Import libraries

In [ ]:
```
# Data Manipulation and Analysis
import pandas as pd
import numpy as np
from collections import defaultdict

# Data Visualization
import matplotlib.pyplot as plt
import seaborn as sns
from matplotlib.widgets import Button
import networkx as nx

# Data Preprocessing
from pandas import read_csv
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Handling Imbalanced Data
from imblearn.over_sampling import SMOTE
from sklearn.utils import resample

# Feature Selection
from sklearn import feature_selection, metrics
from sklearn.feature_selection import SelectKBest

# Model Building
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier, BaggingClassifier, AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression

# Model Evaluation and Optimization
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import confusion_matrix, classification_report


# Miscellaneous
import os
from collections import Counter
from scipy.spatial import Delaunay
import glob
from math import cos, radians
```

Folders/files path

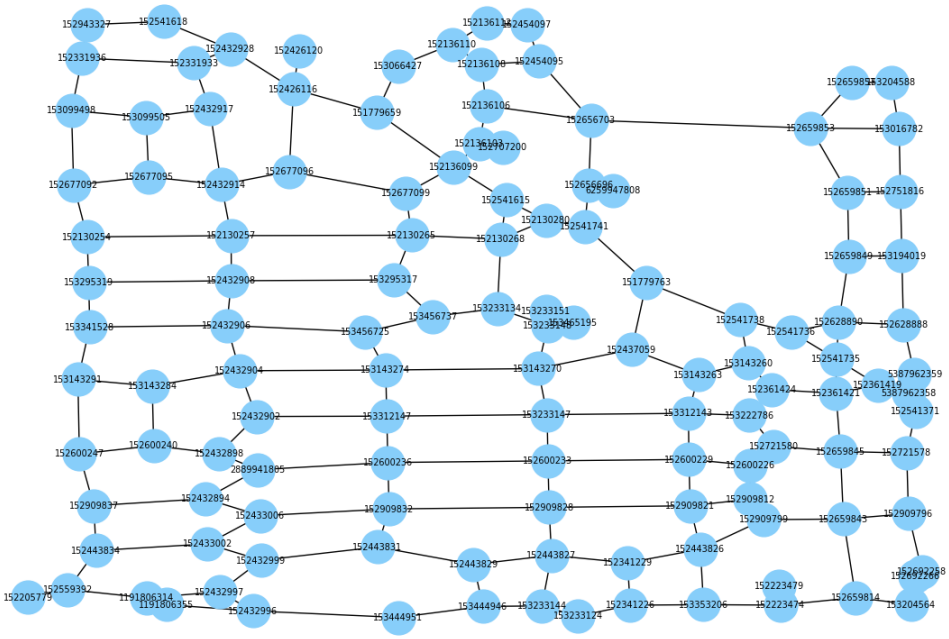
In [ ]:
```
edge_info_path = r"C:\Users\lucap\OneDrive\Desktop\ANNO 2\Machine Learning\ASSIGNEMENT\MLAssignment202324\MLAssignment202324\edge_info.csv"
training_parameters_path = r"C:\Users\lucap\OneDrive\Desktop\ANNO 2\Machine Learning\ASSIGNEMENT\MLAssignment202324\MLAssignment202324\training_parameters.csv"
training_folder_path =r"C:\Users\lucap\OneDrive\Desktop\ANNO 2\Machine Learning\ASSIGNEMENT\MLAssignment202324\MLAssignment202324\training"
test_folder_path =r"C:\Users\lucap\OneDrive\Desktop\ANNO 2\Machine Learning\ASSIGNEMENT\MLAssignment202324\MLAssignment202324\test"
```

TRAINING DATASET MERGER

Before combining all the observation datasets, a new variable was established. This variable records the path length from the nearest initially flooded edge for each edge in the observation dataset. To facilitate this, a graph was constructed using the NetworkX library, with nodes representing the head and tail of each edge, as depicted in the following image.

