MATRIX[8]=0,0,0
        elif raw_size==4:
            #no return
            OD_MATRIX[0],OD_MATRIX[5],OD_MATRIX[10],OD_MATRIX[15]=0,0,0,0
        elif raw_size==2:
            OD_MATRIX[0],OD_MATRIX[3]=0,0
        OD_MATRIX=np.array(
```

E. Sampling function

```
    def sampling_function(self,x,y,sampling_rate):
        #sampling the data so we don`t take whole data
        sampled_x=[]
        sampled_y=[]
        total_index=len(x)
        index_list=[i for i in range(total_index)]
        random_sampling=random.choices(index_list,k=int(total_index*sampli
ng_rate)  )
        for i in random_sampling:
            sampled_x.append(x[i])
            sampled_y.append(y[i])
        return sampled_x,sampled_y
```

F. Adding noise

```
    def
adding_noise(self,x,y,noise_size_x,noise_size_y,mu,sigma_x,sigma_y):


        x=np.array(x)
        y=np.array(y)
        noise_x = np.random.normal(mu,sigma_x, x.shape)
        noise_y = np.random.normal(mu,sigma_y, y.shape)
        new_x = x + noise_size_x * noise_x
        new_y= y + noise_size_y * noise_y
        new_x=new_x.tolist()
        new_y=new_y.tolist()
        return new_x,new_y
```

G.Create a whole trajectory

```
    def
total_path(self,source,destination,step,rang,radius,is_round,points,road_s
ize):
        if is_round==False:

            x_1,y_1=self.create_path(source,step,rang,radius,'source',road
_size,False)
            x_2,y_2=self.create_path(destination,step,rang,radius,'destina
tion',road_size,False)
            x=x_1+x_2
            y=y_1+y_2

        else:

            x_1,y_1=self.create_path(source,step,rang,radius,'source',road
_size,True)
            x_2,y_2=self.create_path(destination,step,rang,radius,'destina
tion',road_size,True)
            x_3,y_3=self.create_round_path(source,destination,points,radiu
s,road_size)
            x=x_1+x_2+x_3
            y=y_1+y_2+y_3
        return x,y
```

H. Cross intersection

```
    def
cross_intersection(self,number_of_trips,radius,rang,step,sampling_rate,noi
se_size_x,noise_size_y,mu,sigma_x,sigma_y,sampling_trip,road_size,fig_coun
t,path,x_min,x_max,y_min,y_max,image_check):
        raw_size=4
        radius=0
        is_round=False
        points=0
        X,Y=[],[]

        OD_MATRIX=self.create_od_matrix(raw_size)
        for i in range(int(number_of_trips*sampling_trip)):
            source,destination=self.source_destination_prob(OD_MATRIX,raw_
size)
            x,y=self.total_path(source,destination,step,rang,radius,is_rou
nd,points,road_size)
            x,y=self.sampling_function(x,y,sampling_rate)
            x,y=self.adding_noise(x,y,noise_size_x,noise_size_y,mu,sigma_x
,sigma_y)
            X+=x
            Y+=y
        self.plotting_data_with_saving(X,Y,fig_count,path,x_min,x_max,y_mi
n,y_max)
        if image_check==True:
            self.check_to_remove(path,fig_count)
```

I. T-intersection

```
    def
T_intersection(self,number_of_trips,radius,rang,step,sampling_rate,noise_s
ize_x,noise_size_y,mu,sigma_x,sigma_y,sampling_trip,road_size,fig_count,pa
th,x_min,x_max,y_min,y_max,fixed_shape,image_check):
        is_round=False
        radius=0
        points=0
        X,Y=[],[]
        if fixed_shape==True:
            raw_size=3
            OD_MATRIX=self.create_od_matrix(raw_size)
        else :
            raw_size=4
            OD_MATRIX=self.create_od_matrix(raw_size)
            canceled_raw=random.randint(0,3)
            for i in range(raw_size):
                OD_MATRIX[canceled_raw][i]=0
                OD_MATRIX[i][canceled_raw]=0
        for i in range(int(number_of_trips*sampling_trip)):
            source,destination=self.source_destination_prob(OD_MATRIX,raw_
size)
            x,y=self.total_path(source,destination,step,rang,radius,is_rou
nd,points,road_size)
            x,y=self.sampling_function(x,y,sampling_rate)
            x,y=self.adding_noise(x,y,noise_size_x,noise_size_y,mu,sigma_x
,sigma_y)
            X+=x
            Y+=y
        self.plotting_data_with_saving(X,Y,fig_count,path,x_min,x_max,y_mi
n,y_max)
        if image_check==True:
            self.check_to_remove(path,fig_count)
```

J. Round intersection

