VILNIUS UNIVERSITY
FACULTY OF MATHEMATICS AND INFORMATICS
MODELLING AND DATA ANALYSIS MASTER'S STUDY PROGRAMME

Master's thesis

Spatial Time Series Prediction using Bayesian Network Models

Erdvinių laiko eilučių modeliavimas naudojant Bajeso tinklų metodus

Justina Gertaitė

Supervisor: doc., dr. Jurgita Markevičiūtė

VILNIUS, 2022

Spatial Time Series Prediction using Bayesian Network Models

Abstract

This Master's thesis is based on book 'Enhanced Bayesian Network Models for Spatial Time Series Prediction' [1]. Methods written in the mentioned book as Bayesian Network with Residual Correction Mechanism, Spatial Bayesian Network and New Fuzzy Bayesian Networks were defined. DBSCAN clustering algorithm was integrated into Spatial Bayesian Network to choose locations. Wild bootstrap was applied to improve predictions. Five different Bayesian Networks were compared on spatial time series data set, where precipitation and weather events of Unites States of America were predicted. The proposed models could help insurers or skiing resort owners to evaluate prices, insurance reserves or season length of skiing resorts. However, mentioned models should be applied on a different data set or longer time series as predictions of thesis are very weak.

Keywords: Bayesian Network, spatial time series, Bayesian Network with residual correction mechanism, spatial Bayesian Network, new fuzzy Bayesian Network, DBSCAN, wild bootstrap.

Erdvinių laiko eilučių modeliavimas naudojant Bajeso tinklų metodus

Santrauka

Šis magistro baigiamasis darbas parašytas remiantis knyga 'Patobulinti Bajeso tinklų metodai erdvinėms laiko eilutėms' [1]. Metodai aprašyti minėtoje knygoje, kaip Bajeso tinklų metodas su paklaidų korekcijos mechanizmu, erdvinis bajeso tinklų metodas, nauji neapibrėžti Bajeso tinklų metodai yra apžvelgiami baigiamajame darbe. DBSCAN klasterizavimo algoritmas buvo integruojamas į erdvinį Bajeso tinklų metodą, kad pasirinkti lokacijas. Buvo pritaikytas savirankos metodas pagerinti prognozėms. Penki skirtingi Bajeso tinklų metodai buvo palyginti ant erdvinių laiko eilučių duomenų rinkinio, kur Jungtinių Amerikos valstijų krituliai ir orų reškiniai yra prognozuojami. Siūlomi modeliai gali padėti draudikams ar slidinėjimo kurortų savininkams įvertinti kainas, draudimo rezervus ar slidinėjimo sezono ilgį. Tačiau, minėti modeliai turėtų būti pritaikyti ant skirtingo duomenų rinkinio ar ilgesnės laiko eilutės, nes baigiamojo darbo prognozės labai silpnos.

Raktiniai žodžiai: Bajeso tinklų metodas, erdvinės laiko eilutės, Bajeso tinklų metodas su paklaidų korekcijos mechanizmu, erdvinis bajeso tinklų metodas, naujas neapibrėžtas Bajeso tinklų metodas, DBSCAN, savirankos metodas.

List of Abbreviations

BN	Bayesian Network
BNRC	Bayesian Network with Residual Correction Mechanism
SpaBN	Spatial Bayesian Network
FBN	Fuzzy Bayesian Network
NFBN	New Fuzzy Bayesian Network
FBNRC	Fuzzy Bayesian Network with Residual Correction Mechanism
SpaFBN	Fuzzy Spatial Bayesian Network
AQI	Air Quality Index
WHO	World Health Organization
NRMSD	Normalized Root Squared Error
MAE	Mean Absolute Error
MAPE	Mean Absolute Percentage Error
R^2	Coefficient of Determination
RMSE	Root Mean Squared Error
NSE	Nash-Sutcliffe Model Efficiency
Dv	Mean Percent Deviation
SEP	Standard Error of Prediction
DAG	Directed Acyclic Graph
CDG	Causal Dependency Graph
SW	Spatial Weight
SD	Spatial Distance
DBSCAN	Density Based Spatial Clustering of Applications with Noise
MP	Minimum Number of Points
USA	United States of America

Contents

Introduction

In 2020 Das et al. [1] released a book, where enhanced Bayesian Networks (BNs) were proposed, which perform better than standard Bayesian Network (BN). In this paper four of released BNs were defined: Bayesian Network with Residual Correction Mechanism (BNRC), Spatial Bayesian Network (SpaBN), Fuzzy Bayesian Network with Residual Correction Mechanism (FBNRC) and Fuzzy Spatial Bayesian Network (SpaFBN). In order to improve SpaBN, DBSCAN clustering algorithm was integrated to choose how many locations should be included to calculate spatial weight. Wild bootstrap resampling was applied to improve predictions. All five methods are compared on climatological data set.

One of the biggest issues in the world today is climate change. Main effects of climate change are rise in temperature, more severe storms, increased drought, species extinction etc. Impact of climate change, especially changing weather conditions throughout the year, affects many industries such as agriculture, skiing or insurance. Prediction of how frequent storms will be in the future or prediction of how much snow and how strong could help mentioned industries to predict the future. For example, for insurance companies knowing next year storm probability could help to calculate reserves of property insurance more precisely. Moreover, in the locations where storm probability is increasing, usually near the seas, prices of insurance should increase as well.

The United States Environmental Protection Agency established the Air Quality Index (AQI), which evaluates air pollution depending on five most common air pollutants. These air pollutants, which are regulated by Clean Air Act [15], are ground-level ozone, particle pollution (also known as particulate matter, including PM2.5 and PM10), carbon monoxide, sulfur dioxide and nitrogen dioxide. Mentioned pollutants cause climate change. It means weather events like snow, storm or fog are dependent on AQI. If following AQI and historical weather events, then it is possible to calculate probability of future weather events more precisely.

The main aim of this work is to define a BN to estimate probability of precipitation and weather events and compare which one of five Bayesian Networks works best on spatial time series data. The proposed model could help insurers or skiing resort owners to evaluate prices, insurance reserves or season length of skiing resorts.

The paper is organized as follows. In section 1, scientific papers related to climate change and Bayesian networks were reviewed. In section 2, Bayesian networks and enhanced BNs are defined. In section 3, the data set is described and results are discussed. In the last section, some conclusions are carried out highlighting the opportunity to improve the Bayesian Network predictions.

1 Literature review

In one of the largest and most-cited research publishers 'Frontiers in Public Health' in 2020 Manisalidis et al. made a review about how air pollutants affect not only health but also the environment [5]. And the research was made not only about the biotic environment, which is various micro-organisms, but also about abiotic, such as the hydrosphere or atmosphere. It is well known and discussed worldwide that climate change and air pollutants are closely related. Groundwater, soil, and air can be significantly affected by air pollution in the future. The World Health Organization (WHO) holds ground-level ozone, particle pollution, also known as particulate matter, carbon monoxide, sulfur dioxide, nitrogen dioxide and lead as major air pollutants. However, the Air Quality Index follows almost all of them, except lead. In the review, it is written how each air pollutant affects health and the environment. For example, carbon monoxide affects greenhouse gases, which leads to increased temperatures and extreme weather conditions.

BNs have been applied to solve problems across many fields, including environment and climatology. Particularly, BNs have found increasing application to the environment in recent days as protecting and restoring nature is one of the world's biggest crises. Ropero et al. (2019) use BNs to evaluate how climate change would affect olive crops [6]. In this paper the author compared regression and classification tasks based on BN with traditional methodologies as linear regression, decision trees or neural networks. As a conclusion, BNs, linear regression and decision trees were possible to interpret compared to logistic regression and neural networks. In BN models, the relationship between features and output variables could be explained clearly. BN models, like linear and logistic regression models, return not only answer value but also the distribution of values. Moreover, BN models could be applied to regression and for classification problems. However, results in the paper showed that BN applied to regression tasks gave smaller errors than applied to classification problems. Aderhold et al. (2012) proposed an enhanced Bayesian model, where Bayesian piecewise linear regression is combined with multiple changepoint processes [8]. The model was created to understand resulting interaction networks in ecosystems in a changing environment and how changing the environment affects biodiversity. In summary, from observed species distributions, the model creates species interaction networks. To solve the ecology problem, the authors enriched the model by a 2-dimensional changepoint process, which corresponds to a richer latent variable structure that allows modelling unobserved effects with smooth geographical variation. In species abundance data in conditions when predation is strong, the proposed model showed similar results compared to L1-regularized sparse regression (LASSO) and regression based on Bayes without changepoints. In conditions when predation is weak, the proposed model outperformed other models. The proposed model could be used as well as a tool for hypothesis generation. Another example of using the Bayesian network is to evaluate flood risk for citizens [7]. In the paper meteorological data from different stations in the Zurich area was followed, so as a result, spatial time series data was forecasted. Spatial time series data or geo-referenced time series data is a dominant category of spatio-temporal data. In spatial time series, data location is fixed and variables change in time [1]. In this case, spatial time series data is precipitation intensity in different districts in the Sihl River valley in Switzerland. So, flood risk to citizens of the Greater Zurich area was modelled using a spatially explicit Bayesian network model adjusted by expert opinion. So, with the BN model it is possible to combine quantitative data, meteorological and location data, with expert judgement or local knowledge.

Volume of spatial time series data is increasing fast nowadays due to the efficiency in collecting data. As a reason, in this paper, BN applied on spatial time series data will be used. Bayesian statistics are useful statistical tools for analysis of dependent structures and hidden patterns through prior knowledge. However, classification and regression models in machine learning as decision trees or Artificial neural networks are more popular and more explored compared to spatial Bayesian networks [4]. Nonetheless, it is better to develop data mining algorithms that have less complexity. Computational complexity could be described as a function that defines the efficiency of an algorithm to process a given amount of data [1]. Enhanced Bayesian network models have a more clear path from inputs to outputs compared to Artificial Intelligence algorithms.

Das, M. et al. (2020), in their book, made a unified view on enhanced models of Bayesian Networks in modelling spatial time series data [1]. One of the major issues is proper modelling of spatio-temporal inter-relationships among variables. One of the Bayesian Network methods that will be used in the paper is Bayesian Network enhanced with residual correction mechanism (BNRC) [1]. This method helps when there is unavailability of influencing variables in spatial time series data. As not always are known all influencing variables and not always all influencing variables are in the data. In order to solve this problem the authors proposed to enhance the hybrid BN model with a residual correction mechanism. The proposed model could be applied to any discrete spatio-temporal prediction. The model was used to predict climatological time series data in one of the papers [2] and in another hydrological time series data [3]. In the first paper, climatological predictions are made for two different locations, which belong to different climate zones. Predictions are made concerning three variables: temperature, humidity and precipitation. In this paper the accuracy of the model was measured using statistical metrics as Normalized Root Squared Error (NRMSD), Mean Absolute Error (MAE), Mean Absolute Percentage Error (MAPE) or the coefficient of determination – R^2. Results showed better spatial time series prediction than using several benchmark or state-of-the-art prediction techniques or standard BN. Compared with standard BN prediction, the overall average of improvement in NRMSD, MAE and MAPE of BNRC was 7,5%, 7% and 24%, respectively. In the hydrological time series data paper [3], the problem is to predict the water level of a reservoir. However, the historical daily water level data set does not have all influencing variables. To solve this problem also the BNRC model was chosen. The effectiveness of the BNRC prediction was evaluated by NRMSD, Nash-Sutcliffe model efficiency (NSE), mean percent deviation (Dv), standard error of prediction (SEP) and R^2. In this study BNRC-based prediction model was found to be more efficient than the standard BN and overall percentage of improvement of statistical measures was more than 70%. It means that the BNRC model compensates for unknown variables in the causal dependency graph. Another method that authors proposed to solve the problem of spatio-temporal inter-relationship among variables is Spatial Bayesian network (SpaBN) [1]. The problem could be fixed by including spatial information of the domain variables of scope. This method authors applied to the same climatological and hydrological problems mentioned before [2, 3]. Compared with standard BN temperature prediction, the overall average of improvement of SpaBN was 20,5% in the climatological problem [2]. In hydrological problem study, the SpaBN-based prediction model was found to be more efficient than the standard BN and overall percentage of improvement of statistical measures was more than 45%. So, involving the same influencing variables from other locations could improve model prediction significantly.

Due to lack of data and knowledge about the problem, mentioned models could be improved by fuzzy methods. Fuzzy Bayesian networks (FBN) are generalizations of BN, where the network consists of variables in fuzzy states. Fuzzy probability theory is based on fuzzy set membership. It helps to handle human generated imprecision present in data. FBNs are useful when discretizing continuous spatial time series data in discrete BN problems. There are many methods of Bayesian Networks with incorporated fuzzy logic [1]. One of the newest papers involving fuzzy logic uses triangular membership function to qualify the data in order to find optimal location for renewable energy sources and the assessment of energy scenarios. Too many linguistic variables make it challenging for decision makers to recognize differences between variables and too few causes inaccuracy. The best number is choosing from three to seven variables. In the paper five linguistic variables are chosen. To enhance the decision making process in a fuzzy process, experts' knowledge could be applied as it removes a certain degree of uncertainty [9].

Authors of the mentioned book proposed new fuzzy Bayesian network (NFBN) [1]. This method simplifies computation by including only the observed values that have non-zero membership in the studied range. NFBN generated probabilistic estimates could improve BNRC and SpaBN models. Authors proposed fuzzy Bayesian network with added residual correction mechanism (FBNRC) and spatial fuzzy Bayesian network (SpaFBN), where models are extended by NFBN. Both models were applied to mentioned above climatological data set [2] and compared with BNRC and SpaBN. Results showed that NFBN reduced the parameter uncertainty greatly, especially in precipitation, where range of values is quite large. In summary, models extended with NFBN had better match with actual spatial time series data.

2 Bayesian Network

Bayesian networks are also known as Bayes nets or belief networks. BN is a probabilistic graphical model that uses a directed acyclic graph (DAG) to describe a set of variables and their conditional dependencies. DAG is a causal network that has nodes that represent random variables in the domain of interest and edges that show how those variables are related to one another. Each edge between variables $N_i \rightarrow N_j$ indicates dependency, here N_i is interpreted as a parent node of N_j. If there is no edge between nodes, then variables are independent of each other. Each node has a conditional probability distribution $P(N_i|Parents(N_i))$ [1]. However, N_i is conditionally independent of its non-descendants. For example, in Figure 1 *Admission* is dependent on *Marks*, but the probability of *Admission* does not depend on the *Exam level* and *IQ level*.

Mathematically it could be expressed as:

$$P(N_i|Parents(N_i), ND(N_i)) = P(N_i|Parents(N_i)) \tag{1}$$

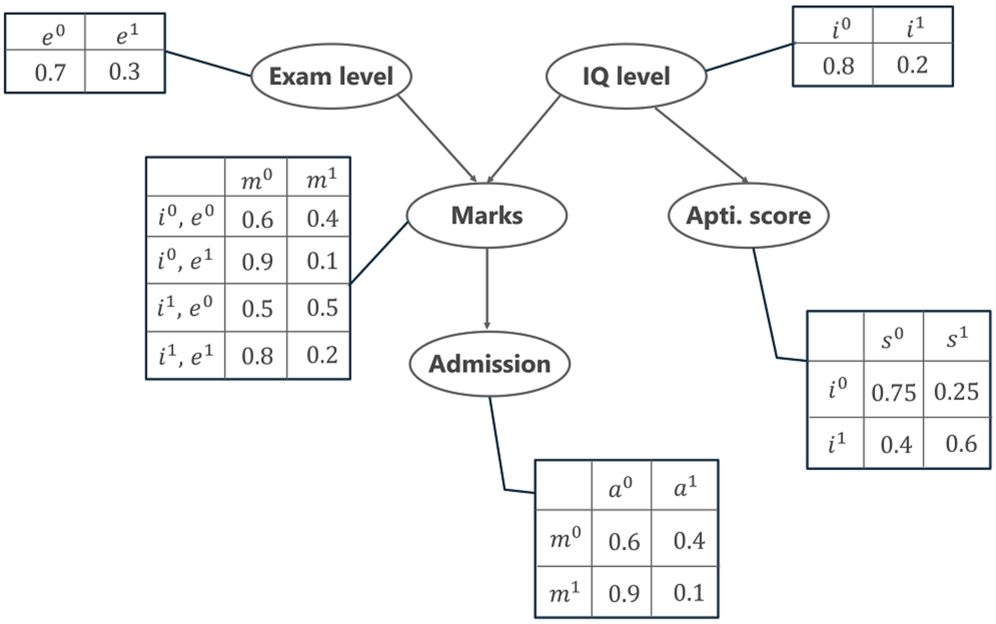

Figure 1: A simple Bayesian network. Source: [12].

where, $ND(N_i)$ is a set of non-descendants of N_i. In Bayesian network dependency structure is simply represented by joint probability density function as follows:

$$P(n_1, n_2, n_3, ..., n_i, ..., n_m) = \prod_{i=1}^{m} P(n_i|Parents(n_i)) \tag{2}$$

where, n_i and $parents(n_i)$ are specific values of the variables N_i and $Parents(N_i)$, respectively. This helps to solve complex problems more easily.

To build a Bayesian network two main things should be considered at first: what nodes will represent and what states or values will nodes have. Values could be either discrete or continuous. In case of fuzzy states, values will have intervals. Then structure of BN should be created. All nodes, which have effect on each other has to be linked. For example, in table 1 Exam level with Marks and Marks with Admission. In this case Marks is a parent node of Admission and Exam level is an ancestor node of Admission. When relationships between variables are considered, then conditional probability distribution is needed for each node. First, all possible combinations of parent nodes and child nodes should be considered. Then, probabilities that child node will take each specific value of parent node have to be indicated. Then joint probability distribution is expressed. And one of the most important feature of BN is to calculate inference. It means to generate posterior probability for the query variable. There are two types of variables: evidence and hidden. To evidence variables are assigned specific values and values of hidden variables are unknown. So, posterior probability of the query variable Q:

$$P(Q|E) = \alpha P(Q, E) = \alpha \sum_{Y} P(Y) \cdot P(Q, E|Y) = \alpha \sum_{Y} P(Q, E, Y) \tag{3}$$

where, E indicates a set of evidence variables, $Y = y_1, y_2, \ldots, y_n$ indicates set of hidden variables, α is a normalization constant. $P(X, E, Y)$ is a joint probability. Using all this procedure, any decision problem could be solved [1].

Described concept Bayesian network will be used in all models below.

2.1 Bayesian Network with Residual Correction Mechanism

In order to describe Bayesian Network with Residual Correction Mechanism first parameter learning will be overviewed, then inference generation and finally residual correction.

During the parameter learning process the data is divided into training and testing years: $y_1, y_2, \ldots, y_t$, $t =$ total number of training years. Each training year is treated separately to learn probabilistic relationships between variables to get a causal dependency graph and each network is denoted as $BN_{y_1}, BN_{y_2}, \ldots, BN_{y_t}$. Further, probabilities of the selected variables are weighted averaged to obtain the corresponding probability of query variable $x \in X$ for the prediction year y_{t+1}. $P^x_{C_i}$ stands for the probability table of the variable x as it was learned from training BN_i on year y_i data. Then the element for variable x in the final probability table becomes:

$$P^x_F = \sum_{i=1}^{t}(TW_i \times P^x_{C_i}) \tag{4}$$

where, TW_i is temporal weight of i^{th} training year, such $\sum_{i=1}^{t} TW = 1$ and it is defined as follows:

$$TW_i = \left(\frac{\frac{1}{dstnc_i}}{\sum_j^t = 1 \frac{1}{dstnc_j}} \right) \tag{5}$$

where, $dstnc$ is a temporal distance, such that $dstnc = [y_{t+1} - y_i]$

The learning process of BNRC is similar to standard BN. However, until inference generation probability distributions of all training year are taken into account.