In [ ]:
```
edge_data = pd.read_csv(edge_info_path)

graph = nx.from_pandas_edgelist(edge_data, 'head_id', 'tail_id')


node_positions = {}
for node in graph.nodes():

    edges = edge_data[(edge_data['head_id'] == node) | (edge_data['tail_id'] == node)]
    avg_longitude = edges['longitude'].mean()
    avg_latitude = edges['latitude'].mean()
    node_positions[node] = (avg_longitude, avg_latitude)

plt.figure(figsize=(12, 8))
nx.draw(graph, pos=node_positions, with_labels=True, node_size=700, node_color="lightskyblue", font_size=7)
plt.title("Graph Representation with Geographical Positions")
plt.show()
```

Graph Representation with Geographical Positions


152943327
152541618
152331936
152331933
152432928
152426120
152426116
153099498
153099505
152432917
152677092
152677095
152432914
152677096
152130254
152130257
153295319
152432908
153341528
152432906
153143291
153143284
152432904
152600247
152600240
152432898
152909837
152432894
152433006
152443834
152433002
152432999
152205779
152559392
1191806314
1191806355
152432997
152432996
153444951
153066427
151779659
152677099
152130265
153295317
153456737
153456725
153143274
152600236
152909832
152443831
153444946
152136110
152136108
152454095
152136106
152707200
152136103
152136099
152541615
152130280
152130268
153233134
153233151
153235148
153465195
153143270
152600233
152909820
152443829
153233144
153233124
152443827
152656703
152656698
6259947808
152541741
151779763
152437059
153312147
153233147
152600229
152909821
152443826
152341229
152341226
153353206
152659853
153204588
153016782
152659851
152751816
152659849
153194019
152541738
152628890
152628888
152541736
152541735
153143260
153143263
152361424
152361421
152361419
5387962359
5387962358
153312143
153222786
152541371
152721580
152659845
152721578
152600226
152909812
152909799
152659843
152909796
152223479
152223474
152659814
153204564
152692268
152692266
```
```

The process involved calculating the distance of each node from the closest node associated with an initially flooded edge. To determine this value for each edge, the smaller of the two distances - one from the head and the other from the tail node - was selected.

In the end, when **flooded_init** is False, we increment the value of **shortest_flooded_distance** by 1. This adjustment is essential to accurately calculate the distance of each edge from the nearest flooded edge. For instance, prior calculations assigned a value of zero to the shortest_flooded_distance for edges that were directly flooded as well as for those that were adjacent to flooded edges.

In [ ]:
```
# Define the path to the training folder
folder_path = training_folder_path

# Get all CSV files from the folder
csv_files = [file for file in os.listdir(folder_path) if file.endswith('.csv')]
# Initialize an empty list to hold dataframes
dfs = []
for file in csv_files:
    # Construct the full file path
    file_path = os.path.join(folder_path, file)
    # Read the CSV file into a DataFrame
    df = pd.read_csv(file_path)

    # Create a graph from the current DataFrame
    graph = nx.from_pandas_edgelist(df, 'head_id', 'tail_id', ['flooded_init'])
    flooded_nodes = set()
    # Identify nodes involved in flooding
    for u, v, attrs in graph.edges(data=True):
        if attrs['flooded_init']:
            flooded_nodes.update([u, v])

    # Initialize a dictionary to store distances
    distances = {}
    # Calculate distances to all 'flooded' nodes
    for node in graph.nodes():
        distances_to_flooded = [nx.shortest_path_length(graph, node, flooded_node) for flooded_node in flooded_nodes if nx.has_path(graph, node, flooded_node)]
        # Assign the minimum distance to the 'distances' dictionary
        if distances_to_flooded:
            distances[node] = min(distances_to_flooded)
        else:
            distances[node] = float('inf')  # Assign infinite distance if no 'flooded' nodes are reachable

    # Calculate and assign the shortest distance to a flooded node for each edge in the DataFrame
    df['shortest_flooded_distance'] = df.apply(lambda row: min(distances[row['head_id']], distances[row['tail_id']]), axis=1)

    # Add the name of the source file as a column
    file_name_without_extension = os.path.splitext(file)[0]
    df['source_file'] = int(file_name_without_extension)

    # Append the modified DataFrame to the list
    dfs.append(df)

# Concatenate all DataFrames into one
training = pd.concat(dfs, ignore_index=True)
# Define the order of columns
training_columns=['source_file', 'head_id','tail_id','shortest_flooded_distance','flooded_init', 'flooded_final']
# Increment distance by 1 where there's no initial flooding
training.loc[training['flooded_init'] == False, 'shortest_flooded_distance'] += 1
# Reorder the DataFrame columns
training=training[training_columns]
# Display the first 10 rows of the resulting DataFrame
training.head(10)
```

Out[ ]:

	source_file	head_id	tail_id	shortest_flooded_distance	flooded_init	flooded_final
0	0	151779659	153066427	7	False	False
1	0	151779659	152426116	7	False	False
2	0	151779659	152136099	6	False	False
3	0	151779763	152437059	3	False	False
4	0	151779763	152541741	4	False	False
5	0	151779763	152541738	3	False	False
6	0	152130254	152130257	6	False	False
7	0	152130254	152677092	7	False	False
8	0	152130254	153295319	6	False	False
9	0	152130257	152130265	6	False	False

In [ ]:
```
# reading the othrer 2 csv files
training_parameters=pd.read_csv(training_parameters_path)
edge_info=pd.read_csv(edge_info_path)
```

Connected Edges

We create another variable storing the number of adjacent edges. In this context, 'adjacent edges' are defined as those that share a common node with a given edge, whether it be the head or the tail node. This variable essentially counts the number of edges connected to either the head or tail node of each edge, providing insight into the connectivity and network structure within the dataset.