```
    def
round_intersection(self,number_of_trips,radius,rang,step,sampling_rate,noi
se_size_x,noise_size_y,mu,sigma_x,sigma_y,sampling_trip,road_size,fig_coun
t,path,x_min,x_max,y_min,y_max,points,image_check):
        is_round=True
        X,Y=[],[]
        raw_size=4
        OD_MATRIX=self.create_od_matrix(raw_size)
        for i in range(int(number_of_trips*sampling_trip)):
            source,destination=self.source_destination_prob(OD_MATRIX,raw_
size)
            x,y=self.total_path(source,destination,step,rang,radius,is_rou
nd,points,road_size)
            x,y=self.sampling_function(x,y,sampling_rate)
            x,y=self.adding_noise(x,y,noise_size_x,noise_size_y,mu,sigma_x
,sigma_y)
            X+=x
            Y+=y
        self.plotting_data_with_saving(X,Y,fig_count,path,x_min,x_max,y_mi
n,y_max)
        if image_check==True:
            self.check_to_remove(path,fig_count)
```

K. Non intersection

```
    def
non_intersection(self,number_of_trips,radius,rang,step,sampling_rate,noise
_size_x,noise_size_y,mu,sigma_x,sigma_y,sampling_trip,road_size,fig_count,
path,x_min,x_max,y_min,y_max,fixed_shape,with_nodes,image_check,strait_pro
b=0.5):
        is_round=False
        X,Y=[],[]
        points=0
        radius=0
        if with_nodes==True:
            strait=[[0,2],[1,3]]
            inter=[[0,1],[2,3],[0,3],[1,2]]
            total=[strait,inter]
            w_1=[strait_prob,1-strait_prob]
            w_2=[0.5,0.5]
            w_3=[0.25,0.25,0.25,0.25]
            type_inter=random.choices(total,weights=w_1)[0]
            if len(type_inter)==4:
                S_AND_D=random.choices(inter,weights=w_3)[0]
                source=S_AND_D[0]
                destination=S_AND_D[1]
            else:
                S_AND_D=random.choices(strait,weights=w_2)[0]
                source=S_AND_D[0]
                destination=S_AND_D[1]
        else:
            if fixed_shape==True:
                source=0
                destination=2
            else:
                strait=[[0,2],[1,3]]
                w_1=[0.5,0.5]
                S_AND_D=random.choices(strait,weights=w_1)[0]
                source=S_AND_D[0]
                destination=S_AND_D[1]


        for i in range(int(number_of_trips*sampling_trip)):

            x_1,y_1=self.total_path(source,destination,step,rang,radius,is
_round,points,road_size)
            x_2,y_2=self.total_path(destination,source,step,rang,radius,is
_round,points,road_size)
            x=x_1+x_2
            y=y_1+y_2
            x,y=self.sampling_function(x,y,sampling_rate)
            x,y=self.adding_noise(x,y,noise_size_x,noise_size_y,mu,sigma_x
,sigma_y)
            X+=x
            Y+=y
        self.plotting_data_with_saving(X,Y,fig_count,path,x_min,x_max,y_mi
n,y_max)
        if image_check==True:
            self.check_to_remove(path,fig_count)
    def check_to_remove(self,path,fig_count):
        img = mpimg.imread(path+'/'+str(fig_count)+'.jpg')
        imgplot = plt.imshow(img)
        plt.show()

        if input('you want to save it yes ,no ?  ')=='no':
            os.remove(path+'//'+str(fig_count)+'.jpg')
        plt.clf()
```

2. Cross testing

A. Required modules

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras
from sklearn.metrics import confusion_matrix,  ConfusionMatrixDisplay
import os
import shutil
```

B. Test model function

```
def
test_model(path_to_model,path_to_data,output_file,model_name,data_name):
  BATCH_SIZE = 60
  IMG_SIZE = (224, 224)
  model = tf.keras.models.load_model(path_to_model)
  val_data = tf.keras.utils.image_dataset_from_directory(path_to_data,
                                                          batch_size=BAT
CH_SIZE,
                                                          image_size=IMG
_SIZE,
                                                          shuffle=True)

  val_images =[]
  true_labels = []
  for image, label in val_data.unbatch():
      val_images.append(image)
      true_labels.append(label)
  val_images = np.array(val_images)
  true_labels = np.array(true_labels)

  loss, acc = model.evaluate(val_images, true_labels,verbose=0)

  file=open(output_file,'a')
  file.write('model name : {0} ,data_name : {1}
'.format(model_name,data_name))
  file.write("Accuracy: {:5.2f}% \n".format(100 * acc))
  file.close()
```