In step of inference generation, it is described how BNRC draws conclusions about the value of a variable by taking into account a certain prediction year in the future y_{t+1}. The value of the prediction variable is inferred based on the evidences, which are assumed to be all geographical attributes and zero or more domain variables. Formula (2) is applied in Bayesian learning. The highest inferred probability in probability table is the most likely inferred value.

The next part explains how the residual correction mechanism adjusts the inferred value to account for the lack of data. This happens when the network inference may be impacted by the absence of confounding variables, which can behave as extraneous variables in the network, or in the case when number of variables is very large, to maintain accuracy, the parameter estimation needs an increasing amount of data. A residual is a quantity that measures how far an element's observed value deviates from its predicted function value. It is also known as fitting error and is an observable estimate of unobserved statistical error. The exponential average principle is used in BNRC. During network learning, the residual value created during inference generation is updated exponentially, and the final residual value is used to make up for the lack of other necessary but unidentified variables that might be present in the network topology. The current residual value ϵ_i is updated as follows each time the network is trained:

$$\epsilon_i = (\alpha e_i) + (1 - \alpha)\epsilon_{i-1} \tag{6}$$

where, $\alpha \in [0, 1]$ and is labelled as smoothing factor. $AV_{(i)}$ is the actual inferred value, for the day d in the year y, $TI^{(i)}_{X_j}$ is the tuned inferred value of the prediction variable x and error for the same day in the year y is called E_i, and it is calculated as follows:

$$E_i = AV_{(i)} - TI^{(i)}_{X_j} \tag{7}$$

The final value of residual ϵ_t is determined after training using the data from t prior years, and the tuned inferred value of x for the day d in the prediction year y is as follows.

$$\begin{aligned}
TI^{(t+1)}_{X_j} &= I^{(t+1)}_{X_j} + \epsilon_t \\
&= I^{(t+1)}_{X_j} + \alpha E_t + (1 - \alpha)\epsilon_{t-1} \\
&= I^{(t+1)}_{X_j} + \alpha[E_t + (1 - \alpha)E_{t-1}] + (1 - \alpha)^2\epsilon_{t-2} \\
&= I^{(t+1)}_{X_j} + \alpha[E_t + (1 - \alpha)E_{t-1} + ... + (1 - \alpha)^{(t+1)}E_1] + (1 - \alpha)^t\epsilon_0
\end{aligned}$$

where I_{t+1} represents the value of x that was inferred by the inference generating process of BNRC for the day d in the year y.

Finally, the predicted value becomes the one linked with the highest probability estimates $P(.)$ during inference generation, out of all the tuned inferred values of the prediction variable. If $pred_x$ is the predicted value of the variable x, then $pred_x = TI^{(t+1)}_{X_j}$ such that $P(I_{X_j}|e) = \max P(X_j|e)$, where the combination of values for the evidence variables is indicated by the symbol e. The predicted value $pred_x$ may alternatively be obtained as a range of values $[LB_j, UB_j]$, because the whole analysis takes into account the discretized values of the variables. The midpoint of the range may be taken into account in order to obtain a single value for the prediction variable, $pred_x = (LB_j + UB_j)/2$, where LB is the lower bound value and UB is the upper bound value.

2.2 Spatial Bayesian Network

Recently, the problem of a large number of influencing factors in a spatial time series prediction scenario was addressed by the spatial Bayesian network. The directed acyclic graph of the SpaBN structure also has composite nodes in addition to the regular standard nodes, where frequently composite nodes consist of numerous standard nodes that represent the same variable but are distributed spatially. As it is showed in a table 2 composite nodes are yellow and they consist of eight standard nodes in different locations.

Reducing learning time and space complexity of the spatial Bayesian network model is the main goal of adding composite nodes to the network. In other case DAG would be as the one in the Figure 2 on the left. So, by reducing structure complexity SpaBN reduces algorithmic complexity during parameter learning and inference generation as well.

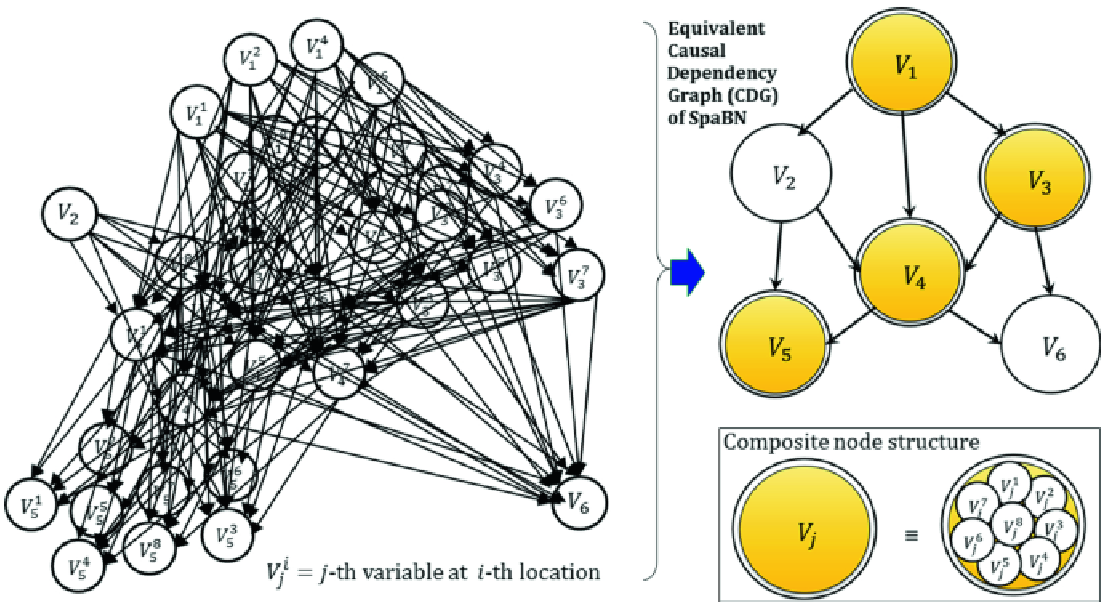
Equivalent Causal Dependency Graph (CDG) of SpaBN
V_1
V_2
V_3
V_4
V_5
V_6
Composite node structure
V_j
≡
V_j^i = j-th variable at i-th location

Figure 2: A CDG of SpaBN. Source: [1].

Before parameter learning process spatial weight must be calculated in SpaBN. Spatial weight of each location $L_i \in L_k$ in relation to the location of the prediction L_p shows how much impact has each variable for prediction variable. It could not only depend on the spatial distance, but also on point of interest (traffic flow prediction) etc. So spatial weight function differs depending on problem. In case when spatial importance (SW) could be calculated with spatial distance (SD) from prediction location, formula would look like:

$$SW_i = \frac{\sum_{j=1}^{|X|} NCorr^i_{X_j} + NISD_i}{\sum_{l=1}^{|L^k|}(\sum_{j=1}^{|X|} NCorr^l_{X_j} + NISD_l)} \tag{8}$$

where, $NCorr^i_{X_j}$ represents normalized correlation between the variable X_j in i^th location and the same variable in the predicted location. $NISD_i$ denotes normalized inverse spatial distance between i^th location and prediction location. Number of correlated variables depends on the problem. In other cases spatial weight could have other parameters instead of correlation, for example spatial weight in mentioned above hydrological problem [3] instead of normalized correlation between variables consists of normalized modified curve number, normalized value of water contributing area and normalized inverse distance.

In the parameter learning step, in case when variables in composite nodes are spatially distributed, all variables in composite nodes are multiplied by spatial weight (SW_i). In Figure 2 yellow nodes are composite nodes and spatial weight should be applied such that:

$$P(V_1) = \beta \left[\sum_{i=1}^{K} P(V_1^i) \cdot SW_i \right] \tag{9}$$

$$P(V_3) = \beta \left[\sum_{i=1}^{K} P(V_3^i) \cdot SW_i \right] \tag{10}$$

$$P(V_4) = \beta \left[\sum_{i=1}^{K} P(V_4^i) \cdot SW_i \right] \tag{11}$$

$$P(V_5) = \beta \left[\sum_{i=1}^{K} P(V_5^i) \cdot SW_i \right] \tag{12}$$

where, V_j^i denotes singular component in variable V_j. β is a normalization constant in order to sum probability to 1. Conditional probabilities between standard and composite nodes are calculated similarly, with normalization constant and spatial weight involved in learning process.

Inference generation process also includes SW. In example case, in Figure 2, inferred value of V_6 is calculated using probability distribution:

$$\begin{aligned} P(V_6|V_1, V_2, ..., V_4) &= \sum_{i=1}^{L} P(V_6|V_1^i, V_2, V_4^i) \cdot SW_i \\ &= \beta \sum_{i=1}^{K} \sum_{V_3} P(V_1^i) \cdot P(V_2|V_1^i) \cdot P(V_3|V_1^i) \cdot P(V_4^i|V_1^i, V_2, V_3^i) \cdot P(V_6|V_3^i, V_4^i) \cdot \\ &\cdot SW_i \end{aligned}$$

where, L is a number of locations.

Prediction value is considered the same as during BNRC process: $pred_x = (LB_j + UB_j)/2$, where LB is the lower bound value and UB is the upper bound value.

2.3 Spatial Bayesian Network with DBSCAN

In this paper in order to decide how many locations are included in SpaBN, Density Based Spatial Clustering of Applications with Noise (DBSCAN) will be applied. DBSCAN is made to find the noise and clusters in a spatial database [10]. It simply needs just an input parameter and helps the user choose a suitable value for it. It finds clusters with any shape. DBSCAN is effective for big spatial databases.

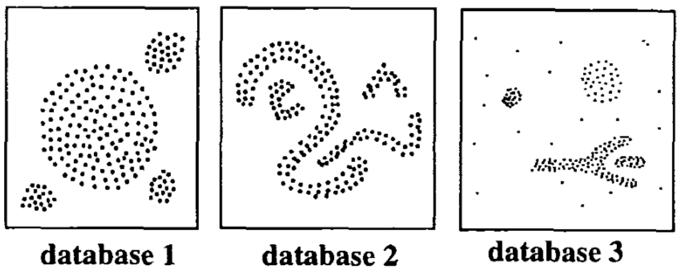

Figure 3: Samples of Databases. Source: [10].

In figure 3, each cluster has a typical density of points that is significantly higher inside the cluster than outside. Additionally, there is less density in the noisy zones than there is within any of the clusters. The main principle is that for each point in a cluster, the neighbourhood must have at least a certain number of points, meaning that the area's density must be above a certain level. The selection of a distance function between two points a and b determines the geometry of a neighbourhood (a, b). An ϵ-neighbourhood of a point a is indicated by the symbol $N_\epsilon(a)$, which is defined as $N_\epsilon(a) = \{b \in D | dist(a, b) \leq \epsilon\}$, where D is a database of points of some k-dimensional space S_k. A naive method can demand that each point in a cluster have a certain minimum number of points (MP) nearby it in an ϵ-neighbourhood. This method, however, fails because there are two types of points in a cluster: core points inside the cluster and periphery points (border points). In comparison to an ϵ-neighbourhood of a core point, a neighbourhood of a border point often has a lot fewer points. In order to include all points from the same cluster, we would therefore need to define the minimum number of points to a somewhat low figure. However, in the absence of noise, this value will not be representative of the particular cluster. As a result, we demand that there be a point b in a cluster C for every point a in a cluster C in order for a to be inside of b ϵ-neighbourhood and for $N_\epsilon(b)$ to include at least MP points. A point a is directly density-reachable from a point b with respect to ϵ and MP if

1. $a \in N_\epsilon(b)$ and
2. $|N_\epsilon(b)| \geq MP$

A point a is density-reachable from a point b with respect to ϵ and MP if there is a chain of points $a_1, \ldots, a_n$, $a_1 = b$, $a_n = a$ such that a_{i+1} is directly density-reachable from a_i. Due to the possibility that the core point condition may not apply to two border points, the same cluster C may not be density-reachable from one another. But C must have a central location from which both of its boundary locations may be density-reachable. As a result, the concept of density-connectivity includes this relationship between boundary points: a point a is density-connected to a point b with respect to ϵ and MP if there is a point c such that both, a and b are density-reachable from f with respect to ϵ and MP. Now, it is possible to define a notion of a cluster that is based on density. A cluster is intuitively understood to be a collection of places with the greatest density connectivity and reachability. In relation to a predetermined set of clusters, the noise will be defined. The collection of points in D that do not belong to any of its clusters is known as noise.

Definition 1. (Cluster) Let D represent a points database. A non-empty subset of D that meets the following criteria is referred to as a cluster C with respect to ϵ and MP:

1. $\forall$ a, b: if $a \in C$ and b is density-reachable from a with respect to ϵ and MP, then $b \in C$. (Maximality)
2. $\forall$ a, $b \in C$: a is density-connected to b with respect to ϵ and MP. (Connectivity)

Definition 2. (Noise) Let $C_1, \ldots, C_k$ be the clusters of the database D with respect to parameters ϵ and MP_i, $i = 1, \ldots, n$. The collection of database D points that do not belong to any cluster C_i is how the noise defined, i.e. $\{a \in D | \forall i : a \notin C_i\}$.

In order to find a cluster, first, it is necessary to select as a seed a point that satisfies the core point criterion. Second, obtain the cluster holding the seed and then extract all points that are density-reachable from the seed. The accuracy of the clustering approach must be confirmed using the following lemmas.

Lemma 1: Let a be a point in D and $|N_\epsilon(a)| > MP$. Then the set $F = \{f \mid f \in D$ and o is density-reachable from a with respect to ϵ and $MP\}$ is a cluster with respect to ϵ and MP.

A cluster C includes just the points that are density-reachable from any core point of C because every point in C is density-reachable from any of the core points of C.

Lemma 2: Let C be a cluster with respect to ϵ and MP and let a be any point in C with $|N_\epsilon(a)| > MP$. Then C equals to the set $F = \{f \mid o$ is density-reachable from a with respect to ϵ and $MP\}$.

According to definitions 1 and 2, DBSCAN is made to identify clusters and noise in a spatial database. Ideally, the suitable parameters ϵ and MP of each cluster and at least one point from the respective cluster are known. However, there is no simple way to obtain this information in advance for all database clusters. As a result, DBSCAN uses global values for ϵ and MP, i.e., values that are the same across all clusters. For these global parameter values describing the lowest density that is not regarded to be noise, the density parameters of the "thinnest" cluster make good candidates. DBSCAN starts with an arbitrary point a and obtains all points density-reachable from a with respect to ϵ and MP in order to locate a cluster. This process produces a cluster regarding ϵ and MP if a is a core point (see Lemma 2). If a is a border point, then DBSCAN scans the next point in the database because a is not density-reachable from any other locations. ϵ and MP are calculated using global values, hence DBSCAN may combine two clusters that meet definition 1 into a single cluster, if two clusters with differing densities are near to one another. Let the distance between two sets of points S_l and $S_2 = \min\{dist(a,b)|a \in S_1, b \in S_2\}$ If the distance between the two sets is greater than ϵ, then two sets of points that have at least the density of the thinnest cluster can be separated from one another. The discovered clusters with a larger value for MP may therefore require a DBSCAN recursive call. This is not a drawback as the recursive application of DBSCAN results in a fundamental algorithm that is efficiently basic.

In SpaBN with DBSCAN clustering algorithm, spatial weights are calculated the same as in SpaBN, except between locations of each cluster.

2.4 New Fuzzy Bayesian Network

The concept of fuzzy set membership is used in the extension of probability theory known as fuzzy probability theory. The same concept is applied in the New Fuzzy Bayesian Network [1]. Problems with the boundary values arise when a continuous variable is discretized, adding extra uncertainties or imprecisions to the data. This limits the ability to choose appropriate samples during the learning phase. NFBN is a variant of Fuzzy Bayesian Network (FBN). Given the fuzzy membership of each particular observed value into the other ranges, NFBN generates more accurate parameter estimates. Additionally, NFBN reduces the amount of time needed by substituting a more straightforward computation involving only the observed values that have non-zero membership in the considered range.

Let $F_1, F_2, ...F_k$ and $G_1, G_2, ...G_l$ be two sets of events respective to the variables x and y, where $F_1, F_2, ...F_k$ and $G_1, G_2, ...G_l$ are in the form of range of values attained by x and y;

$k > 0$ and $l > 0$. Moreover, $\tilde{F}$ and $\tilde{G}$ are any two corresponding fuzzy events. Then, based on NBFN:

$$P(\tilde{G}/\tilde{F}) = \frac{|\{n_i|\mu_{\tilde{G}}(y_{n_i}) > 0, \mu_{\tilde{F}}(x_{n_i}) > 0\}|}{N \cdot P(\tilde{F})} \tag{13}$$

$$= \frac{\left[P(F,G) + \frac{\alpha}{N}\right]}{\left[P(F) + \frac{\beta}{N}\right]} \tag{14}$$

$$= P(G/F) \cdot \frac{\left[1 + \frac{\alpha}{N \cdot P(F,G)}\right]}{\left[1 + \frac{\beta}{N \cdot P(F)}\right]} \tag{15}$$

where, F and G are crisp sets; The set $n_1, n_2, ..., n_N$ contains all observations for the variables x and y, where N is the total number of such observations; x_{n_i} and y_{n_i} are values of the variables x and y, respectively, in the i^{th} observation (n_i); $\mu_{\tilde{F}}(x_{n_i})$ and $\mu_{\tilde{G}}(y_{n_i})$ are memberships of the values x_{n_i} and y_{n_i} respectively in the fuzzy sets $\tilde{F}$ and $\tilde{G}$ correspondingly; α is the number (or count) of observations n_i such $x_{n_i} \notin F$ and $y_{n_i} \notin G$, but $\mu_{\tilde{F}}(x_{n_i}) > 0$ and $\mu_{\tilde{G}}(y_{n_i}) > 0$; β is the count of observations n_i such $x_{n_i} \notin F$, but $\mu_{\tilde{F}}(x_{n_i}) > 0$.

So, fuzzy marginal probability $P(\tilde{F})$ is defined as follows:

$$P(\tilde{F}) = \frac{|\{n_i|\mu_{\tilde{F}}(x_{n_i}) > 0, n_i \in n_1, n_2, ..., n_N\}|}{N} \tag{16}$$

$$= P(F) + \frac{\beta}{N} \tag{17}$$

$$= P(F) \cdot \left[1 + \frac{\beta}{N \cdot P(F)}\right] \tag{18}$$

where, F is a crisp set; $n_1, n_2, ..., n_N$ contains all observations of the variable x, where N is the total number of such observations; $\mu_{\tilde{F}}(x_{n_i})$ is a membership of the value x_{n_i} in the fuzzy set $\tilde{F}$; β is the number or count of observations n_i such that $x_{n_i} \notin F$, but $\mu_{\tilde{F}}(x_{n_i}) > 0$.

2.5 Enhanced Fuzzy Bayesian Network Models

When inserted into a spatial time series prediction framework, the NFBN is already discovered to provide superior prediction accuracy than the usual BN based analysis [1]. So it is reasonable to assume that when extended with NFBN generated probabilistic estimates, SpaBN and BNRC will provide greater prediction performance.

Let P_{PL} represent conditional probability distribution achieved during parameter learning in BNRC and P_{NFBN} achieved using NFBN. In the principle of BNRC, the probability could be expressed as a function of standard Bayesian probability distribution P and the tuning parameter or residual value ϵ:

$$P_{BNRC} = f_1(P, \epsilon) \tag{19}$$

Fuzzy Bayesian network with added residual correction mechanism (FBNRC) is an enhanced version of BNRC improved with NFBN. FBNRC could be expressed as function of P_{NFBN} and the tuning parameter ϵ:

$$P_{FBNRC} = f_2(P_{NFBN}, \epsilon) \tag{20}$$

Spatial fuzzy Bayesian network (SpaFBN) is another enriched version of SpaBN with NFBN [1]. As conditional probability distribution achieved during parameter learning in SpaBN could be expressed as a function of standard Bayesian probability distribution P and spatial weights (SW):

$$P_{SpaBN} = f_3(P, SW). \tag{21}$$

Probability distribution of SpaFBN could be expressed similarly, as a function of P_{NFBN} and spatial weights (SW):

$$P_{SpaFBN} = f_4(P_{NFBN}, SW). \tag{22}$$

2.6 Wild Bootstrap

In order to get more data to increase the accuracy of predictions, more samples were generated using wild bootstrapping. Bootstrapping refers to any test or statistic that includes random sampling with replacement and is a subset of the larger category of resampling techniques. The goal of the residual bootstrap is to resample the response variable based on the residuals values while leaving the regressors at their sample value. In other words, a new y_i^* is calculated based on $\hat{y}_i + \hat{\epsilon}_i v_i$ for each replicate. Therefore, a random variable v_i with a mean of 0 and a variance of 1 is randomly multiplied by the residuals. In this paper, the standard normal distribution is used for the random variable v_i [11].