In [ ]:
```
# Creating a graph-like structure to count the connections
connections = defaultdict(int)

# Counting connections for each head_id and tail_id
for index, row in edge_info.iterrows():
    connections[row['head_id']] += 1
    connections[row['tail_id']] += 1

# Calculating the number of connected edges for each edge
# An edge is considered connected if its head_id or tail_id matches with any other edge's head_id or tail_id
edge_info['connected_edges'] = edge_info.apply(lambda row: connections[row['head_id']] + connections[row['tail_id']] - 2, axis=1)

edge_info.insert(0, 'index', edge_info.index)


edge_info.head()
```

Out[ ]:

	index	head_id	tail_id	longitude	latitude	altitude	connected_edges
0	0	151779659	153066427	-95.382821	29.798740	957.0	3
1	1	151779659	152426116	-95.383237	29.798445	921.2	5
2	2	151779659	152136099	-95.382354	29.797764	876.0	5
3	3	151779763	152437059	-95.380082	29.795224	897.0	4
4	4	151779763	152541741	-95.380394	29.796295	877.6	4

DATASET CREATION

All the initial datasets are merged in order to obtain the definitive one

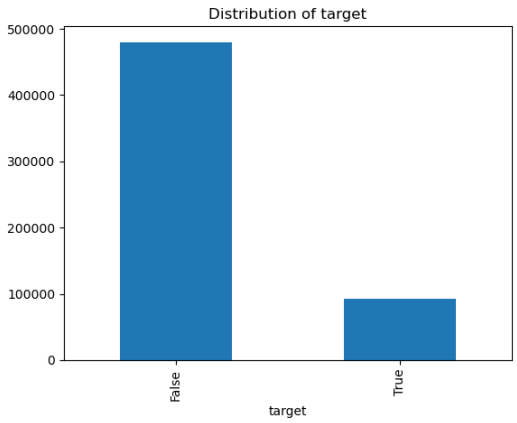
In [ ]:
```
training2=training.merge(training_parameters, left_on='source_file',right_on='ObservationIndex', how='left')
training2=training2.drop(['ObservationIndex'], axis=1)
training_final=training2.merge(edge_info, on=['head_id','tail_id'], how='left')
columns=list(training_final.columns[:4]) + list(training_final.columns[6:])
columns2=list(training_final.columns[4:6])
training_final=training_final[columns+columns2]
training_final = training_final.rename(columns={'flooded_final': 'target'})
training_final.head()
```

Out[ ]:

	source_file	head_id	tail_id	shortest_flooded_distance	SurfaceType	RainfallIntensity	init_max_hour	DrainageSystemCapacity	GreenSpaceRatio	index	longitude	latitude	altitude	connected_edges	flooded_init	target
0	0	151779659	153066427	7	D	5	3	0.11	0.11	0	-95.382821	29.798740	957.0	3	False	False
1	0	151779659	152426116	7	D	5	3	0.11	0.11	1	-95.383237	29.798445	921.2	5	False	False
2	0	151779659	152136099	6	D	5	3	0.11	0.11	2	-95.382354	29.797764	876.0	5	False	False
3	0	151779763	152437059	3	D	5	3	0.11	0.11	3	-95.380082	29.795224	897.0	4	False	False
4	0	151779763	152541741	4	D	5	3	0.11	0.11	4	-95.380394	29.796295	877.6	4	False	False

In [ ]:
```
target_dist=training_final.groupby('target').size()
target_dist.plot.bar(x='',y='',title='Distribution of target')
```

Out[ ]:
```
<Axes: title={'center': 'Distribution of target'}, xlabel='target'>
```



MISSING VALUES AND OUTLIERS DETECTION

In [ ]:
```
training_final.isna().sum()  # this shows the absence of missing values in our dataset
```

Out[ ]:
```
source_file                0
head_id                    0
tail_id                    0
shortest_flooded_distance  0
SurfaceType                0
RainfallIntensity          0
init_max_hour              0
DrainageSystemCapacity     0
GreenSpaceRatio            0
index                      0
longitude                  0
latitude                   0
altitude                   0
connected_edges            0
flooded_init               0
target                     0
dtype: int64
```