3. Kruskal test

A. Required modules

```
import scipy.stats as st
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

B. plot values of T-drive

```
## train on T-drive
##########################################################################
##
t_drive = np.array([89.8, 96.4, 93, 96.8])           # values of test on t
drive
cabspotting = np.array([90.5, 89.5, 91.75, 89.75])      # values of test
on cabspotting
kaggle = np.array([71.5, 75, 74, 74])           # values of test on kaggle
##########################################################################
##
plt.figure(figsize=(9,5))
sns.kdeplot(t_drive,shade=True,color='black')
sns.kdeplot(cabspotting,shade=True,color='red')
sns.kdeplot(kaggle,shade=True,color='Blue')
plt.legend(['test on T-drive','test on Cabspotting','test on ECML-PKDD
15'],fontsize=8)
plt.vlines(x=t_drive.mean(),ymin=0,ymax=0.09,color='black',linestyle='--')
plt.vlines(x=cabspotting.mean(),ymin=0,ymax=0.09,color='red',linestyle='--
')
plt.vlines(x=kaggle.mean(),ymin=0,ymax=0.09,color='Blue',linestyle='--')
plt.title('Train on T-drive')
plt.figtext(0.5, -0.03, "hypothesis of same distribution", ha="center",
fontsize=12, bbox={"facecolor":"white", "alpha":0.5, "pad":5})
_,p_1=st.kruskal(t_drive,cabspotting)
_,p_2=st.kruskal(t_drive,kaggle)
p_1_con="reject"
p_2_con="reject"
if p_1>0.05:
    p_1_con="fails to reject"
if p_2>0.05:
    p_2_con="fails to reject"
```

```
plt.figtext(0.1, -0.1,"- p-value for T-drive and Capsbotting : {:.3f},
{}".format(p_1, p_1_con),fontsize=12)
plt.figtext(0.1, -0.15,"- p-value for T-drive and ECMl-PKDD : {:.3f},
{}".format(p_2, p_2_con),fontsize=12)
plt.savefig("t_drive.jpg", bbox_inches="tight", format='jpg')
plt.show()
```

4. Feature extraction

A. Required modules

```
#required modules
import numpy as np
import pandas as pd
import os
import random
from matplotlib import pyplot as plt
from  sklearn.preprocessing import MinMaxScaler
import seaborn as sns
from datetime import datetime
from datetime import timedelta

#customized moduless filt
from ptrail.core.TrajectoryDF import PTRAILDataFrame
from ptrail.core.Datasets import Datasets
from ptrail.features.kinematic_features import KinematicFeatures as
kinematic
from ptrail.preprocessing.filters import Filters as filt
```

B. Feature extraction

```
start=0
end=400000
for j in range(start,end,100000):
  flag=False

  for i in range(j,j+100000):

      trip=df.iloc[i]
      #list of longitude and latiude  at original time stamp
      trip_location=trip['POLYLINE']
      value =eval(trip_location.split()[0])
      base_time=datetime.fromtimestamp(trip['TIMESTAMP'])
      time_list=[base_time+ timedelta(seconds=j*15) for j in
range(len(value))]
      tmp_dataframe=pd.DataFrame(value,columns=['longitude','latitude'])

      taxi_id_list=[trip['TAXI_ID'] for k in range(len(value))]
      tmp_dataframe['TIMEDATA']=time_list
      tmp_dataframe['TAXI_ID']=taxi_id_list
      tmp_dataframe=PTRAILDataFrame(data_set=tmp_dataframe,latitude='latit
ude',longitude='longitude',datetime='TIMEDATA',traj_id='TAXI_ID')
      tmp_dataframe=kinematic.create_acceleration_column(tmp_dataframe)
      tmp_dataframe=kinematic.create_bearing_column(tmp_dataframe)
      if flag==False:
        tmp_dataframe.to_csv('data_with_feature/kaggle_data_with_feature_{
}.csv'.format(j),index=False,header=True)
        flag=True
      else:
        tmp_dataframe.to_csv('data_with_feature/kaggle_data_with_feature_{
}.csv'.format(j),mode='a',index=False,header=False)
```

5. Feature Rasterization

A. Required Modules

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
```