3 Results

3.1 Case study 1: Precipitation Prediction

3.1.1 Experimental Setup

At first the experimentation was carried out over two merged sets of data to predict precipitation in counties of the United States with respect to the Air Quality Index. The data is provided by the online community platform kaggle.com. One of the datasets contains weather events data of the United States like the type of event, severity, start and end time of event, precipitation and location of a weather event [14]. Another dataset describes air quality in different areas of the United States and the dataset contains state and county name, AQI, in which category AQI belongs and the date when AQI was measured [13]. AQI dataset was merged to weather events dataset by state name, county name and date as AQI is measured once per day.

Air quality index is assigned into three categories: Good, Moderate and Unhealthy, where AQI index is divided into ranges '0-50', '50-100' and '>100', respectively. Precipitation was converted from inches to millimeters by multiplying by 25,4. Precipitation is divided into more precise groups: '0', '0-0.3', '0.3-5', '5-15', '15-30', '30-50', '50-100', '100-600' and '>600'. The dataset (with 501214 records) has been divided into two different subsets: a training set, where data includes years 2016-2018 and a test set, where data is in 2019. Data is collected from the locations marked with points in figure 4.

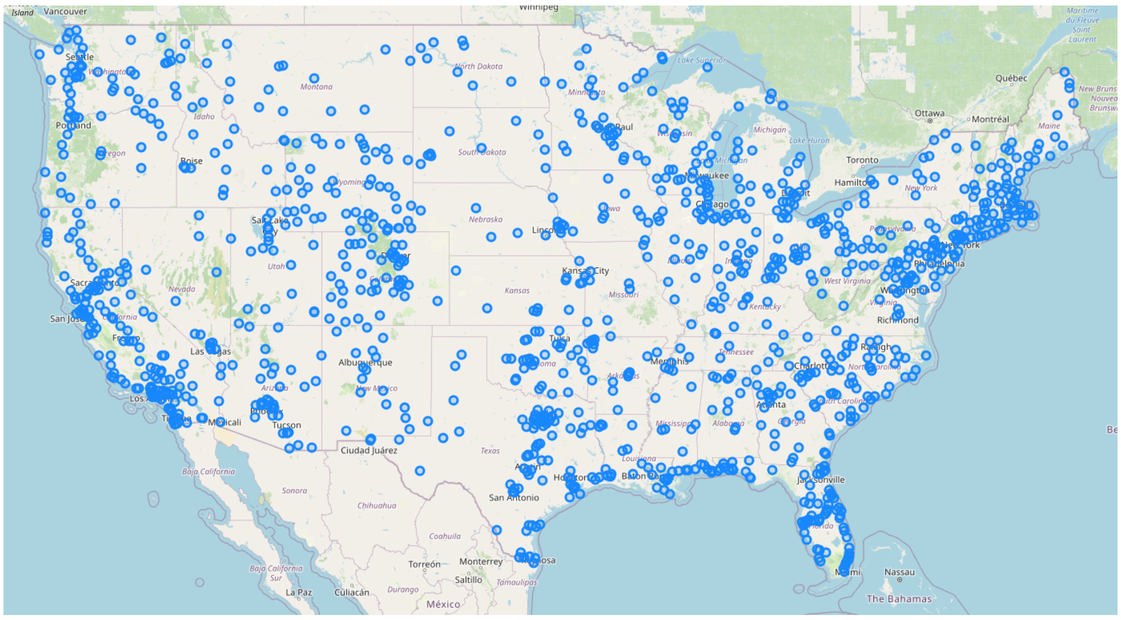

Figure 4: Locations of collected data.

Probabilities were estimated for each training year for every county location in every state of the USA both for AQI and precipitation. Table 1 shows an example of Denver county of Colorado state estimated probability of AQI.

Category	2016	2017	2018
Good	0.573	0.569	0.571
Moderate	0.420	0.417	0.407
Unhealthy	0.008	0.014	0.021

Table 1: Estimated probability of AQI of Denver county, Colorado state.

Table 2 shows an example of Denver county of Colorado state estimated probability of precipitation when weather quality is 'Good'.

The temporal distance between the prediction year and the oldest year of the data is equal to 3. So the probability of prediction year is as follows:

$$P_{2019_{pred}} = 0.1818 \cdot P_{2016} + 0.2727 \cdot P_{2017} + 0.5455 \cdot P_{2018}$$

DAG of such Bayesian Network is displayed in figure 5.

As DAG in figure 5 shows Bayesian Network is simple, so probabilities of precipitations are calculated with formula $P(Precipitation_i|Category_j)$, where i shows intervals of precipitation and j shows intervals of AQI.

Precipitation	2016	2017	2018
0	0.467	0.402	0.438
0-0.3	0.067	0.024	0.075
0.3-5	0.360	0.427	0.388
5-15	0.080	0.122	0.075
15-30	0.027	0.024	0.025
30-50	0.000	0.00	0.00

Table 2: Estimated probability of precipitation of Denver county, Colorado state.

Category	2016	2017	2018	2019_{pred}
Good	0.573	0.569	0.571	0.571
Moderate	0.420	0.417	0.407	0.412
Unhealthy	0.008	0.014	0.021	0.017

Table 3: Estimated probability of AQI of Denver county, Colorado state with prediction year.

When probabilities are calculated, middle values of intervals are chosen as inferred values, where probability is the highest. Then residual correction is applied. Smoothing parameter is chosen 0.5. Results are summarized in table 4.

To improve BNRC, fuzzy states were applied. NFBN method requires a count of observations, which are not included in the original intervals, thus this method does not require high computation power. New intervals were generated, if new generated intervals and original intervals overlap, then those observations that are not included in the original intervals but are in new intervals are summed to probabilities as in formula 16. New intervals of precipitation: '<0.1', '0.1-1', '2.5-7.5', '10-20', '20-40', '40-70', '80-150', '500-700'.

In SpaBN, SpaFBN and SpaBN with DBSCAN methods the scope of prediction is limited to all capital counties of the United States. Without Hawaii and Alaska there are 48 states, so predictions are made only in 48 counties, where the capital cities of states are. Points in figure 6 show the locations of targeted places.

Spatial weight is counted between every county and the capital county in every state. As well correlation of precipitation and AQI between county and capital county. So spatial weight is as follows:

$$SW_i = \frac{\sum_{j=1}^{|AQI|} NCorr^i_{AQI_j} + \sum_{j=1}^{|Prcp|} NCorr^i_{Prcp_j} + NISD_i}{\sum_{l=1}^{|L^k|}(\sum_{j=1}^{|AQI|} NCorr^i_{AQI_j} + \sum_{j=1}^{|Prcp|} NCorr^i_{Prcp_j} + NISD_l)},$$

where AQI responds to the variable Air Quality Index, Prcp corresponds to variable Precipitation, other notations are the same as in formula 8.

Probability tables of AQI and precipitation are multiplied by SW according to the county and state and the normalization constant is applied. Then temporal distance is counted the same as in the BNRC method. Probabilities of precipitations are estimated the same as in BNRC method, with formula $P(Precipitaion_i|Category_j)$, where i shows intervals of precipitation and j shows intervals of AQI. When probabilities are calculated, middle values

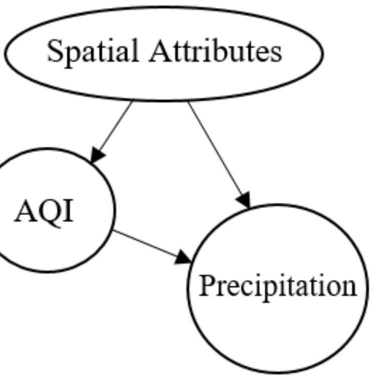

Figure 5: DAG of case study 1.

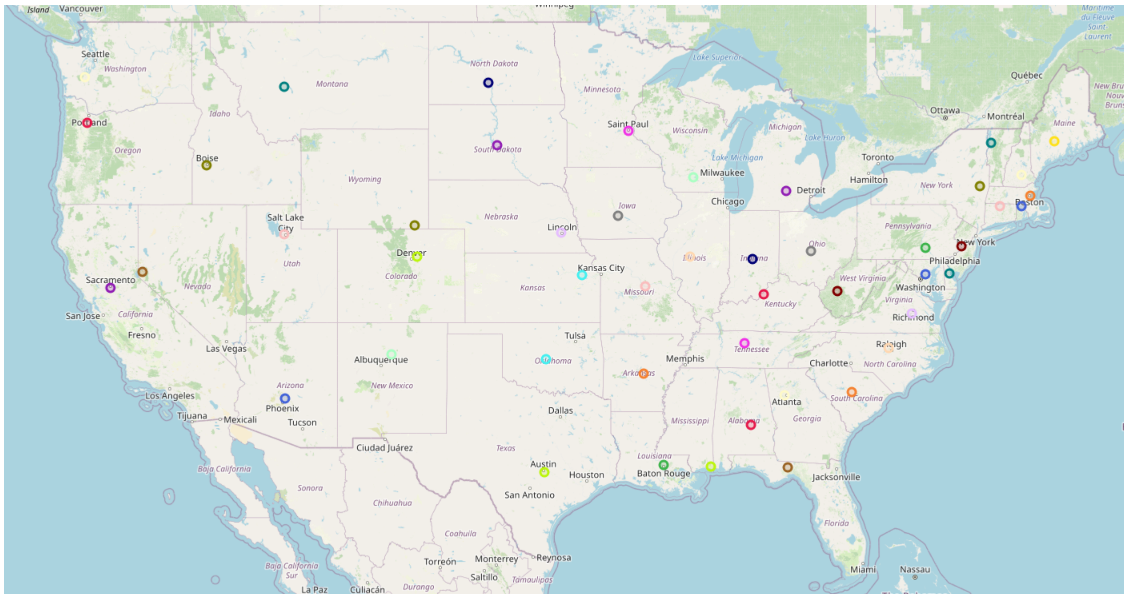

Figure 6: Counties of capitals of the states.

of intervals are chosen as inferred values, where probability is the highest. Results are displayed in table 5.

In SpaBN spatial weight was calculated with respect to all counties of state. In SpaBN with DBSCAN, DBSCAN method creates classes, if the class contains the capital county, then the spatial weight is calculated for every county in class with respect to the capital county. Figure 7 shows clusters of counties calculated with DBSCAN method. During this experiment $\epsilon = 0.5$ and $MP = 1$. Further, estimations are done as in the SpaBN.

In SpaFBN fuzzy states were generated the same as in FBNRC. Further BN analysis is the same as in SpaBN.

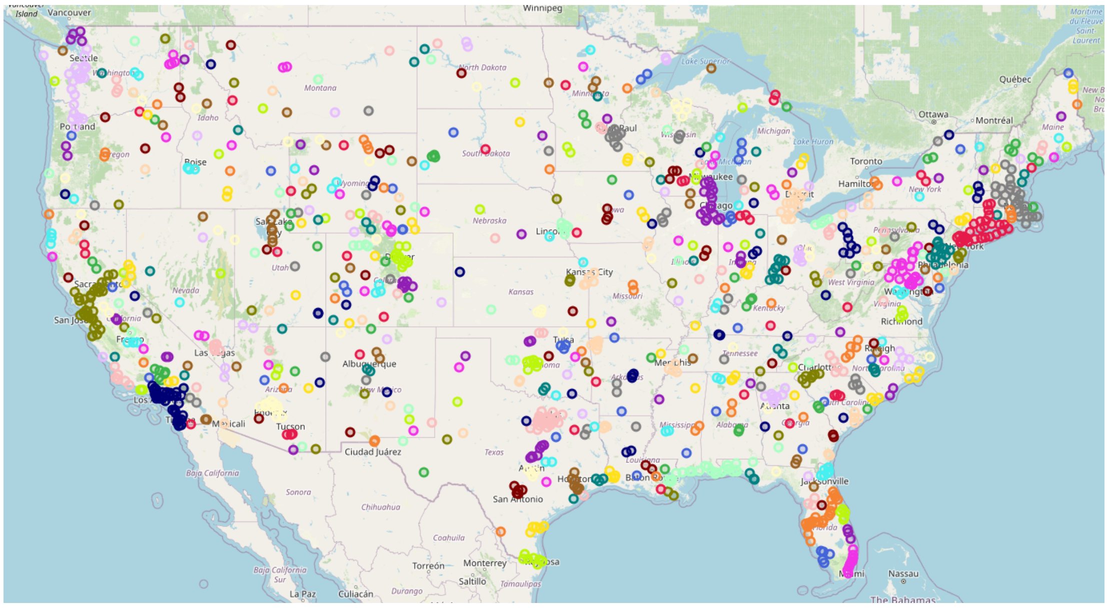

Figure 7: Clusters of counties.

3.1.2 Results

The performance of all methods is measured using statistical measures as coefficient of determination – R^2, Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE). Observed and predicted values are perfectly aligned when the R^2 value is 1 and R^2 is calculated as follows:

$$R^2 = \frac{\left[\sum_{i=1}^{N}(v_{A_i} - \bar{v_A})(v_{P_i} - \bar{v_P})\right]^2}{\sum_{i=1}^{N}(v_{A_i} - \bar{v_A})^2 \cdot \sum_{i=1}^{N}(v_{P_i} - \bar{v_P})^2}, \tag{23}$$

where v_{A_i} is the actual value of i^th observation of the variable, $\bar{v_A}$ is the mean of actual values of measurement variable, v_{P_i} is the predicted value of i^th observation of the variable, $\bar{v_P}$ is the mean of predicted values of the measurement variable, N is the total number of observations.

Other two measures indicates the perfect fit between observed and predicted values when MAE = 0 or RMSE = 0. MAE:

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|v_{A_i} - v_{P_i}| \tag{24}$$

and RMSE:

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(v_{P_i} - v_{A_i})^2} \tag{25}$$

When predictions were made for all counties, where test data of 2019 exist, BNRC and FBNRC showed very good predictions based on R^2 displayed in table 4. Fuzzy states did not improve BNRC based on R^2 measurement. However, if comparing RMSE and MAE, errors do not fall far from true values in FBNRC as much as in BNRC. When predictions are compared on capital counties, BNRC and FBNRC showed worse predictions, except BNRC gave smaller RMSE. FBNRC showed better performance based on R^2 and RMSE. Predictions of precipitation of all spatial BNs was better than BNs with residual correction mechanism. SpaFBN showed largest R^2, but very similar to other SpaBNs. However, if comparing RMSE and MAE, errors fall least far from true values in SpaBN with DBSCAN.

	BNRC	FBNRC
R^2	0.996	0.979
MAE	7.254	6.557
RMSE	43.186	25.438

Table 4: Predictions for all data set, where 2019 data was available.

	BNRC	FBNRC	SpaBN	SpaFBN	SpaBN with DBSCAN
R^2	0.542	0.776	0.879	0.885	0.878
MAE	8.168	8.679	5.519	5.521	5.407
RMSE	29.148	28.875	15.421	15.353	15.071

Table 5: Predictions for capital counties.

Wild bootstrapping was applied to generate 500 samples to re-evaluate the models. Table 6 shows results after wild bootstrapping resampling. Wild bootstrapping improved all methods significantly based on R^2 and RMSE, except Fuzzy BNs. R^2 of FBNRC decreased and MAE and RMSE of SpaFBN increased. However results of predictions did not change drastically of Fuzzy BNs, so wild bootstrapping improved results. SpaBN and SpaBN with DBSCAN displayed very good fit based on R^2. Wild bootstrapping did not reduce MAE in any of Spatial BN. SpaBN with DBSCAN performed better than SpaBN.

	BNRC	FBNRC	SpaBN	SpaFBN	SpaBN with DBSCAN
R^2	0.687	0.697	0.951	0.886	0.952
MAE	8.483	8.532	5.661	5.538	5.591
RMSE	28.816	28.787	14.726	15.361	14.386

Table 6: Predictions for capital counties.

The volume of the conditional probability table increases exponentially with the number of parent nodes, which is one of the main drawbacks of any Bayesian network model. So, very simple DAGs were chosen for the computational complexity and programming part, as BN methods are very new and not implemented in any programming language. Intuitively, the BNRC method should work better than SpaBN on very simple CDG, as BNRC should compensate for unknown variables in the CDG. While SpaBN models, which are better to deal with large numbers of variables, worked better. This might be because there were many locations in one state that influence the capital county in SpaBN, while in BNRC BNs were calculated in every location separately. Moreover, DBSCAN improved SpaBN predictions. The DBSCAN clustering algorithm is useful, because the decision of how many locations to take into spatial weight is not needed. In this paper locations from the same state were taken into spatial weight with respect to the capital county of the state.

3.2 Case Study 2: Weather Type Prediction

As results did not show reliable, trustworthy predictions, experiment was carried out on categorical variables. Second experimental is carried out over the same two merged sets of data to predict weather type (Rain, Snow, Fog, Storm, Cold) in counties of United States with respect to Air Quality Index. BNRC, FBNRC, and SpaFBN methods need numerical measurement variables, so only SpaBN and SpaBN with DBSCAN methods were applied. Spatial weight was estimated only taking spatial distance into account as correlation could not be calculated between categorical variables:

$$SW_i = \frac{NISD_i}{\sum_{l=1}^{|L^k|} NISD_l},$$

where notations are the same as in formula 8.

DAG of such Bayesian Network is displayed in figure 8.

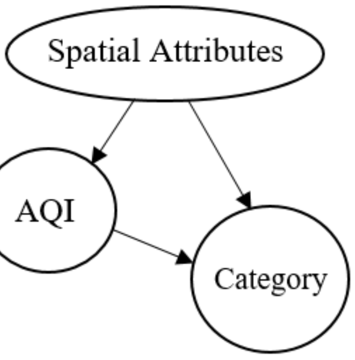

Figure 8: DAG of case study 2.

The following analysis is the same as in the case study 1 for SpaBN and SpaBN with DBSCAN.

In table 7 results show better predictions than a random walk, but results should be improved. In this case DBSCAN did not improve predictions.

	SpaBN	SpaBN with DBSCAN
Accuracy	0.617	0.615

Table 7: Predictions for capital counties.

Conclusions

As an advantage, improvements of enhanced BNs do not increase parameter learning complexity significantly compared to standard BN. Moreover, Bayesian network models have a more clear path from inputs to outputs compared to Artificial Intelligence algorithms. The volume of the conditional probability table increases exponentially with the number of parent nodes, which is one of the main drawbacks of any Bayesian network model. So chosen DAGs are simple due to the computational complexity of BNs and programming, as used BN methods are very new and not implemented in any programming language. Moreover, the dataset did not have many variables to add to DAGs.

Five different Bayesian Networks (BNRC, FBNRC, SpaBN, SpaFBN and SpaBN with DBSCAN) were described and analysed. These methods were compared on spatial time series data (merged dataset of USA area) by predicting precipitation and weather type with respect to the Air Quality Index.

First, precipitation with respect to AQI was predicted. All Bayesian Networks showed tolerable results, but still there are errors, that fall far from true values. Predictions of precipitation of all spatial BNs were better than BNs with residual correction mechanisms. SpaFBN showed the largest R^2, but was very similar to other SpaBNs. However, if comparing RMSE and MAE, errors fall least far from true values in SpaBN with DBSCAN.