B. Features to extract

```
# extract features
feature_density = False
feature_speed = True
feature_acceleration = False
feature_heading = False
```

C. Process of extraction

```
for inter_num in range(len(intersection_boundBox)):
    # extract intersection bound box and points
    min_longitude = intersection_boundBox['min_lon'][inter_num]
    max_longitude = intersection_boundBox['max_lon'][inter_num]
    min_latitude = intersection_boundBox['min_lat'][inter_num]
    max_latitude = intersection_boundBox['max_lat'][inter_num]
    intersection_points = 
taxi_trajectories.loc[(taxi_trajectories['longitude']<max_longitude)&
                                                (taxi_trajectories['longit
ude']>min_longitude)&
                                                (taxi_trajectories['latitu
de']<max_latitude)&
                                                (taxi_trajectories['latitu
de']>min_latitude)]
    # the grid dimensions[#cells = cells_columns_number * 
cells_row_number]
    cells_columns_number = 14
    cells_row_number = 14
    # get the steps values
    longitude_step = np.abs(max_longitude - min_longitude) / 
(cells_row_number*2)
    latitude_step = np.abs(max_latitude - min_latitude) / 
(cells_columns_number*2)
    cells_center_matrix = np.zeros((cells_row_number, 
cells_columns_number, 2))
    # computer the center points
    for i in range(cells_row_number):
        for j in range(cells_columns_number):
            cells_center_matrix[i,j,0] = max_longitude - (1+2*i) * 
longitude_step      # lonitude -> X0
            cells_center_matrix[i,j,1] = min_latitude + (1+2*j) * 
latitude_step        # latitude -> Y0
    # Grid
    density_Grid = np.zeros((cells_row_number, cells_columns_number))
    speed_Grid = np.zeros((cells_row_number, cells_columns_number))
    acceleration_Grid = np.zeros((cells_row_number, cells_columns_number))
    head_Grid = np.zeros((cells_row_number, cells_columns_number))
    # gaussain kernel
    def gaussain_kernel(x0, y0, xi, yi, sigma):
        constant_part= 1 / (np.sqrt(2 * np.pi * sigma))
        expon_part=np.exp(-1 * (np.power((xi-x0),2) + np.power((yi-y0),2))
/ (2 * np.power(sigma,2)))
        w = constant_part * expon_part
        return w
    # compute Grid features
    sigma = 0.0001
    for i in range(cells_row_number):
        for j in range(cells_columns_number):
            x_i = np.array(intersection_points['longitude'])
            y_i = np.array(intersection_points['latitude'])
            x_0 = np.array([cells_center_matrix[i,j,0]])
            y_0 = np.array([cells_center_matrix[i,j,1]])
            distance = np.sqrt(np.power((x_0 - x_i),2) + np.power((y_0 - 
y_i),2))
            x_i = x_i[distance<=(3*sigma)]
            y_i = y_i[distance<=(3*sigma)]
            weights = gaussain_kernel(x_0 , y_0, x_i, y_i, sigma)
            if(feature_density):
                density_Grid[i,j] = np.sum(weights)
            if(feature_speed):
                speed = np.array(intersection_points['speed'])
                speed = speed[distance<=(3*sigma)]
                speed_Grid[i,j] = np.dot(weights, speed)
            if(feature_acceleration):
                acceleration = 
np.array(intersection_points['acceleration'])
                acceleration = acceleration[distance<=(3*sigma)]
                acceleration_Grid[i,j] = np.dot(weights, acceleration)
            if(feature_heading):
                head = np.array(intersection_points['head'])
                head = head[distance<=(3*sigma)]
                head_Grid[i,j] = np.dot(weights, head)

    # Define image dimension
    image_dims = 224
    image = np.zeros((image_dims,image_dims))
    x_steps = image_dims / cells_columns_number
    y_steps = image_dims / cells_row_number
    feature_image = np.zeros((image_dims,image_dims,3))
    channel = 0
    image_name = ''
    # build image channels
    if(feature_density):
        for i in range(image_dims):
            for j in range(image_dims):
                image[i,j] = density_Grid[int(i/x_steps),int(j/y_steps)]
        image = image.T                  # to get the image without reverse
        image = np.flip(image, -1)       # or fliping
        image = np.flip(image, 0)
        scaler = MinMaxScaler()
        image_s = scaler.fit_transform(image)
        # this part because of the problem of 1.0000000000000002 and must 
be 0-1 for RGB
```

```
        for i in range(len(image_s)):
            for j in range(len(image_s[i])):
                if(image_s[i,j]>1):
                    image_s[i,j] = 1.0
        feature_image[:,:,channel] = image_s
        channel += 1
        image_name += 'D'
    if(feature_speed):
        for i in range(image_dims):
            for j in range(image_dims):
                image[i,j] = speed_Grid[int(i/x_steps),int(j/y_steps)]
        image = image.T                      # to get the image without reverse
        image = np.flip(image, -1)           # or fliping
        image = np.flip(image, 0)
        scaler = MinMaxScaler()
        image_s = scaler.fit_transform(image)
        # this part because of the problem of 1.0000000000000002 and must
be 0-1 for RGB
        for i in range(len(image_s)):
            for j in range(len(image_s[i])):
                if(image_s[i,j]>1):
                    image_s[i,j] = 1.0
        feature_image[:,:,channel] = image_s
        channel += 1
        image_name += 'S'
    if(feature_acceleration):
        for i in range(image_dims):
            for j in range(image_dims):
                image[i,j] =
acceleration_Grid[int(i/x_steps),int(j/y_steps)]
        image = image.T                      # to get the image without reverse
        image = np.flip(image, -1)           # or fliping
        image = np.flip(image, 0)
        scaler = MinMaxScaler()
        image_s = scaler.fit_transform(image)
        # this part because of the problem of 1.0000000000000002 and must
be 0-1 for RGB
        for i in range(len(image_s)):
            for j in range(len(image_s[i])):
                if(image_s[i,j]>1):
                    image_s[i,j] = 1.0
        feature_image[:,:,channel] = image_s
        channel += 1
        image_name += 'A'
    if(feature_heading):
        for i in range(image_dims):
            for j in range(image_dims):
                image[i,j] = head_Grid[int(i/x_steps),int(j/y_steps)]
        image = image.T                         # to get the image without reverse
        image = np.flip(image, -1)              # or fliping
        image = np.flip(image, 0)
        scaler = MinMaxScaler()
        image_s = scaler.fit_transform(image)
        # this part because of the problem of 1.0000000000000002 and must
be 0-1 for RGB
        for i in range(len(image_s)):
            for j in range(len(image_s[i])):
                if(image_s[i,j]>1):
                    image_s[i,j] = 1.0
        feature_image[:,:,channel] = image_s
        channel += 1
        image_name += 'H'
    # extract image
    image_name = str(intersection_boundBox['label'][inter_num]) +
str(inter_num) + '_' + image_name + '.jpg'
    plt.imsave(image_name,feature_image,format='jpg')
```

6. Ensemble bootstrapping heuristic

L. Required Modules

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras
from sklearn.metrics import accuracy_score
import random
```

M. needed functions

```
def load_feature_data(feature_name):
    BATCH_SIZE = 32
    IMG_SIZE = (224, 224)
    feature =
tf.keras.utils.image_dataset_from_directory("feature_images_30x30/{0}/test
".format(feature_name),
                                                                      batch_size
=BATCH_SIZE,
                                                                      image_size
=IMG_SIZE,
                                                                      shuffle=Fa
lse)
    return feature

def load_network(network_name,feature_model):
    if (network_name=="NasNet"):
        model =
tf.keras.models.load_model('{0}/{1}.h5'.format(network_name,feature_model)
, compile=False)
    else:
        model =
tf.keras.models.load_model('{0}/{1}.h5'.format(network_name,feature_model)
)
    return model

def evaluation_prob(dataset,model):
    val_images =[]
    true_labels = []
    for image, label in dataset.unbatch():
        val_images.append(image)
        true_labels.append(label)
    val_images = np.array(val_images)
    true_labels_arr = np.array(true_labels)
    predictions_prob = model.predict(val_images)
    return predictions_prob,true_labels_arr

def calculate_acc(predictions,true_labels):
    predict_labels = predictions.argmax(axis=1)
    return accuracy_score(true_labels,predict_labels)
```

N. Needed Arrays

```
features =
['D','S','A','H','DS','DA','DH','SA','SH','AH','DSA','DSH','DAH','SAH']
networks = ['MobileNet-','EfficientNet-','ResNet-','NasNet-','DenseNet-']
datasets = {}
MobileNet_models = {}
EfficientNet_models = {}
ResNet_models = {}
NasNet_models = {}
```