In order to improve results, wild bootstrapping was implemented into all models. This resampling technique improved some of the models. Wild bootstrapping improved all methods significantly based on R^2 and RMSE, except Fuzzy BNs. However results of predictions did not change drastically of Fuzzy BNs, so wild bootstrapping improved results. SpaBN and SpaBN with DBSCAN displayed a very good fit based on R^2. In case study 1 SpaBN with DBSCAN performed better than SpaBN.

In the first case, Bayesian Networks could be improved by changing intervals of precipitation or AQI, changing fuzzy states. Longer time series could improve results as well, especially for SpaBNs, as after bootstrapping it showed better results.

In case sudy 2, weather types were predicted with respect to AQI. BNRC, FBNRC, and SpaFBN methods need numerical measurement variables, so only SpaBN and SpaBN with DBSCAN methods were applied. With simple spatial weight, the accuracy of both methods was very similar 0.617 and 0.615, DBSCAN clustering algorithm did not improve the result. However, the results of weather type prediction are not reliable enough.

Probabilities of precipitation and weather events were calculated. The results of the thesis are not very strong, as RMSEs are very high of all methods. All five methods could be compared on different datasets with a longer history of data. SpaBN models, which are better to deal with large numbers of variables, worked better. This might be because there was many locations in one state that influence the capital country in SpaBN, while in BNRC BNs were calculated in every location separately. Moreover, DBSCAN improved SpaBN predictions in case study 1 and almost did not change accuracy in case study 2. The DBSCAN clustering algorithm is useful, because the decision of how many locations to take into spatial weight is not needed. Wild bootstrapping resampling technique also improved predictions. Programming part could be used for further Bayesian Networks analysis and as a template for more complex CDGs.

References

[1] Das, Monidipa & Ghosh, Soumya. (2020). Enhanced Bayesian Network Models for Spatial Time Series Prediction: Recent Research Trend in Data-Driven Predictive Analytics. 10.1007/978-3-030-27749-9.

[2] Das, M., Ghosh, S.K. (2017). Spatio-Temporal Prediction of Meteorological Time Series Data: An Approach Based on Spatial Bayesian Network (SpaBN). In: Shankar, B., Ghosh, K., Mandal, D., Ray, S., Zhang, D., Pal, S. (eds) Pattern Recognition and Machine Intelligence. PReMI 2017. Lecture Notes in Computer Science(), vol 10597. Springer, Cham. https://doi.org/10.1007/978-3-319-69900-4_78

[3] M. Das, S. K. Ghosh, P. Gupta, V. M. Chowdary, R. Nagaraja and V. K. Dadhwal, "FORWARD: A Model for Forecasting Reservoir Water Dynamics Using Spatial Bayesian Network (SpaBN)," in IEEE Transactions on Knowledge and Data Engineering, vol. 29, no. 4, pp. 842-855, 1 April 2017, doi: 10.1109/TKDE.2016.2647240.

[4] Louzada F, Nascimento DCd, Egbon OA. Spatial Statistical Models: An Overview under the Bayesian Approach. Axioms. 2021; 10(4):307. https://doi.org/10.3390/axioms10040307

[5] Manisalidis I, Stavropoulou E, Stavropoulos A, Bezirtzoglou E. Environmental and Health Impacts of Air Pollution: A Review. Front Public Health. 2020 Feb 20;8:14. doi: 10.3389/fpubh.2020.00014. PMID: 32154200; PMCID: PMC7044178.

[6] Ropero, RF, Rumí, R, Aguilera, PA. Bayesian networks for evaluating climate change influence in olive crops in Andalusia, Spain. Nat Resour Model. 2019; 32:e12169. https://doi.org/10.1111/nrm.12169

[7] Balbi, S., Villa, F., Mojtahed, V., Hegetschweiler, K. T., and Giupponi, C.: A spatial Bayesian network model to assess the benefits of early warning for urban flood risk to people, Nat. Hazards Earth Syst. Sci., 16, 1323–1337, https://doi.org/10.5194/nhess-16-1323-2016, 2016.

[8] Aderhold, A., et al., Hierarchical Bayesian models in ecology: Reconstructing species interaction networks from nonhomogeneous species abundance data, Ecological Informatics (2012), doi:10.1016/j.ecoinf.2012.05.002

[9] M. Mrówczyńska, M. Skiba, A. Leśniak, A. Bazan-Krzywoszańska, F. Janowiec, M. Sztubecka, R. Grech, J.K. Kazak, A new fuzzy model of multi-criteria decision support based on Bayesian networks for the urban areas' decarbonization planning, Energy Conversion and Management, Volume 268, 2022, 116035, ISSN 0196-8904, https://doi.org/10.1016/j.enconman.2022.116035.

[10] Ester, M, Kriegel, H P, Sander, J, and Xiaowei, Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. United States: N. p., 1996. Web.

[11] Russell Davidson, Emmanuel Flachaire. The wild bootstrap, tamed at last. Econometrics, 2008, 146 (1), pp.162. ff10.1016/j.jeconom.2008.08.003ff. ffhal-00520648f

[12] Z. Lateef. "How To Implement Bayesian Networks In Python? – Bayesian Networks Explained With Examples". https://www.edureka.co/blog/bayesian-networks/. Last updated on Nov 25, 2020.

[13] Air Quality in USA Dataset (2015-2020). https://www.kaggle.com/code/zarinhelena/air-quality-in-usa-2015-2020/data?select=daily_aqi_by_county_2015.csv. Last updated in 2021.

[14] Weather Dataset. https://www.kaggle.com/datasets/muthuj7/weather-dataset. Last updated in 2018.

[15] The Clean Air Act. (42 U.S.C. 7401 et seq.) https://www.govinfo.gov/content/pkg/USCODE-2010-title42/html/USCODE-2010-title42-chap85.htm. Last updated in 2010.

Appendix

Appendix 1. Python Code of Case Study 1.

Data Merge

```
import pandas as pd
import numpy as np
from sklearn.metrics import mean_absolute_error, accuracy_score, mean_squared_error
import matplotlib.pyplot as plt
plt.style.use('ggplot')
import seaborn as sns
from sklearn.cluster import KMeans, DBSCAN
from sklearn.metrics import silhouette_score
import folium
import math
import scipy.stats as stats
import datetime

def rsquared_book(x, y):
    x_list = list(x)
    y_list = list(y)
    n = len(x_list)
    x_mean = sum(x_list)/n
    y_mean = sum(y_list)/n
    r1 = 0
    l1 = 0
    k1 = 0
    for i, j in zip(x_list, x_list):
        r = (i - x_mean) * (j - y_mean)
        r1 = r1 + r
        l = (i - x_mean) ** 2
        l1 = l1 + l
        k = (j - y_mean) ** 2
        k1 = k1 + k

    return r1 ** 2 / (l1 * k1)

# Weather events data
filename = r'C:\Users\Vartotojas\Desktop\WeatherEvents_Jan2016-Dec2021.csv'
weather_events = pd.read_csv(filename)
weather_events['StartTime(UTC)'] = pd.to_datetime(weather_events['StartTime(UTC)'])
weather_events['EndTime(UTC)'] = pd.to_datetime(weather_events['EndTime(UTC)'])
weather_events['Duration'] = weather_events['EndTime(UTC)'] -
weather_events['StartTime(UTC)']
weather_events['StartTime'] = weather_events['StartTime(UTC)'].dt.strftime('%Y-%m-%d')
weather_events['EndTime'] = weather_events['EndTime(UTC)'].dt.strftime('%Y-%m-%d')
```

```
weather_events["Year"] = pd.DatetimeIndex(weather_events["StartTime(UTC)"]).year

states_w = weather_events['State'].unique()
states_names_data = {
    'State name': ['Alabama', 'Arizona', 'Arkansas', 'California', 'Colorado',
     'Connecticut', 'Delaware', 'District of Columbia', 'Florida', 'Georgia',
     'Idaho', 'Illinois', 'Indiana', 'Iowa', 'Kansas', 'Kentucky', 'Louisiana',
     'Maine', 'Maryland', 'Massachusetts', 'Michigan', 'Minnesota', 'Mississippi',
     'Missouri', 'Montana', 'Nebraska', 'Nevada', 'New Hampshire', 'New Jersey',
     'New Mexico', 'New York', 'North Carolina', 'North Dakota', 'Ohio',
     'Oklahoma', 'Oregon', 'Pennsylvania', 'Rhode Island', 'South Carolina',
     'South Dakota', 'Tennessee', 'Texas', 'Utah', 'Vermont', 'Virginia',
     'Washington', 'West Virginia','Wisconsin', 'Wyoming'],
    'Code': ['AL', 'AZ', 'AR', 'CA', 'CO', 'CT', 'DE', 'DC', 'FL', 'GA', 'ID',
    'IL', 'IN', 'IA', 'KS', 'KY', 'LA', 'ME', 'MD', 'MA', 'MI', 'MN', 'MS', 'MO',
    'MT', 'NE', 'NV', 'NH', 'NJ', 'NM', 'NY', 'NC', 'ND', 'OH', 'OK', 'OR', 'PA',
    'RI', 'SC', 'SD', 'TN', 'TX', 'UT', 'VT', 'VA', 'WA', 'WV', 'WI', 'WY']}
states_names = pd.DataFrame(data=states_names_data)
merge_states = weather_events.merge(states_names, left_on=['State'], right_on=['Code'])
merge_states = merge_states.drop(columns='Code')

aqi = []
for i in [15, 16, 17, 18, 19, 20]:
    filename = r'C:\Users\Vartotojas\Desktop\\daily_aqi_by_county_20%s.csv' % i
    data_aqi = pd.read_csv(filename)
    aqi.append(data_aqi)

cols = ['State Name', 'county Name', 'State Code', 'County Code', 'Date', 'AQI',
'Category', 'Defining Parameter', 'Defining Site', 'Number of Sites Reporting']
aqi_df = pd.concat(aqi)

merged_data = merge_states.merge(aqi_df, left_on=['StartTime', 'County', 'State name'],
                                 right_on=['Date', 'county Name', 'State Name'])

merged_data.loc[merged_data['Category'] == 'Unhealthy for Sensitive Groups',
'Category'] = 'Unhealthy'
merged_data.loc[merged_data['Category'] == 'Very Unhealthy', 'Category'] = 'Unhealthy'
merged_data.loc[merged_data['Category'] == 'Hazardous', 'Category'] = 'Unhealthy'
merged_data.loc[merged_data['Severity'] == 'Other', 'Severity'] = 'UNK'
merged_data.loc[merged_data['Type'] == 'Hail', 'Type'] = 'Storm'
merged_data['Precipitation(mm)'] = merged_data['Precipitation(in)'] * 25.4
merged_data_max = merged_data.groupby(['State name', 'County', 'Date'],
as_index=False, sort=False)['Precipitation(mm)'].max()
merged_data_precip = merged_data.merge(merged_data_max, left_on=['State name',
'County', 'Date', 'Precipitation(mm)'],right_on=['State name', 'County', 'Date',
'Precipitation(mm)'])
merged_final = merged_data_precip.drop_duplicates(subset=['State name', 'County',
'Date', 'Precipitation(mm)'])
intervals = []
for row in merged_final['Precipitation(mm)']:
    if row == 0.0:
        intervals.append('0')
    elif 0.0 < row <= 0.3:
        intervals.append('0-0.3')
    elif 0.3 < row <= 5.0:
        intervals.append('0.3-5')
    elif 5.0 < row <= 15.0:
        intervals.append('5-15')
    elif 15.0 < row <= 30.0:
        intervals.append('15-30')
    elif 30.0 < row <= 50.0:
        intervals.append('30-50')
    elif 50.0 < row <= 100.0:
        intervals.append('50-100')
    elif 100.0 < row <= 600.0:
        intervals.append('100-600')
    else:
        intervals.append('600-28045')

merged_final['Precip_grouped'] = intervals
freq = merged_final['Precip_grouped'].value_counts()

capitals_keys = {'Alabama': 'Montgomery', 'Alaska': 'Juneau', 'Arizona': 'Maricopa',
'Arkansas': 'Pulaski','California': 'Sacramento', 'Colorado': 'Denver', 'Connecticut':
'Hartford', 'Delaware': 'Kent','Florida': 'Leon', 'Georgia': 'Fulton', 'Hawaii':
'Honolulu', 'Idaho': 'Ada', 'Illinois': 'Sangamon', 'Indiana': 'Marion', 'Iowa':
'Polk', 'Kansas': 'Shawnee', 'Kentucky': 'Jefferson', 'Louisiana': 'East Baton Rouge',
'Maine': 'Kennebec', 'Maryland': 'Anne Arundel', 'Massachusetts': 'Suffolk', 'Michigan':
'Ingham', 'Minnesota': 'Ramsey', 'Mississippi': 'Jackson', 'Missouri': 'Callaway',
'Montana': 'Lewis and Clark', 'Nebraska': 'Lancaster', 'Nevada': 'Carson City',
'New Hampshire': 'Merrimack', 'New Jersey': 'Mercer', 'New Mexico': 'Santa Fe',
'New York': 'Albany', 'North Carolina': 'Wake', 'North Dakota': 'Burleigh', 'Ohio':
'Franklin', 'Oklahoma': 'Oklahoma', 'Oregon': 'Marion', 'Pennsylvania': 'Dauphin',
'Rhode Island': 'Providence', 'South Carolina': 'Richland', 'South Dakota': 'Hughes',
'Tennessee': 'Davidson', 'Texas': 'Travis', 'Utah': 'Salt Lake', 'Vermont':
'Chittenden', 'Virginia': 'Chesterfield', 'Washington': 'Thurston', 'West Virginia':
'Kanawha', 'Wisconsin': 'Dane', 'Wyoming': 'Laramie'}
capitals = pd.DataFrame(capitals_keys, index=["Capital"])
capitals_T = capitals.T.reset_index()
unique_states = merged_final.drop_duplicates(subset=['State name'])
```

BNRC

```
states = unique_states['State name']
```

```
states_prob_df_BNRC = pd.DataFrame()
for state in states:
    state_data = merged_final.loc[merged_final['State name'] == state]
    location = state_data.drop_duplicates(subset=['county Name'])
    counties = location['county Name']
    county_prob_df = pd.DataFrame()
    for county in counties:
        county_data = state_data.loc[
            (state_data['County'] == county) & (state_data['Year'] != 2020)]
        prob_category = pd.crosstab(county_data['Category'], county_data['Year'],
        normalize='columns')

        years = county_data.drop_duplicates(subset=['Year'])
        year_list = years['Year']
        test_year = pd.Series('2019_pred')
        year_list_test = year_list.append(test_year)
        categories = county_data.drop_duplicates(subset=['Category'])
        category_list = categories['Category']

        if len(year_list) == 4:
            TW1 = (1 / 1) / (1 / 3 + 1 / 2 + 1 / 1)
            TW2 = (1 / 2) / (1 / 3 + 1 / 2 + 1 / 1)
            TW3 = (1 / 3) / (1 / 3 + 1 / 2 + 1 / 1)
            prob_category['2019_pred'] = TW3 * prob_category[2016] + TW2 *
            prob_category[2017] + TW1 * prob_category[2018]

            county_data_good = county_data.loc[county_data['Category'] == 'Good']
            county_data_moderate = county_data.loc[county_data['Category'] ==
            'Moderate']
            county_data_unhealthy = county_data.loc[county_data['Category'] ==
            'Unhealthy']

            prob_good = pd.crosstab(county_data_good['Precip_grouped'],
            county_data_good['Year'], normalize='columns')
            prob_moderate = pd.crosstab(county_data_moderate['Precip_grouped'],
            county_data_moderate['Year'],normalize='columns')
            prob_unhealthy = pd.crosstab(county_data_unhealthy['Precip_grouped'],
            county_data_unhealthy['Year'],normalize='columns')

            keys_category = {'Good': prob_good, 'Moderate': prob_moderate, 'Unhealthy':
            prob_unhealthy}
            prob_category_type = pd.concat([prob_good, prob_moderate, prob_unhealthy],
            keys=keys_category.keys())
            prob_category_type['2019_pred'] = TW3 * prob_category_type[2016] + TW2 *
            prob_category_type[2017] + TW1 * prob_category_type[2018]

            prob_df_1 = pd.DataFrame()
            for year in year_list_test:
                for category in category_list:
                    prob = prob_category_type.loc[category, year]
                    prob_category.loc[category, year]
                    prob_category_multiplied_type = pd.DataFrame(prob)
                    normalization_constant = 1 / prob_category_multiplied_type.sum()
                    prob_df_normalized = prob_category_multiplied_type *
                    normalization_constant
                    prob_df_normalized['Category'] = category
                    prob_df_1 = pd.concat([prob_df_1, prob_df_normalized])

            prob_df_index = prob_df_1.reset_index()
            prob_df_index = prob_df_index.set_index(['Category', 'Precip_grouped'],
            inplace=False)
            prob_df = prob_df_index.groupby(['Category', 'Precip_grouped']).sum()
            prob_df['County'] = county
            prob_df['State'] = state

            county_prob_df = pd.concat([county_prob_df, prob_df])
    states_prob_df_BNRC = pd.concat([states_prob_df_BNRC, county_prob_df])

BNRC_set = states_prob_df_BNRC.reset_index()
max_2019_pred = BNRC_set.groupby(['State', 'County', 'Category'], as_index=False,
sort=False)['2019_pred'].max()
BNRC_set_2019_pred = BNRC_set.merge(max_2019_pred, left_on=['State', 'County',
'Category', '2019_pred'], right_on=['State', 'County', 'Category', '2019_pred'])
BNRC_set_2019_pred = BNRC_set_2019_pred.drop([2016, 2017, 2018, 2019], axis=1)
BNRC_set_2019_pred_f = BNRC_set_2019_pred.drop_duplicates(subset=['State', 'County',
'Category'])

merged_probability = merged_final.merge(BNRC_set_2019_pred_f, how='left',
left_on=['State name', 'County', 'Category'], right_on=['State', 'County',
'Category'])

value = []
for row in merged_probability['Precip_grouped_y']:
    if row == '0':
        value.append(0.0)
    elif row == '0-0.3':
        value.append(0.15)
    elif row == '0.3-5':
        value.append(2.65)
    elif row == '5-15':
        value.append(10)
    elif row == '15-30':
        value.append(22.5)
    elif row == '30-50':
        value.append(40)
    elif row == '50-100':
        value.append(75)
    elif row == '100-600':
        value.append(350)
    else:
        value.append(1000)

merged_probability['Precip_interval_middle'] = value

final_prob_df_state = pd.DataFrame()
for state in states:
    state_data = merged_probability.loc[merged_probability['State name'] == state]
    location = state_data.drop_duplicates(subset=['county Name'])
    counties = location['county Name']
    final_prob_df_county = pd.DataFrame()
    for county in counties:
        county_data = state_data.loc[(state_data['County'] == county) &
        (state_data['Year'] != 2020)]
        county_data['MM-DD'] = county_data['StartTime(UTC)'].dt.strftime('%m-%d')
        categories = county_data.drop_duplicates(subset=['Category'])
        category_list = categories['Category']
        for category in category_list:
            category_df = county_data.loc[(county_data['Category'] == category)]
            years = category_df.drop_duplicates(subset=['Year'])
            year_list = years['Year']
            if len(year_list) == 4:
                pivot_table = category_df.pivot_table(values=['Precipitation(mm)',
                'Precip_interval_middle'], index=category_df['MM-DD'], columns='Year',
                aggfunc='first')
                pivot_table_2019 = pivot_table.dropna(subset=[('Precipitation(mm)',
                2019)])
                pivot_table_f = pivot_table_2019.fillna(0)

                pivot_table_f['E_2016'] = pivot_table_f[('Precipitation(mm)', 2016)] -
                pivot_table_f[('Precip_interval_middle', 2016)]
                pivot_table_f['epsilon_2017'] = pivot_table_f['E_2016']/2
                pivot_table_f['tuned_infer_2017'] =
                pivot_table_f[('Precip_interval_middle', 2017)] +
                pivot_table_f['epsilon_2017']
                pivot_table_f['E_2017'] = pivot_table_f[('Precipitation(mm)', 2017)] -
                pivot_table_f['tuned_infer_2017']
                pivot_table_f['epsilon_2018'] = pivot_table_f['E_2017']/2 +
                pivot_table_f['epsilon_2017']/2
                pivot_table_f['tuned_infer_2018'] =
                pivot_table_f[('Precip_interval_middle', 2018)] +
                pivot_table_f['epsilon_2018']
                pivot_table_f['E_2018'] = pivot_table_f[('Precipitation(mm)', 2018)] -
                pivot_table_f['tuned_infer_2018']
                pivot_table_f['epsilon_2019'] = pivot_table_f['E_2018']/2 +
                pivot_table_f['epsilon_2018']/2
                pivot_table_f['tuned_infer_2019'] =
                pivot_table_f[('Precip_interval_middle', 2019)] +
                pivot_table_f['epsilon_2019']
                final_prob_df_county = pd.concat([final_prob_df_county, pivot_table_f])
                final_prob_df_county['County'] = county
                final_prob_df_county['State'] = state
    final_prob_df_state = pd.concat([final_prob_df_state, final_prob_df_county])

BNRC_capitals_df = pd.DataFrame()
for capital in capitals_T['Capital']:
    set1 = final_prob_df_state.loc[final_prob_df_state['County'] == capital]
    BNRC_capitals_df = pd.concat([BNRC_capitals_df, set1])

r2_book_BNRC = rsquared_book(BNRC_capitals_df['tuned_infer_2019'],
BNRC_capitals_df[('Precipitation(mm)', 2019)])
r2_book_BNRC_2 = rsquared_book(final_prob_df_state['tuned_infer_2019'],
final_prob_df_state[('Precipitation(mm)', 2019)])
r2_MAE = mean_absolute_error(BNRC_capitals_df['tuned_infer_2019'],
BNRC_capitals_df[('Precipitation(mm)', 2019)])
r2_MAE_2 = mean_absolute_error(final_prob_df_state['tuned_infer_2019'],
final_prob_df_state[('Precipitation(mm)', 2019)])
rmse_BNRC = math.sqrt(mean_squared_error(BNRC_capitals_df['tuned_infer_2019'],
BNRC_capitals_df[('Precipitation(mm)', 2019)]))
rmse_BNRC_2 = math.sqrt(mean_squared_error(final_prob_df_state['tuned_infer_2019'],
final_prob_df_state[('Precipitation(mm)', 2019)]))
```