```
DenseNet_models = {}
margin_accuracy_vector = {}
```

O. Loading Features and Networks

```
def load_feature_data_network(feature):
    datasets[feature] = load_feature_data(feature)
    MobileNet_models[feature] = load_network("MobileNet",feature)
    EfficientNet_models[feature] = load_network("EfficientNet",feature)
    ResNet_models[feature] = load_network("ResNet",feature)
    NasNet_models[feature] = load_network("NasNet",feature)
    DenseNet_models[feature] = load_network("DenseNet",feature)
```

P. Initialize all models' vectors

```
# load all networks and data
for i in features:
    load_feature_data_network(i)
# initalize all models vector
for i in networks:
    for j in features:
        # compute the original accuracy of every model
        if(i == 'MobileNet-'):
            prob,true_label =
evaluation_prob(datasets[j],MobileNet_models[j])
        elif(i=='EfficientNet-'):
            prob,true_label =
evaluation_prob(datasets[j],EfficientNet_models[j])
        elif(i == 'ResNet-'):
            prob,true_label =
evaluation_prob(datasets[j],ResNet_models[j])
        elif(i=='NasNet-'):
            prob,true_label =
evaluation_prob(datasets[j],NasNet_models[j])
        elif(i=='DenseNet-'):
            prob,true_label =
evaluation_prob(datasets[j],DenseNet_models[j])

        accuracy = calculate_acc(prob,true_label)
        margin_accuracy_vector[i+j] = [accuracy,1]          # [accuracy ,
count]
```

Q. chose models and add accuracy

```
# the number of time run the random number of models
num_of_trail = 100
# all models names
models_names = margin_accuracy_vector.keys()
max_accur = 0
for i in range(0,num_of_trail):
    num_of_models =random.randint(2,10)
    # get random model
    random_models= random.sample(list(models_names),num_of_models)
    # accuracy of all models
    probability_of_all = 0
    for i in random_models:
        model_feature = i.split('-')        # split the name to the
network and feature
        f = model_feature[1]
        if(model_feature[0] == 'MobileNet'):
            prob,true_label =
evaluation_prob(datasets[f],MobileNet_models[f])
        elif(model_feature[0]=='EfficientNet'):
            prob,true_label =
evaluation_prob(datasets[f],EfficientNet_models[f])
        elif(model_feature[0] == 'ResNet'):
            prob,true_label =
evaluation_prob(datasets[f],ResNet_models[f])
        elif(model_feature[0]=='NasNet'):
            prob,true_label =
evaluation_prob(datasets[f],NasNet_models[f])
        elif(model_feature[0]=='DenseNet'):
            prob,true_label =
evaluation_prob(datasets[f],DenseNet_models[f])
        probability_of_all += np.log(prob)
    accuracy = calculate_acc(probability_of_all,true_label)
    if(accuracy>max_accur):
        max_combination = random_models
        max_accur = accuracy
    # add accuracy to the randome models and increase counter
    for i in random_models:
        margin_accuracy_vector[i][0] += accuracy
        margin_accuracy_vector[i][1] += 1
```

R. Calculate marginal accuracy

```
models_margin_accuracy = {}
models_names = margin_accuracy_vector.keys()
for i in models_names:
    models_margin_accuracy[i] = margin_accuracy_vector[i][0] /
margin_accuracy_vector[i][1]
```

S.Calculate accuracy for each model

```
combinations_dict = {}
# sort models by margin accuracy value
ranked_models = sorted(models_margin_accuracy.items(), key=lambda
x:x[1],reverse=True)
# perform step wise
for j in range(1,71):
    step_wise_models = []
    for i in range(0,j):
        step_wise_models.append(ranked_models[i][0])
    probability_of_all = 0
    for model in step_wise_models:
        model_feature = model.split('-')        # split the name to the
network and feature
        f = model_feature[1]
        if(model_feature[0] == 'MobileNet'):
            prob,true_label =
evaluation_prob(datasets[f],MobileNet_models[f])
        elif(model_feature[0]=='EfficientNet'):
            prob,true_label =
evaluation_prob(datasets[f],EfficientNet_models[f])
        elif(model_feature[0] == 'ResNet'):
            prob,true_label =
evaluation_prob(datasets[f],ResNet_models[f])
        elif(model_feature[0]=='NasNet'):
            prob,true_label =
evaluation_prob(datasets[f],NasNet_models[f])
        elif(model_feature[0]=='DenseNet'):
            prob,true_label =
evaluation_prob(datasets[f],DenseNet_models[f])
        probability_of_all += np.log(prob)
    accuracy = calculate_acc(probability_of_all,true_label)
    combinations_dict[str(j)] = [step_wise_models,accuracy]
```

Chapter 10: Conclusion

The problem of intersection classification using trajectory data is tackled in this paper. Five CNNs are used: (NasNet, ResNet, MobileNet, EfficientNet, and DenseNet), using three different datasets: T-drive, cab-spotting, and ECML PKDD 15.

First, a dataset of images is created using longitude and latitude data from the trajectory data. Two sets of images are constructed: the first set of images is formed from simulated data, while the other set of images is formed from real data. The same 10,000 simulated photos are used to train all models which are then tested on 500 photos of real data from the dataset. The achieved F1 scores range from 80% to 94%. The best scoring model in this task is the DenseNet with F1 scores of (.96 - .96 - .93) on the three datasets. The lowest accuracy comes from the ECML PKDD 15 dataset which is likely due to the large amount of noise in it, unlike the simulated data.

To test how well the trained models generalize, the same models trained on the three datasets are tested on the other two datasets that are not used in training. The Kruskal test with a p-value of .05 is used to visualize the outputs. The models that generalize well on the other datasets are the models trained on cab-spotting and ECML PKDD 15, whereas the models trained on T-drive do not generalize as well.

Later, the PTRAIL library is used to extract features from the raw trajectory data. The following features are chosen: density, speed, acceleration, and heading. A total of 14 combinations of the selected features are used, 1000 images of each combination are created, 900 images for training, and the remaining 100 images for testing. Two sizes of grids are experimented with; 20x20 and 30x30. With the 20x20 grid, the achieved accuracy ranges between 39 – 85%, while the 30x30 grid significantly improved the results with an accuracy range of 55 – 90%.

Finally, the 70 models that have been trained on the various combinations are ensembled based on their accuracies. The achieved accuracy levels range between 84 –

91% using only 900 photos for training compared to the use of 10,000 photos from the scatter plots. Results with the scatter plot range from 80 – 93% accuracy, and Results with the features range from 50 – 90 % accuracy without the ensemble and 84 % - 91% with the ensemble.

For the object detection the initial mAP results were: 24.19, 24, and 36 % for EfficientDet D1, YOLOV4, fater R-CNN resnet50 V1 respectively. Which is relatively low but the dataset consists of less than 3000 photos, so this is a good starting point. As we use it for detecting were intersections are then we use classification to make sure which type the intersection is.

Chapter 11: References

[1] O. Ronneberger, P. Fischer, and T. Brox, "U-Net: Convolutional Networks for Biomedical Image Segmentation," ArXiv, vol. abs/1505.04597, 2015.

[2] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," arXiv preprint arXiv:1409.1556, 2014.

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in Proceedings of the IEEE conference on computer vision and pattern recognition, 2016, pp. 770-778.

[4] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," 2009.

[5] F. H. W. Administration. "Intersection Safety."

https://safety.fhwa.dot.gov/intersection/about/ (accessed.

[6] V. Mnih and G. E. Hinton, "Learning to detect roads in high-resolution aerial images," in European conference on computer vision, 2010: Springer, pp. 210-223.

[7] L. Zhou, C. Zhang, and M. Wu, "D-LinkNet: LinkNet with Pretrained Encoder and Dilated Convolution for High Resolution Satellite Imagery Road Extraction," 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), pp. 192-1924, 2018.