FBNRC

```
value = []
for row in merged_final['Precipitation(mm)']:
    if row <= 0.1:
        value.append('< 0.1')
    elif 0.1 < row <= 1:
        value.append('0.1-1')
    elif 2.5 < row <= 7.5:
        value.append('2.5-7.5')
    elif 10.0 < row <= 20.0:
        value.append('10-20')
    elif 20.0 < row <= 40.0:
        value.append('20-40')
    elif 40.0 < row <= 70.0:
        value.append('40-70')
    elif 80.0 < row <= 150.0:
        value.append('80-150')
    elif 500.0 < row <= 700.0:
        value.append('500-700')
    else:
        value.append('not fuzzy')

merged_final['Fuzzy_set'] = value

merged_final.loc[((merged_final['Fuzzy_set'] == '< 0.1') &
(merged_final['Precip_grouped'] != '0')), 'FS_fix'] = '0'
merged_final.loc[((merged_final['Fuzzy_set'] == '< 0.1') &
(merged_final['Precip_grouped'] != '0-0.3')), 'FS_fix'] = '0-0.3'
merged_final.loc[((merged_final['Fuzzy_set'] == '0.1-1') &
(merged_final['Precip_grouped'] != '0-0.3')), 'FS_fix'] = '0-0.3'
merged_final.loc[((merged_final['Fuzzy_set'] == '0.1-1') &
(merged_final['Precip_grouped'] != '0.3-5')), 'FS_fix'] = '0.3-5'
merged_final.loc[((merged_final['Fuzzy_set'] == '2.5-7.5') &
(merged_final['Precip_grouped'] != '0.3-5')), 'FS_fix'] = '0.3-5'
merged_final.loc[((merged_final['Fuzzy_set'] == '2.5-7.5') &
(merged_final['Precip_grouped'] != '5-15')), 'FS_fix'] = '5-15'
merged_final.loc[((merged_final['Fuzzy_set'] == '10-20') &
(merged_final['Precip_grouped'] != '5-15')), 'FS_fix'] = '5-15'
merged_final.loc[((merged_final['Fuzzy_set'] == '10-20') &
(merged_final['Precip_grouped'] != '15-30')), 'FS_fix'] = '15-30'
merged_final.loc[((merged_final['Fuzzy_set'] == '20-40') &
(merged_final['Precip_grouped'] != '15-30')), 'FS_fix'] = '15-30'
merged_final.loc[((merged_final['Fuzzy_set'] == '20-40') &
(merged_final['Precip_grouped'] != '30-50')), 'FS_fix'] = '30-50'
merged_final.loc[((merged_final['Fuzzy_set'] == '40-70') &
(merged_final['Precip_grouped'] != '30-50')), 'FS_fix'] = '30-50'
merged_final.loc[((merged_final['Fuzzy_set'] == '40-70') &
(merged_final['Precip_grouped'] != '50-100')), 'FS_fix'] = '50-100'
merged_final.loc[((merged_final['Fuzzy_set'] == '80-150') &
(merged_final['Precip_grouped'] != '50-100')), 'FS_fix'] = '50-100'
merged_final.loc[((merged_final['Fuzzy_set'] == '80-150') &
(merged_final['Precip_grouped'] != '50-100')), 'FS_fix'] = '100-600'
merged_final.loc[((merged_final['Fuzzy_set'] == '500-700') &
(merged_final['Precip_grouped'] != '50-100')), 'FS_fix'] = '100-600'

states = unique_states['State name']
states_prob_df_BNRC = pd.DataFrame()
for state in states:
    state_data = merged_final.loc[merged_final['State name'] == state]
    location = state_data.drop_duplicates(subset=['county Name'])
    counties = location['county Name']
    county_prob_df = pd.DataFrame()
    for county in counties:
        county_data = state_data.loc[
            (state_data['County'] == county) & (state_data['Year'] != 2020)]
        prob_category = pd.crosstab(county_data['Category'], county_data['Year'],
        normalize='columns')

        years = county_data.drop_duplicates(subset=['Year'])
        year_list = years['Year']
        test_year = pd.Series('2019_pred')
        year_list_test = year_list.append(test_year)
        categories = county_data.drop_duplicates(subset=['Category'])
        category_list = categories['Category']

        if len(year_list) == 4:
            TW1 = (1 / 1) / (1 / 3 + 1 / 2 + 1 / 1)
            TW2 = (1 / 2) / (1 / 3 + 1 / 2 + 1 / 1)
            TW3 = (1 / 3) / (1 / 3 + 1 / 2 + 1 / 1)
            prob_category['2019_pred'] = TW3 * prob_category[2016] + TW2 *
            prob_category[2017] + TW1 * prob_category[2018]

            county_data_good = county_data.loc[county_data['Category'] == 'Good']
            county_data_moderate = county_data.loc[county_data['Category'] ==
            'Moderate']
            county_data_unhealthy = county_data.loc[county_data['Category'] ==
            'Unhealthy']

            prob_good = pd.crosstab(county_data_good['Precip_grouped'],
            county_data_good['Year'], normalize='columns')
            prob_moderate = pd.crosstab(county_data_moderate['Precip_grouped'],
            county_data_moderate['Year'], normalize='columns')
            prob_unhealthy = pd.crosstab(county_data_unhealthy['Precip_grouped'],
            county_data_unhealthy['Year'], normalize='columns')

            keys_category = {'Good': prob_good, 'Moderate': prob_moderate,
            'Unhealthy': prob_unhealthy}
            prob_category_type = pd.concat([prob_good, prob_moderate,
            prob_unhealthy], keys=keys_category.keys())

            prob_good_fs = pd.crosstab(county_data_good['FS_fix'],
            county_data_good['Year'])/len(county_data_good)
            prob_moderate_fs = pd.crosstab(county_data_moderate['FS_fix'],
            county_data_moderate['Year'])/len(county_data_moderate)
            prob_unhealthy_fs = pd.crosstab(county_data_unhealthy['FS_fix'],
            county_data_unhealthy['Year'])/len(county_data_unhealthy)

            keys_category = {'Good': prob_good_fs, 'Moderate': prob_moderate_fs,
            'Unhealthy': prob_unhealthy_fs}
            prob_category_type_fs = pd.concat([prob_good_fs, prob_moderate_fs,
            prob_unhealthy_fs], keys=keys_category.keys())
```

```
prob_category_type_idx = prob_category_type.reset_index()
prob_category_type_fs_idx = prob_category_type_fs.reset_index()
prob_category_type_new =
prob_category_type_idx.merge(prob_category_type_fs_idx, how='left',
left_on=['level_0', 'Precip_grouped'], right_on=['level_0', 'FS_fix'])
prob_category_type_new[2016] = prob_category_type_new['2016_x'] +
prob_category_type_new['2016_y']
prob_category_type_new[2017] = prob_category_type_new['2017_x'] +
prob_category_type_new['2017_y']
prob_category_type_new[2018] = prob_category_type_new['2018_x'] +
prob_category_type_new['2018_y']
prob_category_type_new[2019] = prob_category_type_new['2019_x'] +
prob_category_type_new['2019_y']
category_type_df = pd.DataFrame(prob_category_type_new,
columns=['level_0', 'Precip_grouped', 2016, 2017, 2018, 2019])

normalized_df = pd.DataFrame()
for category in category_list:
    for_category = category_type_df.loc[category_type_df['level_0']
    == category]
    for_category = for_category.set_index(['level_0',
    'Precip_grouped'], inplace=False)
    normalization_constant_2 = 1 / for_category.sum()
    prob_ctgr_type = for_category * normalization_constant_2
    normalized_df = pd.concat([normalized_df, prob_ctgr_type])

normalized_df['2019_pred'] = TW3 * normalized_df[2016] + TW2 *
normalized_df[2017] + TW1 * normalized_df[2018]

prob_df_1 = pd.DataFrame()
for year in year_list_test:
    for category in category_list:
        prob = normalized_df.loc[category, year] *
        prob_category.loc[category, year]
        prob_category_multiplied_type = pd.DataFrame(prob)
        normalization_constant = 1 / prob_category_multiplied_type.sum()
        prob_df_normalized = prob_category_multiplied_type *
        normalization_constant
        prob_df_normalized['Category'] = category
        prob_df_1 = pd.concat([prob_df_1, prob_df_normalized])

prob_df_index = prob_df_1.reset_index()
prob_df_index = prob_df_index.set_index(['Category', 'Precip_grouped'],
inplace=False)
prob_df = prob_df_index.groupby(['Category', 'Precip_grouped']).sum()
prob_df['County'] = county
```

```
            prob_df['State'] = state

            county_prob_df = pd.concat([county_prob_df, prob_df])
    states_prob_df_BNRC = pd.concat([states_prob_df_BNRC, county_prob_df])

BNRC_set = states_prob_df_BNRC.reset_index()
max_2019_pred = BNRC_set.groupby(['State', 'County', 'Category'], as_index=False,
sort=False)['2019_pred'].max()
BNRC_set_2019_pred = BNRC_set.merge(max_2019_pred, left_on=['State', 'County',
'Category', '2019_pred'], right_on=['State', 'County', 'Category', '2019_pred'])
BNRC_set_2019_pred = BNRC_set_2019_pred.drop([2016, 2017, 2018, 2019], axis=1)
BNRC_set_2019_pred_f = BNRC_set_2019_pred.drop_duplicates(subset=['State', 'County',
'Category'])

merged_probability = merged_final.merge(BNRC_set_2019_pred_f, how='left',
left_on=['State name', 'County', 'Category'], right_on=['State', 'County',
'Category'])

value = []
for row in merged_probability['Precip_grouped_y']:
    if row == '0':
        value.append(0)
    elif row == '0-0.3':
        value.append(0.15)
    elif row == '0.3-5':
        value.append(2.65)
    elif row == '5-15':
        value.append(10)
    elif row == '15-30':
        value.append(22.5)
    elif row == '30-50':
        value.append(40)
    elif row == '50-100':
        value.append(75)
    elif row == '100-600':
        value.append(350)
    else:
        value.append(1000)

merged_probability['Precip_interval_middle'] = value

final_prob_df_state = pd.DataFrame()
for state in states:
    state_data = merged_probability.loc[merged_probability['State name'] == state]
    location = state_data.drop_duplicates(subset=['county Name'])
    counties = location['county Name']
    final_prob_df_county = pd.DataFrame()
```

```
for county in counties:
    county_data = state_data.loc[(state_data['County'] == county) &
    (state_data['Year'] != 2020)]
    county_data['MM-DD'] = county_data['StartTime(UTC)'].dt.strftime('%m-%d')
    categories = county_data.drop_duplicates(subset=['Category'])
    category_list = categories['Category']
    for category in category_list:
        category_df = county_data.loc[(county_data['Category'] == category)]
        years = category_df.drop_duplicates(subset=['Year'])
        year_list = years['Year']
        if len(year_list) == 4:
            pivot_table = category_df.pivot_table(values=['Precipitation(mm)',
            'Precip_interval_middle'], index=category_df['MM-DD'], columns='Year',
            aggfunc='first')
            pivot_table_2019 = pivot_table.dropna(subset=[('Precipitation(mm)',
            2019)])
            pivot_table_f = pivot_table_2019.fillna(0)

            pivot_table_f['E_2016'] = pivot_table_f[('Precipitation(mm)', 2016)] -
            pivot_table_f[('Precip_interval_middle', 2016)]
            pivot_table_f['epsilon_2017'] = pivot_table_f['E_2016']/2
            pivot_table_f['tuned_infer_2017'] =
            pivot_table_f[('Precip_interval_middle', 2017)] +
            pivot_table_f['epsilon_2017']
            pivot_table_f['E_2017'] = pivot_table_f[('Precipitation(mm)', 2017)] -
            pivot_table_f['tuned_infer_2017']
            pivot_table_f['epsilon_2018'] = pivot_table_f['E_2017']/2 +
            pivot_table_f['epsilon_2017']/2
            pivot_table_f['tuned_infer_2018'] =
            pivot_table_f[('Precip_interval_middle', 2018)] +
            pivot_table_f['epsilon_2018']
            pivot_table_f['E_2018'] = pivot_table_f[('Precipitation(mm)', 2018)] -
            pivot_table_f['tuned_infer_2018']
            pivot_table_f['epsilon_2019'] = pivot_table_f['E_2018']/2 +
            pivot_table_f['epsilon_2018']/2
            pivot_table_f['tuned_infer_2019'] =
            pivot_table_f[('Precip_interval_middle', 2019)] +
            pivot_table_f['epsilon_2019']
            final_prob_df_county = pd.concat([final_prob_df_county, pivot_table_f])
            final_prob_df_county['County'] = county
            final_prob_df_county['State'] = state
        if len(year_list) == 3 and year_list.iloc[0] == 2016 and year_list.iloc[1]
         == 2017 and year_list.iloc[2] == 2018:
            pivot_table = category_df.pivot_table(values=['Precipitation(mm)',
            'Precip_interval_middle'], index=category_df['MM-DD'], columns='Year',
            aggfunc='first')
            pivot_table_2019 = pivot_table.dropna(subset=[('Precipitation(mm)',
                2018)])
                pivot_table_f = pivot_table_2019.fillna(0)

                pivot_table_f['E_2016'] = pivot_table_f[('Precipitation(mm)', 2016)] -
                pivot_table_f[('Precip_interval_middle', 2016)]
                pivot_table_f['epsilon_2017'] = pivot_table_f['E_2016']/2
                pivot_table_f['tuned_infer_2017'] =
                pivot_table_f[('Precip_interval_middle', 2017)] +
                pivot_table_f['epsilon_2017']
                pivot_table_f['E_2017'] = pivot_table_f[('Precipitation(mm)', 2017)] -
                pivot_table_f['tuned_infer_2017']
                pivot_table_f['epsilon_2018'] = pivot_table_f['E_2017']/2 +
                pivot_table_f['epsilon_2017']/2
                pivot_table_f['tuned_infer_2018'] =
                pivot_table_f[('Precip_interval_middle', 2018)] +
                pivot_table_f['epsilon_2018']
                final_prob_df_county = pd.concat([final_prob_df_county, pivot_table_f])
                final_prob_df_county['County'] = county
                final_prob_df_county['State'] = state
    final_prob_df_state = pd.concat([final_prob_df_state, final_prob_df_county])
    final_prob = final_prob_df_state.fillna(0)

BNRC_capitals_df = pd.DataFrame()
for capital in capitals_T['Capital']:
    set1 = final_prob.loc[final_prob['County'] == capital]
    if len(set1) != 0:
        r_capital = r2_score(set1['tuned_infer_2019'], set1[('Precipitation(mm)', 2019)])
        print(r_capital)
    BNRC_capitals_df = pd.concat([BNRC_capitals_df, set1])


r2_book_FBNRC = rsquared_book(BNRC_capitals_df['tuned_infer_2019'],
BNRC_capitals_df[('Precipitation(mm)', 2019)])
r2_book_FBNRC_2 = rsquared_book(final_prob_df_state['tuned_infer_2019'],
final_prob_df_state[('Precipitation(mm)', 2019)])
r2_MAE_FBNRC = mean_absolute_error(BNRC_capitals_df['tuned_infer_2019'],
BNRC_capitals_df[('Precipitation(mm)', 2019)])
r2_MAE_FBNRC_2 = mean_absolute_error(final_prob['tuned_infer_2019'],
final_prob[('Precipitation(mm)', 2019)])
rmse_FBNRC = math.sqrt(mean_squared_error(BNRC_capitals_df['tuned_infer_2019'],
BNRC_capitals_df[('Precipitation(mm)', 2019)]))
rmse_FBNRC_2 = math.sqrt(mean_squared_error(final_prob_df_state['tuned_infer_2019'],
final_prob_df_state[('Precipitation(mm)', 2019)]))
```

SpaBN

```
states_prob_precip_df = pd.DataFrame()
```

```
capital_aqi_df = pd.DataFrame()
for state in states:
    state_data = merged_final.loc[(merged_final['State name'] == state) &
    (merged_final['Year'] != 2020)]
    capital = capitals_T.loc[capitals_T['index'] == state, ['Capital']]
    capital_value = capital.iloc[0, 0]
    capital_data = state_data.loc[state_data['county Name'] == capital_value]
    capital_aqi = aqi_df.loc[aqi_df['county Name'] == capital_value, ['Date', 'AQI']]
    capital_precipitation = capital_data.groupby('Date', as_index=False)
    ['Precipitation(mm)'].sum()
    location = state_data.drop_duplicates(subset=['county Name'])
    counties = location['county Name']
    distance_df = pd.DataFrame()
    location = state_data.drop_duplicates(subset=['county Name'])
    distance = location[['LocationLat', 'LocationLng', 'county Name',
    'State name']].sort_values(by=['LocationLng'])
    distance["lat_radians"] = np.radians(distance["LocationLat"])
    distance["lng_radians"] = np.radians(distance["LocationLng"])
    lat = distance.loc[distance['county Name'] == capital_value, ['LocationLat']]
    lng = distance.loc[distance['county Name'] == capital_value, ['LocationLng']]
    for row in distance.index:
        distance.loc[row, 'Capital_lat_rad'] = lat['LocationLat'].values[0]
        distance.loc[row, 'Capital_lng_rad'] = lng['LocationLng'].values[0]
    a = np.sin((distance["lat_radians"] - distance["Capital_lat_rad"]) / 2.0) ** 2
    + np.cos(distance["Capital_lat_rad"]) * np.cos(distance["lat_radians"]) *
    np.sin((distance["lng_radians"] - distance["Capital_lng_rad"]) / 2.0) ** 2
    distance['Distance'] = 6371 * 2 * np.arcsin(np.sqrt(a))
    distance["Inverse Distance"] = 1 / distance["Distance"]
    distance.loc[distance['county Name'] == capital_value, 'Inverse Distance'] = 0
    distance['NISD'] = distance['Inverse Distance'] / sum(distance['Inverse Distance'])
    distance_df = pd.concat([distance_df, distance['county Name'], distance['NISD'],
    distance['State name']], axis=1)

    corr_category_df = pd.DataFrame()
    corr_precip_df = pd.DataFrame()
    corr_category_county = pd.DataFrame()
    for county in counties:
        county_data = state_data.loc[(state_data['County'] == county) &
        (state_data['Year'] != 2020)]
        county_aqi = aqi_df.loc[aqi_df['county Name'] == county, ['Date', 'AQI']]
        county_aqi['AQI_county'] = county_aqi['AQI']
        merged_categories = capital_aqi.merge(county_aqi, left_on=['Date'],
        right_on=['Date'])
        corr_category, _ = stats.pearsonr(merged_categories['AQI_x'],
        merged_categories['AQI_county'])
        corr_category_county = pd.DataFrame([corr_category])
        corr_category_county['County'] = county
        corr_category_df = pd.concat([corr_category_df, corr_category_county])

        county_precipitation = county_data.groupby('Date', as_index=False)
        ['Precipitation(mm)'].sum()
        county_precipitation['Precipitation_county'] =
        county_precipitation['Precipitation(mm)']
        merged_precip = capital_precipitation.merge(county_precipitation,
        left_on=['Date'], right_on=['Date'])
        corr_precip, _ = stats.pearsonr(merged_precip['Precipitation(mm)_x'],
        merged_precip['Precipitation_county'])
        corr_precip_county = pd.DataFrame([corr_precip])
        corr_precip_county['County'] = county
        corr_precip_df = pd.concat([corr_precip_df, corr_precip_county])

    corr_category_df.loc[corr_category_df['County'] == capital_value, 0] = 0
    corr_category_df['NCorr_ctgr'] = corr_category_df[0] / np.nansum(corr_category_df[0])
    corr_precip_df.loc[corr_precip_df['County'] == capital_value, 0] = 0
    corr_precip_df['NCorr_precip'] = corr_precip_df[0] / np.nansum(corr_precip_df[0])
    merged_SW_1 = distance_df.merge(corr_category_df, left_on=['county Name'],
    right_on=['County'])
    merged_SW_2 = merged_SW_1.merge(corr_precip_df, left_on=['county Name'],
    right_on=['County'])
    merged_SW_2['SW'] = (merged_SW_2['NCorr_ctgr'] + merged_SW_2['NCorr_precip'] +
    merged_SW_2['NISD']) / np.nansum(merged_SW_2['NCorr_ctgr'] +
    merged_SW_2['NCorr_precip'] + merged_SW_2['NISD'])
    merged_SW_2.loc[merged_SW_2['county Name'] == capital_value, 'SW'] = 1

    prob_category_df = pd.DataFrame()
    prob_ctgr_precip_df = pd.DataFrame()
    for county in counties:
        county_data = state_data.loc[(state_data['County'] == county) &
        (state_data['Year'] != 2020)]
        sw = merged_SW_2['SW'].loc[merged_SW_2['county Name'] == county].values[0]

        prob_category = pd.crosstab(county_data['Category'], county_data['Year'],
        normalize='columns')
        prob_category = prob_category * sw
        prob_category_df = pd.concat([prob_category_df, prob_category])

        county_data_good = county_data.loc[county_data['Category'] == 'Good']
        county_data_moderate = county_data.loc[county_data['Category'] ==
        'Moderate']
        county_data_unhealthy = county_data.loc[county_data['Category'] ==
        'Unhealthy']

        prob_good = pd.crosstab(county_data_good['Precip_grouped'],
        county_data_good['Year'], normalize='columns')
        prob_moderate = pd.crosstab(county_data_moderate['Precip_grouped'],
        county_data_moderate['Year'], normalize='columns')
        prob_unhealthy = pd.crosstab(county_data_unhealthy['Precip_grouped'],
        county_data_unhealthy['Year'], normalize='columns')

        keys_category = {'Good': prob_good, 'Moderate': prob_moderate, 'Unhealthy':
        prob_unhealthy}
        prob_category_precip = pd.concat([prob_good, prob_moderate, prob_unhealthy],
        keys=keys_category.keys())
        prob_category_precip = prob_category_precip * sw
        prob_ctgr_precip_df = pd.concat([prob_ctgr_precip_df, prob_category_precip])

    prob_ctgr_precip_index = prob_ctgr_precip_df.reset_index()
    prob_ctgr_precip_index = prob_ctgr_precip_index.set_index(['level_0',
    'Precip_grouped'], inplace=False)

    prob_category_df_sum = prob_category_df.groupby(['Category']).sum()
    normalization_constant_1 = 1 / prob_category_df_sum.sum()
    prob_category_df_multiplied = prob_category_df_sum * normalization_constant_1

    prob_ctgr_precip_df_sum = prob_ctgr_precip_index.groupby(['level_0',
    'Precip_grouped']).sum()
    prob_ctgr_precip_df_sum_indx = prob_ctgr_precip_df_sum.reset_index()
    category_list = pd.Series(prob_category_df_sum.index)
    normalized_df = pd.DataFrame()
    for category in category_list:
        for_category =
        prob_ctgr_precip_df_sum_indx.loc[prob_ctgr_precip_df_sum_indx['level_0'] ==
        category]
        for_category = for_category.set_index(['level_0', 'Precip_grouped'],
        inplace=False)
        normalization_constant_2 = 1 / for_category.sum()
        prob_ctgr_precip_df_multiplied = for_category * normalization_constant_2
        normalized_df = pd.concat([normalized_df, prob_ctgr_precip_df_multiplied])

    year_list = pd.Series(prob_category_df_sum.columns)
    test_year = pd.Series(['2019_pred'])
    year_list_test = pd.concat([year_list, test_year])

    if len(year_list) == 4:
        TW1 = (1 / 1) / (1 / 3 + 1 / 2 + 1 / 1)
        TW2 = (1 / 2) / (1 / 3 + 1 / 2 + 1 / 1)
        TW3 = (1 / 3) / (1 / 3 + 1 / 2 + 1 / 1)
        TW = (1 / 3)
        prob_category_df_multiplied['2019_pred'] = TW3 *
        prob_category_df_multiplied[2016] + TW2 * prob_category_df_multiplied[2017]
        + TW1 * prob_category_df_multiplied[2018]
```

```
        normalized_df['2019_pred'] = TW3 * normalized_df[2016] + TW2 *
                                     normalized_df[2017] + TW1 *
                                     normalized_df[2018]

        prob_df_1 = pd.DataFrame()
        for year in year_list_test:
            for category in category_list:
                prob = prob_category_df_multiplied.loc[category, year] *
                normalized_df.loc[category, year]
                prob_category_multiplied_type = pd.DataFrame(prob)
                normalization_constant = 1 / prob_category_multiplied_type.sum()
                prob_df_normalized = prob_category_multiplied_type *
                normalization_constant
                prob_df_normalized['Category'] = category
                prob_df_1 = pd.concat([prob_df_1, prob_df_normalized])

        prob_df_index = prob_df_1.reset_index()
        prob_df_index = prob_df_index.set_index(['Category', 'Precip_grouped'],
        inplace=False)
        prob_df = prob_df_index.groupby(['Category', 'Precip_grouped']).sum()
        prob_df['State'] = state
        prob_df['County'] = capital_value

    states_prob_precip_df = pd.concat([states_prob_precip_df, prob_df])

SpaBN_set = states_prob_precip_df.reset_index()
max_2019_pred = SpaBN_set.groupby(['State', 'County', 'Category'], as_index=False,
sort=False)['2019_pred'].max()
SpaBN_set_2019_pred = SpaBN_set.merge(max_2019_pred, left_on=['State', 'County',
'Category', '2019_pred'], right_on=['State', 'County', 'Category', '2019_pred'])
SpaBN_set_2019_pred = SpaBN_set_2019_pred.drop([2016, 2017, 2018, 2019], axis=1)
SpaBN_set_2019_pred_f = SpaBN_set_2019_pred.drop_duplicates(subset=['State', 'County',
'Category'])

merged_probability = merged_final.merge(SpaBN_set_2019_pred_f, how='left',
left_on=['State name', 'County', 'Category'], right_on=['State', 'County', 'Category'])

value = []
for row in merged_probability['Precip_grouped_y']:
    if row == '0':
        value.append(0.0)
    elif row == '0-0.3':
        value.append(0.15)
    elif row == '0.3-5':
        value.append(2.65)
    elif row == '5-15':
        value.append(10)
    elif row == '15-30':
        value.append(22.5)
    elif row == '30-50':
        value.append(40)
    elif row == '50-100':
        value.append(75)
    elif row == '100-600':
        value.append(350)
    else:
        value.append(1000)

merged_probability['Precip_interval_middle'] = value
merged_final_df = merged_probability.loc[(merged_probability['Year'] == 2019)]
merged_final_df = merged_final_df.dropna(subset=['2019_pred'])
r2_book_SpaBN = rsquared_book(merged_final_df['Precipitation(mm)'],
merged_final_df['Precip_interval_middle'])
MAE_SpaBN = mean_absolute_error(merged_final_df['Precipitation(mm)'],
merged_final_df['Precip_interval_middle'])
rmse_SpaBN = math.sqrt(mean_squared_error(merged_final_df['Precipitation(mm)'],
merged_final_df['Precip_interval_middle']))
```

SpaBN with DBSCAN

```
cols = ['#46f0f0', '#e6194b', '#3cb44b', '#ffe119', '#4363d8', '#f58231', '#911eb4',
        '#f032e6', '#bcf60c', '#fabebe', '#008080', '#e6beff',
        '#9a6324', '#fffac8', '#800000', '#aaffc3', '#808000', '#ffd8b1',
        '#000075', '#808080'] * 10000

sns.set(style="white")

df = merged_final.drop_duplicates(subset=['county Name', 'LocationLat', 'LocationLng'])

X = np.array(df[['LocationLng', 'LocationLat']], dtype='float64')
plt.scatter(X[:, 0], X[:, 1], alpha=0.2, s=50)

m = folium.Map(location=[df.LocationLat.mean(), df.LocationLng.mean()], zoom_start=9)
# tiles='OpenStreet Map')
for _, row in df.iterrows():
    folium.CircleMarker(
        location=[row.LocationLat, row.LocationLng],
        radius=5,
        # popup=re.sub(r'[^a-zA-Z ]+', '', row.MMSI),
        color='#1787FE',
        fill=True,
        fill_colour='#1787FE'
    ).add_to(m)
```

```
m.save("mymap.html")


def create_map(df, cluster_column):
    m = folium.Map(location=[df.LocationLat.mean(), df.LocationLng.mean()],
    zoom_start=9)  # tiles='OpenStreet Map')
    for _, row in df.iterrows():
        if row[cluster_column] == -1:
            cluster_colour = '#000000'
        else:
            cluster_colour = cols[row[cluster_column]]
        folium.CircleMarker(
            location=[row['LocationLat'], row['LocationLng']],
            radius=5,
            popup=row[cluster_column],
            color=cluster_colour,
            fill=True,
            fill_color=cluster_colour
        ).add_to(m)

    return m


model = DBSCAN(eps=0.5, min_samples=1).fit(X)
class_predictions = model.labels_
df['CLUSTERS_DBSCAN'] = class_predictions
m = create_map(df, 'CLUSTERS_DBSCAN')
print(f'Number of clusters found: {len(np.unique(class_predictions))}')
print(f'Number of outliers found: {len(class_predictions[class_predictions == -1])}')
print(f'Silhouette ignoring outliers: {silhouette_score(X[class_predictions != -1],
class_predictions[class_predictions != -1])}')
no_outliers = 0
no_outliers = np.array([(counter + 2) * x if x == -1 else x for counter,
x in enumerate(class_predictions)])
print(f'Silhouette outliers as singletons: {silhouette_score(X, no_outliers)}')
m.save("mymap_clusters.html")

capital_counties = pd.DataFrame()
for state in states:
    state_data = merged_final.loc[(merged_final['State name'] == state) &
    (merged_final['Year'] != 2020)]
    capital = capitals_T.loc[capitals_T['index'] == state, ['Capital']]
    capital_value = capital.iloc[0, 0]
    capital_data = state_data.loc[state_data['county Name'] == capital_value]
    capital_counties = pd.concat([capital_counties, capital_data])

capital_counties2 = capital_counties.drop_duplicates(subset=['State name'])
```

```
m = create_map(capital_counties2, 'State Code')
m.save("counties.html")

states_precip_dbscan = pd.DataFrame()
max_class = max(class_predictions)
for class1 in range(0, max_class):
    class_data = df.loc[df['CLUSTERS_DBSCAN'] == class1]
    state_class_data = class_data.drop_duplicates(subset=['State Name'])
    state_class = state_class_data['State Name'].values[0]
    capital = capitals_T.loc[capitals_T['index'] == state_class, ['Capital']].values
    county_class_data = class_data['county Name'].values
    if capital in county_class_data:
        distance_df = pd.DataFrame()
        distance = class_data[['LocationLat', 'LocationLng', 'county Name',
        'State name']].sort_values(
            by=['LocationLng'])
        counties = distance['county Name']
        distance["lat_radians"] = np.radians(distance["LocationLat"])
        distance["lng_radians"] = np.radians(distance["LocationLng"])
        capital = capitals_T.loc[capitals_T['index'] == state_class, ['Capital']]
        capital_value = capital.iloc[0, 0]
        lat = distance.loc[distance['county Name'] == capital_value, ['LocationLat']]
        lng = distance.loc[distance['county Name'] == capital_value, ['LocationLng']]
        for row in distance.index:
            distance.loc[row, 'Capital_lat_rad'] = lat['LocationLat'].values[0]
            distance.loc[row, 'Capital_lng_rad'] = lng['LocationLng'].values[0]
        a = np.sin((distance["lat_radians"] - distance["Capital_lat_rad"]) / 2.0) ** 2 +
        np.cos(distance["Capital_lat_rad"]) * np.cos(distance["lat_radians"]) *
        np.sin((distance["lng_radians"] - distance["Capital_lng_rad"]) / 2.0) ** 2
        distance['Distance'] = 6371 * 2 * np.arcsin(np.sqrt(a))
        distance["Inverse Distance"] = 1 / distance["Distance"]
        distance.loc[distance['county Name'] == capital_value, 'Inverse Distance'] = 0
        distance['NISD'] = distance['Inverse Distance'] /
        sum(distance['Inverse Distance'])
        distance_df = pd.concat([distance_df, distance['county Name'], distance['NISD'],
        distance['State name']],axis=1)
        capital_aqi = aqi_df.loc[aqi_df['county Name'] == capital_value, ['Date', 'AQI']]
        capital_data = merged_final.loc[merged_final['county Name'] == capital_value]
        capital_precipitation = capital_data.groupby('Date',
        as_index=False)['Precipitation(mm)'].sum()

        corr_category_df = pd.DataFrame()
        corr_precip_df = pd.DataFrame()
        corr_category_county = pd.DataFrame()
        for county in counties:
            county_data = merged_final.loc[(merged_final['County'] == county) &
            (merged_final['Year'] != 2020)]
            county_aqi = aqi_df.loc[aqi_df['county Name'] == county, ['Date', 'AQI']]
            county_aqi['AQI_county'] = county_aqi['AQI']
            merged_categories = capital_aqi.merge(county_aqi, left_on=['Date'],
            right_on=['Date'])
            corr_category, _ = stats.pearsonr(merged_categories['AQI_x'],
            merged_categories['AQI_county'])
            corr_category_county = pd.DataFrame([corr_category])
            corr_category_county['County'] = county
            corr_category_df = pd.concat([corr_category_df, corr_category_county])

            county_precipitation = county_data.groupby('Date', as_index=False)
            ['Precipitation(mm)'].sum()
            county_precipitation['Precipitation_county'] =
            county_precipitation['Precipitation(mm)']
            merged_precip = capital_precipitation.merge(county_precipitation,
            left_on=['Date'], right_on=['Date'])
            corr_precip, _ = stats.pearsonr(merged_precip['Precipitation(mm)_x'],
            merged_precip['Precipitation_county'])
            corr_precip_county = pd.DataFrame([corr_precip])
            corr_precip_county['County'] = county
            corr_precip_df = pd.concat([corr_precip_df, corr_precip_county])

        corr_category_df.loc[corr_category_df['County'] == capital_value, 0] = 0
        corr_category_df['NCorr_ctgr'] = corr_category_df[0] /
        np.nansum(corr_category_df[0])
        corr_precip_df.loc[corr_precip_df['County'] == capital_value, 0] = 0
        corr_precip_df['NCorr_precip'] = corr_precip_df[0] /
        np.nansum(corr_precip_df[0])
        merged_SW_1 = distance_df.merge(corr_category_df, left_on=['county Name'],
        right_on=['County'])
        merged_SW_2 = merged_SW_1.merge(corr_precip_df, left_on=['county Name'],
        right_on=['County'])
        merged_SW_2['SW'] = (merged_SW_2['NCorr_ctgr'] + merged_SW_2['NCorr_precip']
        + merged_SW_2['NISD']) / np.nansum(merged_SW_2['NCorr_ctgr'] +
        merged_SW_2['NCorr_precip'] + merged_SW_2['NISD'])
        merged_SW_2.loc[merged_SW_2['county Name'] == capital_value, 'SW'] = 1

        prob_category_df = pd.DataFrame()
        prob_ctgr_precip_df = pd.DataFrame()
        for county in counties:
            county_data = merged_final.loc[(merged_final['County'] == county) &
            (merged_final['Year'] != 2020)]
            sw = merged_SW_2['SW'].loc[merged_SW_2['county Name'] == county].values[0]

            prob_category = pd.crosstab(county_data['Category'], county_data['Year'],
            normalize='columns')
            prob_category = prob_category * sw
            prob_category_df = pd.concat([prob_category_df, prob_category])

            county_data_good = county_data.loc[county_data['Category'] == 'Good']
            county_data_moderate = county_data.loc[county_data['Category'] ==
            'Moderate']
            county_data_unhealthy = county_data.loc[county_data['Category'] ==
            'Unhealthy']

            prob_good = pd.crosstab(county_data_good['Precip_grouped'],
            county_data_good['Year'], normalize='columns')
            prob_moderate = pd.crosstab(county_data_moderate['Precip_grouped'],
            county_data_moderate['Year'], normalize='columns')
            prob_unhealthy = pd.crosstab(county_data_unhealthy['Precip_grouped'],
            county_data_unhealthy['Year'], normalize='columns')

            keys_category = {'Good': prob_good, 'Moderate': prob_moderate,
            'Unhealthy': prob_unhealthy}
            prob_category_precip = pd.concat([prob_good, prob_moderate,
            prob_unhealthy], keys=keys_category.keys())
            prob_category_precip = prob_category_precip * sw
            prob_ctgr_precip_df = pd.concat([prob_ctgr_precip_df, prob_category_precip])

        prob_ctgr_precip_index = prob_ctgr_precip_df.reset_index()
        prob_ctgr_precip_index = prob_ctgr_precip_index.set_index(['level_0',
        'Precip_grouped'], inplace=False)

        prob_category_df_sum = prob_category_df.groupby(['Category']).sum()
        normalization_constant_1 = 1 / prob_category_df_sum.sum()
        prob_category_df_multiplied = prob_category_df_sum * normalization_constant_1

        prob_ctgr_precip_df_sum = prob_ctgr_precip_index.groupby(['level_0',
        'Precip_grouped']).sum()
        prob_ctgr_precip_df_sum_indx = prob_ctgr_precip_df_sum.reset_index()
        category_list = pd.Series(prob_category_df_sum.index)
        normalized_df = pd.DataFrame()
        for category in category_list:
            for_category =
            prob_ctgr_precip_df_sum_indx.loc[prob_ctgr_precip_df_sum_indx['level_0']
            == category]
            for_category = for_category.set_index(['level_0', 'Precip_grouped'],
            inplace=False)
            normalization_constant_2 = 1 / for_category.sum()
            prob_ctgr_precip_df_multiplied = for_category * normalization_constant_2
            normalized_df = pd.concat([normalized_df, prob_ctgr_precip_df_multiplied])

        year_list = pd.Series(prob_category_df_sum.columns)
```

```
        test_year = pd.Series(['2019_pred'])
        year_list_test = pd.concat([year_list, test_year])
        category_list = pd.Series(prob_category_df_sum.index)

        if len(year_list) == 4:
            TW1 = (1 / 1) / (1 / 3 + 1 / 2 + 1 / 1)
            TW2 = (1 / 2) / (1 / 3 + 1 / 2 + 1 / 1)
            TW3 = (1 / 3) / (1 / 3 + 1 / 2 + 1 / 1)
            TW = (1 / 3)
            prob_category_df_multiplied['2019_pred'] = TW3 *
            prob_category_df_multiplied[2016] + TW2 * prob_category_df_multiplied[2017]
            + TW1 * prob_category_df_multiplied[2018]
            normalized_df['2019_pred'] = TW3 * normalized_df[2016] + TW2 * \
                                         normalized_df[2017] + TW1 * \
                                         normalized_df[2018]

            prob_df_1 = pd.DataFrame()
            for year in year_list_test:
                for category in category_list:
                    prob = prob_category_df_multiplied.loc[category, year] *
                    normalized_df.loc[category, year]
                    prob_category_multiplied_type = pd.DataFrame(prob)
                    normalization_constant = 1 / prob_category_multiplied_type.sum()
                    prob_df_normalized = prob_category_multiplied_type *
                    normalization_constant
                    prob_df_normalized['Category'] = category
                    prob_df_1 = pd.concat([prob_df_1, prob_df_normalized])

            prob_df_index = prob_df_1.reset_index()
            prob_df_index = prob_df_index.set_index(['Category', 'Precip_grouped'],
            inplace=False)
            prob_df = prob_df_index.groupby(['Category', 'Precip_grouped']).sum()
            prob_df['State'] = state_class
            prob_df['County'] = capital_value

        states_precip_dbscan = pd.concat([states_precip_dbscan, prob_df])

SpaBN_DBSCAN_set = states_precip_dbscan.reset_index()
max_2019_pred = SpaBN_DBSCAN_set.groupby(['State', 'County', 'Category'],
as_index=False, sort=False)['2019_pred'].max()
DBSCAN_set_2019_pred = SpaBN_DBSCAN_set.merge(max_2019_pred,
left_on=['State', 'County', 'Category', '2019_pred'],
right_on=['State', 'County', 'Category', '2019_pred'])
DBSCAN_set_2019_pred = DBSCAN_set_2019_pred.drop([2016, 2017, 2018, 2019], axis=1)
DBSCAN_set_2019_pred_f = DBSCAN_set_2019_pred.drop_duplicates(subset=['State',
'County', 'Category'])
```

```
merged_probability = merged_final.merge(DBSCAN_set_2019_pred_f, how='left',
                                       left_on=['State name', 'County', 'Category'],
                                       right_on=['State', 'County', 'Category'])

value = []
for row in merged_probability['Precip_grouped_y']:
    if row == '0':
        value.append(0.0)
    elif row == '0-0.3':
        value.append(0.15)
    elif row == '0.3-5':
        value.append(2.65)
    elif row == '5-15':
        value.append(10)
    elif row == '15-30':
        value.append(22.5)
    elif row == '30-50':
        value.append(40)
    elif row == '50-100':
        value.append(75)
    elif row == '100-600':
        value.append(350)
    else:
        value.append(1000)

merged_probability['Precip_interval_middle'] = value
merged_final_df = merged_probability.loc[(merged_probability['Year'] == 2019)]
merged_final_df = merged_final_df.dropna(subset=['2019_pred'])
r2_book_DBSCAN = rsquared_book(merged_final_df['Precipitation(mm)'],
merged_final_df['Precip_interval_middle'])
MAE_DBSCAN = mean_absolute_error(merged_final_df['Precipitation(mm)'],
merged_final_df['Precip_interval_middle'])
rmse_DBSCAN = math.sqrt(mean_squared_error(merged_final_df['Precipitation(mm)'],
merged_final_df['Precip_interval_middle']))
```