[8] Y. Li, L. Xiang, C. Zhang, and H. Wu, "Fusing Taxi Trajectories and RS Images to Build Road Map via DCNN," IEEE Access, vol. 7, pp. 161487-161498, 2019.

[9] L. Cao and J. Krumm, "From GPS traces to a routable road map," Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, 2009.

[10] B. Niehöfer, R. Burda, C. Wietfeld, F. Bauer, and O. Lueert, "GPS Community Map Generation for Enhanced Routing Methods Based on Trace-Collection by Mobile Phones," 2009 First International Conference on Advances in Satellite and Space Communications, pp. 156-161, 2009.

[11] J. Qiu and R. Wang, "Road Map Inference: A Segmentation and Grouping Framework," ISPRS Int. J. Geo Inf., vol. 5, p. 130, 2016.

[12] S. Edelkamp and S. Schrödl, "Route Planning and Map Inference with Global Positioning Traces," in Computer Science in Perspective, 2003.

[13] J. J. Davies, A. R. Beresford, and A. Hopper, "Scalable, Distributed, Real-Time Map Generation," IEEE Pervasive Computing, vol. 5, pp. 47-54, 2006.

[14] J. Biagioni and J. Eriksson, "Inferring road maps from global positioning system traces: Survey and comparative evaluation," Transportation research record, vol. 2291, no. 1, pp. 61-71, 2012.

[15] Z. Zhang, Q. Liu, and Y. Wang, "Road Extraction by Deep Residual U-Net," IEEE Geoscience and Remote Sensing Letters, vol. 15, pp. 749-753, 2018.

[16] A. Chaurasia and E. Culurciello, "LinkNet: Exploiting encoder representations for efficient semantic segmentation," 2017 IEEE Visual Communications and Image Processing (VCIP), pp. 1-4, 2017.

[17] Y. Guo, B.-j. Li, Z. Lu, and J. Zhou, "A novel method for road network mining from floating car data," Geo-spatial Information Science, vol. 25, pp. 197 - 211, 2022.

[18] C. Nguyen, T. Dinh, V.-H. Nguyen, N. Tran, and A. Le, "Histogram-based Feature Extraction for GPS Trajectory Clustering," EAI Endorsed Transactions on Industrial Networks and Intelligent Systems, vol. 7, no. 22, 2020.

[19] L. Zhang, G. Zhang, Z. Liang, and E. F. Ozioko, "Multi-features taxi destination prediction with frequency domain processing," PloS one, vol. 13, no. 3, p. e0194629, 2018.

[20] M. Elhenawy, H. I. Ashqar, M. Masoud, M. H. Almannaa, A. Rakotonirainy, and H. A. Rakha, "Deep transfer learning for vulnerable road users detection using smartphone sensors data," Remote Sensing, vol. 12, no. 21, p. 3508, 2020.

[21] M. Daley, M. Elhenawy, M. Masoud, S. Glaser, and A. Rakotonirainy, "Detecting road user mode of

transportation using deep learning to enhance VRU safety in the C-ITS environment," in Proceedings of the 2021 Australasian Road Safety Conference, 2021: Australasian College of Road Safety (ACRS), pp. 450-452.

[22] S. Ruan et al., "Learning to generate maps from trajectories," in Proceedings of the AAAI conference on artificial intelligence, 2020, vol. 34, no. 01, pp. 890-897.

[23] A. Prabowo, P. Koniusz, W. Shao, and F. D. Salim, "Coltrane: Convolutional trajectory network for deep map inference," in Proceedings of the 6th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation, 2019, pp. 21-30.

[24] E. Eftelioglu, R. Garg, V. Kango, C. Gohil, and A. R. Chowdhury, "RING-Net: road inference from GPS trajectories using a deep segmentation network," in Proceedings of the 10th ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data, 2022, pp. 17-26.

[25] A. Howard et al., "Searching for mobilenetv3," in Proceedings of the IEEE/CVF international conference on computer vision, 2019, pp. 1314-1324.

[26] M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in International conference on machine learning, 2019: PMLR, pp. 6105-6114.

[27] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in Proceedings of the IEEE conference on computer vision and pattern recognition, 2018, pp. 8697-8710.

[28] G. Huang, Z. Liu, K. Weinberger, and L. van der Maaten, "Densely connected convolutional networks. CVPR 2017. arXiv 2016," arXiv preprint arXiv:1608.06993.

[29] J. Yuan, Y. Zheng, X. Xie, and G. Sun, "Driving with knowledge from the physical world," in Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining, 2011, pp. 316-324.

[30] J. Yuan et al., "T-drive: driving directions based on taxi trajectories," in Proceedings of the 18th SIGSPATIAL International conference on advances in geographic information systems, 2010, pp. 99-108.

[31] M. Piorkowski, N. Sarafijanovic-Djukic, and M. Grossglauser, "CRAWDAD data set epfl/mobility (v. 2009-02-24)," ed, 2009.

[32] L. Moreira-Matias, J. Gama, M. Ferreira, J. Mendes-Moreira, and L. Damas, "Predicting taxi–passenger demand using streaming data," IEEE Transactions on Intelligent Transportation Systems, vol. 14, no. 3, pp. 1393-1402, 2013.

[33] Y. Haranwala. "PTRAIL library." https://github.com/YakshHaranwala/PTRAIL (accessed.