SpaFBN

```
value = []
for row in merged_final['Precipitation(mm)']:
    if row <= 0.1:
        value.append('< 0.1')
    elif 0.1 < row <= 1:
        value.append('0.1-1')
    elif 2.5 < row <= 7.5:
        value.append('2.5-7.5')
    elif 10.0 < row <= 20.0:
        value.append('10-20')
    elif 20.0 < row <= 40.0:
        value.append('20-40')
    elif 40.0 < row <= 70.0:
        value.append('40-70')
    elif 80.0 < row <= 150.0:
        value.append('80-150')
    elif 500.0 < row <= 700.0:
        value.append('500-700')
    else:
        value.append('not fuzzy')

merged_final['Fuzzy_set'] = value

merged_final.loc[((merged_final['Fuzzy_set'] == '< 0.1') &
(merged_final['Precip_grouped'] != '0')), 'FS_fix'] = '0'
merged_final.loc[((merged_final['Fuzzy_set'] == '< 0.1') &
(merged_final['Precip_grouped'] != '0-0.3')), 'FS_fix'] = '0-0.3'
merged_final.loc[((merged_final['Fuzzy_set'] == '< 1') &
(merged_final['Precip_grouped'] != '0-0.3')), 'FS_fix'] = '0-0.3'
merged_final.loc[((merged_final['Fuzzy_set'] == '< 1') &
(merged_final['Precip_grouped'] != '0.3-5')), 'FS_fix'] = '0.3-5'
merged_final.loc[((merged_final['Fuzzy_set'] == '2.5-7.5') &
(merged_final['Precip_grouped'] != '0.3-5')), 'FS_fix'] = '0.3-5'
merged_final.loc[((merged_final['Fuzzy_set'] == '2.5-7.5') &
(merged_final['Precip_grouped'] != '5-15')), 'FS_fix'] = '5-15'
merged_final.loc[((merged_final['Fuzzy_set'] == '10-20') &
(merged_final['Precip_grouped'] != '5-15')), 'FS_fix'] = '5-15'
merged_final.loc[((merged_final['Fuzzy_set'] == '10-20') &
(merged_final['Precip_grouped'] != '15-30')), 'FS_fix'] = '15-30'
merged_final.loc[((merged_final['Fuzzy_set'] == '20-40') &
(merged_final['Precip_grouped'] != '15-30')), 'FS_fix'] = '15-30'
merged_final.loc[((merged_final['Fuzzy_set'] == '20-40') &
(merged_final['Precip_grouped'] != '30-50')), 'FS_fix'] = '30-50'
merged_final.loc[((merged_final['Fuzzy_set'] == '40-70') &
(merged_final['Precip_grouped'] != '30-50')), 'FS_fix'] = '30-50'
merged_final.loc[((merged_final['Fuzzy_set'] == '40-70') &
(merged_final['Precip_grouped'] != '50-100')), 'FS_fix'] = '50-100'
merged_final.loc[((merged_final['Fuzzy_set'] == '80-150') &
(merged_final['Precip_grouped'] != '50-100')), 'FS_fix'] = '50-100'
merged_final.loc[((merged_final['Fuzzy_set'] == '80-150') &
(merged_final['Precip_grouped'] != '50-100')), 'FS_fix'] = '100-600'
merged_final.loc[((merged_final['Fuzzy_set'] == '500-700') &
(merged_final['Precip_grouped'] != '50-100')), 'FS_fix'] = '100-600'

states_prob_precip_df = pd.DataFrame()
capital_aqi_df = pd.DataFrame()
```

```
for state in states:
    state_data = merged_final.loc[(merged_final['State name'] == state) &
    (merged_final['Year'] != 2020)]
    capital = capitals_T.loc[capitals_T['index'] == state, ['Capital']]
    capital_value = capital.iloc[0, 0]
    capital_data = state_data.loc[state_data['county Name'] == capital_value]
    capital_aqi = aqi_df.loc[aqi_df['county Name'] == capital_value, ['Date', 'AQI']]
    capital_precipitation = capital_data.groupby('Date', as_index=False)
    ['Precipitation(mm)'].sum()
    location = state_data.drop_duplicates(subset=['county Name'])
    counties = location['county Name']
    distance_df = pd.DataFrame()
    location = state_data.drop_duplicates(subset=['county Name'])
    distance = location[['LocationLat', 'LocationLng', 'county Name',
    'State name']].sort_values(by=['LocationLng'])
    distance["lat_radians"] = np.radians(distance["LocationLat"])
    distance["lng_radians"] = np.radians(distance["LocationLng"])
    lat = distance.loc[distance['county Name'] == capital_value, ['LocationLat']]
    lng = distance.loc[distance['county Name'] == capital_value, ['LocationLng']]
    for row in distance.index:
        distance.loc[row, 'Capital_lat_rad'] = lat['LocationLat'].values[0]
        distance.loc[row, 'Capital_lng_rad'] = lng['LocationLng'].values[0]
    a = np.sin((distance["lat_radians"] - distance["Capital_lat_rad"]) / 2.0) ** 2
    + np.cos(distance["Capital_lat_rad"]) * np.cos(distance["lat_radians"]) *
    np.sin((distance["lng_radians"] - distance["Capital_lng_rad"]) / 2.0) ** 2
    distance['Distance'] = 6371 * 2 * np.arcsin(np.sqrt(a))
    distance["Inverse Distance"] = 1 / distance["Distance"]
    distance.loc[distance['county Name'] == capital_value, 'Inverse Distance'] = 0
    distance['NISD'] = distance['Inverse Distance'] / sum(distance['Inverse Distance'])
    distance_df = pd.concat([distance_df, distance['county Name'], distance['NISD'],
     distance['State name']], axis=1)

    corr_category_df = pd.DataFrame()
    corr_precip_df = pd.DataFrame()
    corr_category_county = pd.DataFrame()
    for county in counties:
        county_data = state_data.loc[(state_data['County'] == county) &
        (state_data['Year'] != 2020)]
        county_aqi = aqi_df.loc[aqi_df['county Name'] == county, ['Date', 'AQI']]
        county_aqi['AQI_county'] = county_aqi['AQI']
        merged_categories = capital_aqi.merge(county_aqi,
        left_on=['Date'], right_on=['Date'])
        corr_category, _ = stats.pearsonr(merged_categories['AQI_x'],
        merged_categories['AQI_county'])
        corr_category_county = pd.DataFrame([corr_category])
        corr_category_county['County'] = county
        corr_category_df = pd.concat([corr_category_df, corr_category_county])
        county_precipitation = county_data.groupby('Date', as_index=False)
        ['Precipitation(mm)'].sum()
        county_precipitation['Precipitation_county'] =
        county_precipitation['Precipitation(mm)']
        merged_precip = capital_precipitation.merge(county_precipitation,
        left_on=['Date'], right_on=['Date'])
        corr_precip, _ = stats.pearsonr(merged_precip['Precipitation(mm)_x'],
        merged_precip['Precipitation_county'])
        corr_precip_county = pd.DataFrame([corr_precip])
        corr_precip_county['County'] = county
        corr_precip_df = pd.concat([corr_precip_df, corr_precip_county])

    corr_category_df.loc[corr_category_df['County'] == capital_value, 0] = 0
    corr_category_df['NCorr_ctgr'] = corr_category_df[0] / np.nansum(corr_category_df[0])
    corr_precip_df.loc[corr_precip_df['County'] == capital_value, 0] = 0
    corr_precip_df['NCorr_precip'] = corr_precip_df[0] / np.nansum(corr_precip_df[0])
    merged_SW_1 = distance_df.merge(corr_category_df, left_on=['county Name'],
    right_on=['County'])
    merged_SW_2 = merged_SW_1.merge(corr_precip_df, left_on=['county Name'],
    right_on=['County'])
    merged_SW_2['SW'] = (merged_SW_2['NCorr_ctgr'] + merged_SW_2['NCorr_precip'] +
    merged_SW_2['NISD']) / np.nansum(merged_SW_2['NCorr_ctgr'] +
    merged_SW_2['NCorr_precip'] + merged_SW_2['NISD'])
    merged_SW_2.loc[merged_SW_2['county Name'] == capital_value, 'SW'] = 1

    prob_category_df = pd.DataFrame()
    prob_ctgr_precip_df = pd.DataFrame()
    for county in counties:
        county_data = state_data.loc[(state_data['County'] == county) &
        (state_data['Year'] != 2020)]
        sw = merged_SW_2['SW'].loc[merged_SW_2['county Name'] == county].values[0]

        prob_category = pd.crosstab(county_data['Category'], county_data['Year'],
        normalize='columns')
        prob_category = prob_category * sw
        prob_category_df = pd.concat([prob_category_df, prob_category])

        county_data_good = county_data.loc[county_data['Category'] == 'Good']
        county_data_moderate = county_data.loc[county_data['Category'] ==
        'Moderate']
        county_data_unhealthy = county_data.loc[county_data['Category'] ==
        'Unhealthy']

        prob_good = pd.crosstab(county_data_good['Precip_grouped'],
        county_data_good['Year'], normalize='columns')
        prob_moderate = pd.crosstab(county_data_moderate['Precip_grouped'],
    county_data_moderate['Year'], normalize='columns')
    prob_unhealthy = pd.crosstab(county_data_unhealthy['Precip_grouped'],
    county_data_unhealthy['Year'], normalize='columns')

    keys_category = {'Good': prob_good, 'Moderate': prob_moderate, 'Unhealthy':
    prob_unhealthy}
    prob_category_precip = pd.concat([prob_good, prob_moderate, prob_unhealthy],
    keys=keys_category.keys())
    prob_category_precip = prob_category_precip * sw

    year_list = pd.Series(prob_good.columns)
    if len(year_list) == 4:

        prob_good_fs = pd.crosstab(county_data_good['FS_fix'],
        county_data_good['Year']) / len(county_data_good)
        prob_moderate_fs = pd.crosstab(county_data_moderate['FS_fix'],
        county_data_moderate['Year']) / len(county_data_moderate)
        prob_unhealthy_fs = pd.crosstab(county_data_unhealthy['FS_fix'],
        county_data_unhealthy['Year']) / len(county_data_unhealthy)

        keys_category = {'Good': prob_good_fs, 'Moderate': prob_moderate_fs,
        'Unhealthy': prob_unhealthy_fs}
        prob_category_type_fs = pd.concat([prob_good_fs, prob_moderate_fs,
        prob_unhealthy_fs],keys=keys_category.keys())

        prob_category_type_idx = prob_category_precip.reset_index()
        prob_category_type_fs_idx = prob_category_type_fs.reset_index()
        prob_category_type_new =
        prob_category_type_idx.merge(prob_category_type_fs_idx,
        how='left', left_on=['level_0', 'Precip_grouped'], right_on=['level_0',
        'FS_fix'])
        prob_category_type_new[2016] = prob_category_type_new['2016_x'] +
        prob_category_type_new['2016_y']
        prob_category_type_new[2017] = prob_category_type_new['2017_x'] +
        prob_category_type_new['2017_y']
        prob_category_type_new[2018] = prob_category_type_new['2018_x'] +
        prob_category_type_new['2018_y']
        prob_category_type_new[2019] = prob_category_type_new['2019_x'] +
        prob_category_type_new['2019_y']
        category_type_df = pd.DataFrame(prob_category_type_new,
        columns=['level_0', 'Precip_grouped', 2016, 2017, 2018, 2019])
    else:
        l_idx = prob_category_precip.reset_index()
        category_type_df = l_idx

    prob_ctgr_precip_df = pd.concat([prob_ctgr_precip_df, category_type_df])
```

```
prob_ctgr_precip_index = prob_ctgr_precip_df.set_index(['level_0',
'Precip_grouped'], inplace=False)

prob_category_df_sum = prob_category_df.groupby(['Category']).sum()
normalization_constant_1 = 1 / prob_category_df_sum.sum()
prob_category_df_multiplied = prob_category_df_sum * normalization_constant_1

prob_ctgr_precip_df_sum = prob_ctgr_precip_index.groupby(['level_0',
'Precip_grouped']).sum()
prob_ctgr_precip_df_sum_indx = prob_ctgr_precip_df_sum.reset_index()

category_list = pd.Series(prob_category_df_sum.index)
normalized_df = pd.DataFrame()
for category in category_list:
    for_category =
    prob_ctgr_precip_df_sum_indx.loc[prob_ctgr_precip_df_sum_indx['level_0']
    == category]
    for_category = for_category.set_index(['level_0', 'Precip_grouped'],
    inplace=False)
    normalization_constant_2 = 1 / for_category.sum()
    prob_ctgr_precip_df_multiplied = for_category * normalization_constant_2
    normalized_df = pd.concat([normalized_df, prob_ctgr_precip_df_multiplied])

year_list = pd.Series(prob_category_df_sum.columns)
test_year = pd.Series(['2019_pred'])
year_list_test = pd.concat([year_list, test_year])

if len(year_list) == 4:
    TW1 = (1 / 1) / (1 / 3 + 1 / 2 + 1 / 1)
    TW2 = (1 / 2) / (1 / 3 + 1 / 2 + 1 / 1)
    TW3 = (1 / 3) / (1 / 3 + 1 / 2 + 1 / 1)
    TW = (1 / 3)
    prob_category_df_multiplied['2019_pred'] = TW3 *
    prob_category_df_multiplied[2016] + TW2 * prob_category_df_multiplied[2017]
    + TW1 * prob_category_df_multiplied[2018]
    normalized_df['2019_pred'] = TW3 * normalized_df[2016] + TW2 * \
                                 normalized_df[2017] + TW1 * \
                                 normalized_df[2018]

    prob_df_1 = pd.DataFrame()
    for year in year_list_test:
        for category in category_list:
            prob = prob_category_df_multiplied.loc[category, year] *
            normalized_df.loc[category, year]
            prob_category_multiplied_type = pd.DataFrame(prob)
            normalization_constant = 1 / prob_category_multiplied_type.sum()
            prob_df_normalized = prob_category_multiplied_type *
```

```
                normalization_constant
                prob_df_normalized['Category'] = category
                prob_df_1 = pd.concat([prob_df_1, prob_df_normalized])

        prob_df_index = prob_df_1.reset_index()
        prob_df_index = prob_df_index.set_index(['Category', 'Precip_grouped'],
        inplace=False)
        prob_df = prob_df_index.groupby(['Category', 'Precip_grouped']).sum()
        prob_df['State'] = state
        prob_df['County'] = capital_value

    states_prob_precip_df = pd.concat([states_prob_precip_df, prob_df])

SpaBN_set = states_prob_precip_df.reset_index()
max_2019_pred = SpaBN_set.groupby(['State', 'County', 'Category'], as_index=False,
sort=False)['2019_pred'].max()
SpaBN_set_2019_pred = SpaBN_set.merge(max_2019_pred, left_on=['State', 'County',
'Category', '2019_pred'], right_on=['State', 'County', 'Category', '2019_pred'])
SpaBN_set_2019_pred = SpaBN_set_2019_pred.drop([2016, 2017, 2018, 2019], axis=1)
SpaBN_set_2019_pred_f = SpaBN_set_2019_pred.drop_duplicates(subset=['State',
'County', 'Category'])

merged_probability = merged_final.merge(SpaBN_set_2019_pred_f, how='left',
left_on=['State name', 'County', 'Category'], right_on=['State', 'County', 'Category'])

value = []
for row in merged_probability['Precip_grouped_y']:
    if row == '0':
        value.append(0.0)
    elif row == '0-0.3':
        value.append(0.15)
    elif row == '0.3-5':
        value.append(2.65)
    elif row == '5-15':
        value.append(10)
    elif row == '15-30':
        value.append(22.5)
    elif row == '30-50':
        value.append(40)
    elif row == '50-100':
        value.append(75)
    elif row == '100-600':
        value.append(350)
    else:
        value.append(1000)

merged_probability['Precip_interval_middle'] = value
```

```
merged_final_df = merged_probability.loc[(merged_probability['Year'] == 2019)]
merged_final_df = merged_final_df.dropna(subset=['2019_pred'])
r2_book_SpaFBN = rsquared_book(merged_final_df['Precipitation(mm)'],
merged_final_df['Precip_interval_middle'])
MAE_SpaFBN = mean_absolute_error(merged_final_df['Precipitation(mm)'],
merged_final_df['Precip_interval_middle'])
rmse_SpaFBN = math.sqrt(mean_squared_error(merged_final_df['Precipitation(mm)'],
merged_final_df['Precip_interval_middle']))
```

Wild Bootstrapping

```
#Wild bootstrapping was applied as new for cycle:

rand_df = pd.DataFrame()
for i in range(500):
    x = np.random.normal(0, 0.5, len(merged_final))
    y = merged_final['Precipitation(mm)'] + x
    y.loc[y < 0] = 0
    merged_final['Precip_bootstrap'] = y

    intervals = []
    for row in merged_final['Precip_bootstrap']:
        if row == 0.0:
            intervals.append('0')
        elif 0.0 < row <= 0.3:
            intervals.append('0-0.3')
        elif 0.3 < row <= 5.0:
            intervals.append('0.3-5')
        elif 5.0 < row <= 15.0:
            intervals.append('5-15')
        elif 15.0 < row <= 30.0:
            intervals.append('15-30')
        elif 30.0 < row <= 50.0:
            intervals.append('30-50')
        elif 50.0 < row <= 100.0:
            intervals.append('50-100')
        elif 100.0 < row <= 600.0:
            intervals.append('100-600')
        else:
            intervals.append('600-28045')

    merged_final['Precip_grouped2'] = intervals

     # Insert any model from above.
     # Precipitation(mm) column has to be changed into Precip_bootstrap
     # Precip_grouped column has to be changed into Precip_grouped2
    rand_df = pd.concat([rand_df, states_prob_precip_df])

rand_df2 = rand_df.reset_index()
rand_df_gr = rand_df2.groupby(['State', 'County', 'Category', 'Precip_grouped2'],
as_index=False, sort=False)['2019_pred'].mean()
max_2019_pred = rand_df_gr.groupby(['State', 'County', 'Category'],
as_index=False, sort=False)['2019_pred'].max()
set_2019_pred = rand_df_gr.merge(max_2019_pred,
left_on=['State', 'County', 'Category', '2019_pred'],
right_on=['State', 'County', 'Category', '2019_pred'])
set_2019_pred_f = set_2019_pred.drop_duplicates(subset=['State', 'County', 'Category'])

merged_probability = merged_final.merge(set_2019_pred_f, how='left',
left_on=['State name', 'County', 'Category'],
right_on=['State', 'County', 'Category'])

value = []
for row in merged_probability['Precip_grouped2_y']:
    if row == '0':
        value.append(0.0)
    elif row == '0-0.3':
        value.append(0.15)
    elif row == '0.3-5':
        value.append(2.65)
    elif row == '5-15':
        value.append(10)
    elif row == '15-30':
        value.append(22.5)
    elif row == '30-50':
        value.append(40)
    elif row == '50-100':
        value.append(75)
    elif row == '100-600':
        value.append(350)
    else:
        value.append(1000)

merged_probability['Precip_interval_middle'] = value
merged_final_df = merged_probability.loc[(merged_probability['Year'] == 2019)]
merged_final_df = merged_final_df.dropna(subset=['2019_pred'])
r2_book = rsquared_book(merged_final_df['Precipitation(mm)'],
merged_final_df['Precip_interval_middle'])
MAE = mean_absolute_error(merged_final_df['Precipitation(mm)'],
merged_final_df['Precip_interval_middle'])
rmse = math.sqrt(mean_squared_error(merged_final_df['Precipitation(mm)'],
merged_final_df['Precip_interval_middle']))
```

Appendix 2. Python Code of Case Study 2.

SpaBN

```
states = unique_states['State name']
states_prob_df = pd.DataFrame()
for state in states:
    state_data = merged_data.loc[merged_data['State name'] == state]
    location = state_data.drop_duplicates(subset=['county Name'])
    counties = location['county Name']
    distance_df = pd.DataFrame()
    distance = location[['LocationLat', 'LocationLng', 'county Name',
    'State name']].sort_values(by=['LocationLng'])
    distance["lat_radians"] = np.radians(distance["LocationLat"])
    distance["lng_radians"] = np.radians(distance["LocationLng"])
    capital = capitals_T.loc[capitals_T['index'] == state, ['Capital']]
    capital_value = capital.iloc[0, 0]
    lat = distance.loc[distance['county Name'] == capital_value, ['LocationLat']]
    lng = distance.loc[distance['county Name'] == capital_value, ['LocationLng']]
    for row in distance.index:
        distance.loc[row, 'Capital_lat_rad'] = lat['LocationLat'].values[0]
        distance.loc[row, 'Capital_lng_rad'] = lng['LocationLng'].values[0]
    a = np.sin((distance["lat_radians"] - distance["Capital_lat_rad"]) / 2.0) ** 2
    + np.cos(distance["Capital_lat_rad"]) * np.cos(distance["lat_radians"]) *
    np.sin((distance["lng_radians"] - distance["Capital_lng_rad"]) / 2.0) ** 2
    distance['Distance'] = 6371 * 2 * np.arcsin(np.sqrt(a))
    distance["Inverse Distance"] = 1 / distance["Distance"]
    distance.loc[distance['county Name'] == capital_value, 'Inverse Distance'] = 0
    distance['Spatial Weight'] = distance['Inverse Distance'] /
    sum(distance['Inverse Distance'])
    distance.loc[distance['county Name'] == capital_value, 'Spatial Weight'] = 1
    distance_df = pd.concat([distance_df, distance['county Name'],
    distance['Spatial Weight'], distance['State name']],axis=1)

    county_prob_df = pd.DataFrame()
    for county in counties:
        county_data = state_data.loc[
            (state_data['County'] == county) & (state_data['Year'] != 2020)]
        prob_category = pd.crosstab(county_data['Category'],
        county_data['Year'], normalize='columns')

        years = county_data.drop_duplicates(subset=['Year'])
        year_list = years['Year']
        test_year = pd.Series('2019_pred')
        year_list_test = year_list.append(test_year)
        categories = county_data.drop_duplicates(subset=['Category'])
        category_list = categories['Category']
```

```
if len(year_list) == 4:
    TW1 = (1 / 1) / (1 / 3 + 1 / 2 + 1 / 1)
    TW2 = (1 / 2) / (1 / 3 + 1 / 2 + 1 / 1)
    TW3 = (1 / 3) / (1 / 3 + 1 / 2 + 1 / 1)
    prob_category['2019_pred'] = TW3 * prob_category[2016] +
    TW2 * prob_category[2017] + TW1 * prob_category[2018]

    county_data_good = county_data.loc[county_data['Category'] == 'Good']
    county_data_moderate = county_data.loc[county_data['Category'] ==
    'Moderate']
    county_data_unhealthy = county_data.loc[county_data['Category'] ==
    'Unhealthy']

    prob_good = pd.crosstab(county_data_good['Type'],
    county_data_good['Year'], normalize='columns')
    prob_moderate = pd.crosstab(county_data_moderate['Type'],
     county_data_moderate['Year'], normalize='columns')
    prob_unhealthy = pd.crosstab(county_data_unhealthy['Type'],
    county_data_unhealthy['Year'], normalize='columns')

    keys_category = {'Good': prob_good, 'Moderate': prob_moderate,
    'Unhealthy': prob_unhealthy}
    prob_category_type = pd.concat([prob_good, prob_moderate, prob_unhealthy],
    keys=keys_category.keys())
    prob_category_type['2019_pred'] = TW3 * prob_category_type[2016] +
    TW2 * prob_category_type[2017] + TW1 * prob_category_type[2018]

    prob_df_1 = pd.DataFrame()
    for year in year_list_test:
        for category in category_list:
            prob = prob_category_type.loc[category, year] *
            prob_category.loc[category, year]
            prob_category_multiplied_type = pd.DataFrame(prob)
            prob_category_multiplied_type['Category'] = category
            prob_df_1 = pd.concat([prob_df_1, prob_category_multiplied_type])

    prob_df_index = prob_df_1.reset_index()
    prob_df_index = prob_df_index.set_index(['Category', 'Type'], inplace=False)
    prob_df = prob_df_index.groupby(['Category', 'Type']).sum()
    prob_df['County'] = county

    spatial_index = distance['Spatial Weight'].loc[distance['county Name']
    == county].values[0]
    index_spatial_df = prob_df.index
    columns_spatial_df = prob_df.columns.drop('County')
    spatial_probability_df = pd.DataFrame(index=index_spatial_df,
                                                   columns=columns_spatial_df)
            for row in range(len(prob_df)):
                for column in range(len(prob_df.columns) - 1):
                    spatial_probability = prob_df.iloc[row, column] * spatial_index
                    spatial_probability_df.iloc[row, column] = spatial_probability

            county_prob_df = pd.concat([county_prob_df, spatial_probability_df])
    county_prob_df_sum = county_prob_df.groupby(['Category', 'Type']).sum()
    normalization_constant = 1/county_prob_df_sum.sum()
    county_prob_df_multiplied = county_prob_df_sum * normalization_constant
    county_prob_df_multiplied["State"] = state
    county_prob_df_multiplied["Capital"] = capital_value
    states_prob_df = pd.concat([states_prob_df, county_prob_df_multiplied])

SpaBN_set2 = states_prob_df.reset_index()
max_2019_pred = SpaBN_set2.groupby(['State', 'Capital', 'Category'],
as_index=False, sort=False)['2019_pred'].max()
SpaBN_set_2019_pred = SpaBN_set2.merge(max_2019_pred,
left_on=['State', 'Capital', 'Category', '2019_pred'],
right_on=['State', 'Capital', 'Category', '2019_pred'])
SpaBN_set_2019_pred = SpaBN_set_2019_pred.drop([2016, 2017, 2018, 2019], axis=1)
SpaBN_set_2019_pred_f = SpaBN_set_2019_pred.drop_duplicates(subset=['State',
'Capital', 'Category'])

merged_probability = merged_data.merge(SpaBN_set_2019_pred_f, how='left',
left_on=['State name', 'County', 'Category'],
right_on=['State', 'Capital', 'Category'])

merged_final_df = merged_probability.loc[(merged_probability['Year'] == 2019)]
merged_final_df = merged_final_df.dropna(subset=['2019_pred'])
accuracy1 = accuracy_score(merged_final_df['Type_x'], merged_final_df['Type_y'])
```

SpaBN with DBSCAN

```
cols = ['#e6194b', '#3cb44b', '#ffe119', '#4363d8', '#f58231', '#911eb4',
        '#46f0f0', '#f032e6', '#bcf60c', '#fabebe', '#008080', '#e6beff',
        '#9a6324', '#fffac8', '#800000', '#aaffc3', '#808000', '#ffd8b1',
        '#000075', '#808080'] * 10000

sns.set(style="white")

df = merged_data.drop_duplicates(subset=['county Name', 'LocationLat', 'LocationLng'])

X = np.array(df[['LocationLng', 'LocationLat']], dtype='float64')
plt.scatter(X[:, 0], X[:, 1], alpha=0.2, s=50)

m = folium.Map(location=[df.LocationLat.mean(), df.LocationLng.mean()], zoom_start=9)
# tiles='OpenStreet Map')
```

```
for _, row in df.iterrows():
    folium.CircleMarker(
        location=[row.LocationLat, row.LocationLng],
        radius=5,
        # popup=re.sub(r'[^a-zA-Z ]+', '', row.MMSI),
        color='#1787FE',
        fill=True,
        fill_colour='#1787FE'
    ).add_to(m)
m.save("mymap.html")


def create_map(df, cluster_column):
    m = folium.Map(location=[df.LocationLat.mean(), df.LocationLng.mean()],
    zoom_start=9)  # tiles='OpenStreet Map')
    for _, row in df.iterrows():
        if row[cluster_column] == -1:
            cluster_colour = '#000000'
        else:
            cluster_colour = cols[row[cluster_column]]
        folium.CircleMarker(
            location=[row['LocationLat'], row['LocationLng']],
            radius=5,
            popup=row[cluster_column],
            color=cluster_colour,
            fill=True,
            fill_color=cluster_colour
        ).add_to(m)

    return m


model = DBSCAN(eps=0.5, min_samples=1).fit(X)
class_predictions = model.labels_
df['CLUSTERS_DBSCAN'] = class_predictions
m = create_map(df, 'CLUSTERS_DBSCAN')
print(f'Number of clusters found: {len(np.unique(class_predictions))}')
print(f'Number of outliers found: {len(class_predictions[class_predictions == -1])}')
print(f'Silhouette ignoring outliers: {silhouette_score(X[class_predictions != -1],
class_predictions[class_predictions != -1])}')
no_outliers = 0
no_outliers = np.array([(counter + 2) * x if x == -1 else x for counter,
x in enumerate(class_predictions)])
print(f'Silhouette outliers as singletons: {silhouette_score(X, no_outliers)}')
m.save("mymap_clusters.html")

max_class = max(class_predictions)
```

```
for class1 in range(0, max_class):
    class_data = df.loc[df['CLUSTERS_DBSCAN'] == class1]
    state_class_data = class_data.drop_duplicates(subset=['State Name'])
    state_class = state_class_data['State Name'].values[0]
    capital = capitals_T.loc[capitals_T['index'] == state_class, ['Capital']].values
    county_class_data = class_data['county Name'].values
    if capital in county_class_data:
        distance_df = pd.DataFrame()
        distance = class_data[['LocationLat', 'LocationLng', 'county Name',
        'State name']].sort_values(by=['LocationLng'])
        counties = distance['county Name']
        distance["lat_radians"] = np.radians(distance["LocationLat"])
        distance["lng_radians"] = np.radians(distance["LocationLng"])
        capital = capitals_T.loc[capitals_T['index'] == state_class, ['Capital']]
        capital_value = capital.iloc[0, 0]
        lat = distance.loc[distance['county Name'] == capital_value, ['LocationLat']]
        lng = distance.loc[distance['county Name'] == capital_value, ['LocationLng']]
        for row in distance.index:
            distance.loc[row, 'Capital_lat_rad'] = lat['LocationLat'].values[0]
            distance.loc[row, 'Capital_lng_rad'] = lng['LocationLng'].values[0]
        a = np.sin((distance["lat_radians"] - distance["Capital_lat_rad"]) / 2.0) ** 2
        + np.cos(distance["Capital_lat_rad"]) * np.cos(distance["lat_radians"]) *
        np.sin((distance["lng_radians"] - distance["Capital_lng_rad"]) / 2.0) ** 2
        distance['Distance'] = 6371 * 2 * np.arcsin(np.sqrt(a))
        distance["Inverse Distance"] = 1 / distance["Distance"]
        distance.loc[distance['county Name'] == capital_value, 'Inverse Distance'] = 0
        distance['Spatial Weight'] = distance['Inverse Distance'] /
        sum(distance['Inverse Distance'])
        distance.loc[distance['county Name'] == capital_value, 'Spatial Weight'] = 1
        distance_df = pd.concat([distance_df, distance['county Name'],
        distance['Spatial Weight'], distance['State name']], axis=1)
        county_prob_df = pd.DataFrame()
        for county in counties:
            county_data = merged_data.loc[(merged_data['County'] == county) &
            (merged_data['Year'] != 2020)]
            prob_category = pd.crosstab(county_data['Category'], county_data['Year'],
            normalize='columns')

            years = county_data.drop_duplicates(subset=['Year'])
            year_list = years['Year']
            test_year = pd.Series('2019_pred')
            year_list_test = year_list.append(test_year)
            categories = county_data.drop_duplicates(subset=['Category'])
            category_list = categories['Category']

            if len(year_list) == 4:
                TW1 = (1 / 1) / (1 / 3 + 1 / 2 + 1 / 1)
```

```
TW2 = (1 / 2) / (1 / 3 + 1 / 2 + 1 / 1)
TW3 = (1 / 3) / (1 / 3 + 1 / 2 + 1 / 1)
prob_category['2019_pred'] = TW3 * prob_category[2016] +
TW2 * prob_category[2017] + TW1 * prob_category[2018]

county_data_good = county_data.loc[county_data['Category'] == 'Good']
county_data_moderate = county_data.loc[county_data['Category'] ==
'Moderate']
county_data_unhealthy = county_data.loc[county_data['Category'] ==
'Unhealthy']

prob_good = pd.crosstab(county_data_good['Type'],
county_data_good['Year'], normalize='columns')
prob_moderate = pd.crosstab(county_data_moderate['Type'],
county_data_moderate['Year'], normalize='columns')
prob_unhealthy = pd.crosstab(county_data_unhealthy['Type'],
county_data_unhealthy['Year'], normalize='columns')

keys_category = {'Good': prob_good, 'Moderate': prob_moderate,
'Unhealthy': prob_unhealthy}
prob_category_type = pd.concat([prob_good, prob_moderate,
prob_unhealthy], keys=keys_category.keys())
prob_category_type['2019_pred'] = TW3 * prob_category_type[2016] +
TW2 * prob_category_type[2017] + TW1 * prob_category_type[2018]

prob_df_1 = pd.DataFrame()
for year in year_list_test:
    for category in category_list:
        prob = prob_category_type.loc[category, year] *
        prob_category.loc[category, year]
        prob_category_multiplied_type = pd.DataFrame(prob)
        prob_category_multiplied_type['Category'] = category
        prob_df_1 = pd.concat([prob_df_1,
        prob_category_multiplied_type])

prob_df_index = prob_df_1.reset_index()
prob_df_index = prob_df_index.set_index(['Category', 'Type'],
inplace=False)
prob_df = prob_df_index.groupby(['Category', 'Type']).sum()
prob_df['County'] = county

spatial_index = distance['Spatial Weight'].loc[distance['county Name']
== county].values[0]
index_spatial_df = prob_df.index
columns_spatial_df = prob_df.columns.drop('County')
spatial_probability_df = pd.DataFrame(index=index_spatial_df,
                                      columns=columns_spatial_df)
```

```
                for row in range(len(prob_df)):
                    for column in range(len(prob_df.columns) - 1):
                        spatial_probability = prob_df.iloc[row, column] * spatial_index
                        spatial_probability_df.iloc[row, column] = spatial_probability

                county_prob_df = pd.concat([county_prob_df, spatial_probability_df])
        county_prob_df_sum = county_prob_df.groupby(['Category', 'Type']).sum()
        normalization_constant = 1/county_prob_df_sum.sum()
        county_prob_df_multiplied = county_prob_df_sum * normalization_constant
        county_prob_df_multiplied["State"] = state_class
        county_prob_df_multiplied["Capital"] = capital_value
        states_prob_df = pd.concat([states_prob_df, county_prob_df_multiplied])

SpaBN_set2 = states_prob_df.reset_index()
max_2019_pred = SpaBN_set2.groupby(['State', 'Capital', 'Category'],
as_index=False, sort=False)['2019_pred'].max()
SpaBN_set_2019_pred = SpaBN_set2.merge(max_2019_pred, left_on=['State', 'Capital',
'Category', '2019_pred'], right_on=['State', 'Capital', 'Category', '2019_pred'])
SpaBN_set_2019_pred = SpaBN_set_2019_pred.drop([2016, 2017, 2018, 2019], axis=1)
SpaBN_set_2019_pred_f = SpaBN_set_2019_pred.drop_duplicates(subset=['State', 'Capital',
'Category'])

merged_probability = merged_data.merge(SpaBN_set_2019_pred_f, how='left',
left_on=['State name', 'County', 'Category'],
right_on=['State', 'Capital', 'Category'])
merged_final_df = merged_probability.loc[(merged_probability['Year'] == 2019)]
merged_final_df = merged_final_df.dropna(subset=['2019_pred'])
accuracy2 = accuracy_score(merged_final_df['Type_x'], merged_final_df['Type_y'])
```