In [ ]:
```
training_final.describe(include="all")  # to analyze the description of each column of the df
```

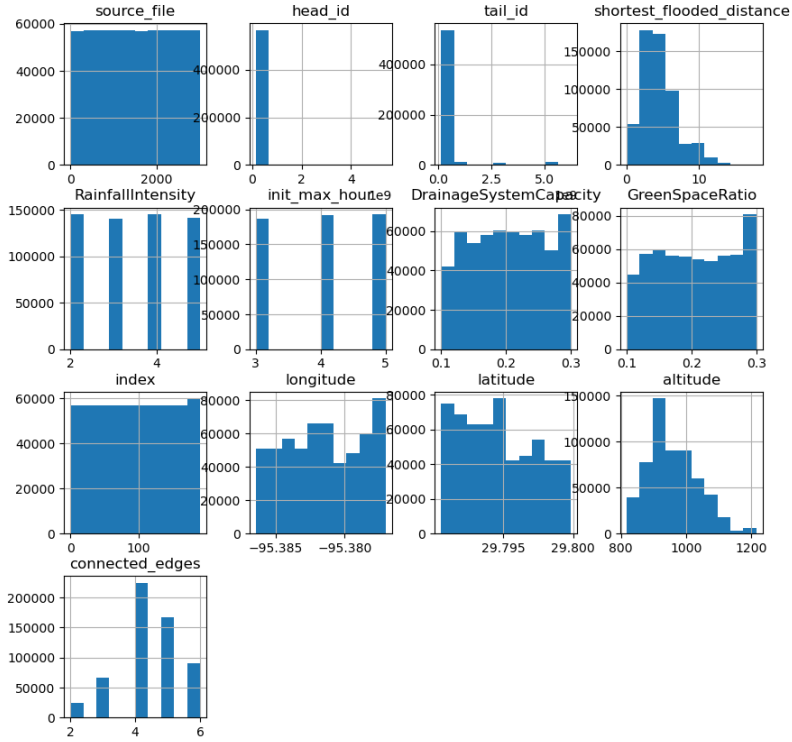
Out[ ]:

	source_file	head_id	tail_id	shortest_flooded_distance	SurfaceType	RainfallIntensity	init_max_hour	DrainageSystemCapacity	GreenSpaceRatio	index	longitude	latitude	altitude	connected_edg
count	573000.000000	5.730000e+05	5.730000e+05	573000.000000	573000	573000.000000	573000.000000	573000.000000	573000.000000	573000.000000	573000.000000	573000.000000	573000.000000	573000.0000
unique	NaN	NaN	NaN	NaN	4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na
top	NaN	NaN	NaN	NaN	C	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na
freq	NaN	NaN	NaN	NaN	150508	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na
mean	1499.500000	1.853797e+08	3.591893e+08	4.450736	NaN	2.157667	4.010667	0.199047	0.200780	95.000000	-95.381516	29.794657	958.667016	4.4083
std	866.026111	3.848152e+08	9.310898e+08	2.544439	NaN	36.598613	0.814793	0.057167	0.059035	55.136243	0.002809	0.002673	76.164736	1.0185
min	0.000000	1.517797e+08	1.521303e+08	0.000000	NaN	-999.000000	3.000000	0.100000	0.100000	0.000000	-95.386469	29.790537	817.500000	2.0000
25%	749.750000	1.523614e+08	1.525417e+08	3.000000	NaN	2.000000	3.000000	0.150000	0.150000	47.000000	-95.384290	29.792273	908.400000	4.0000
50%	1499.500000	1.524438e+08	1.529098e+08	4.000000	NaN	4.000000	4.000000	0.200000	0.200000	95.000000	-95.381482	29.794362	942.100000	4.0000
75%	2249.250000	1.526598e+08	1.532331e+08	6.000000	NaN	4.000000	5.000000	0.250000	0.250000	143.000000	-95.378971	29.796821	1009.400000	5.0000
max	2999.000000	5.387962e+09	6.259948e+09	18.000000	NaN	5.000000	5.000000	0.300000	0.300000	190.000000	-95.377019	29.799847	1215.300000	6.0000

We decided to eliminate the Observations 79, 141, 1737 and 1738 since they presented a RainfallIntensity value of -999, which likely indicates missing or erroneous data. Since the number of observations removed is considerably smaller relatively to the magnitude of the dataset, this won't affect the final result.

In [ ]:
```
df = training_final[training_final['RainfallIntensity'] >= 0]
df.hist(figsize=(10,10))
```

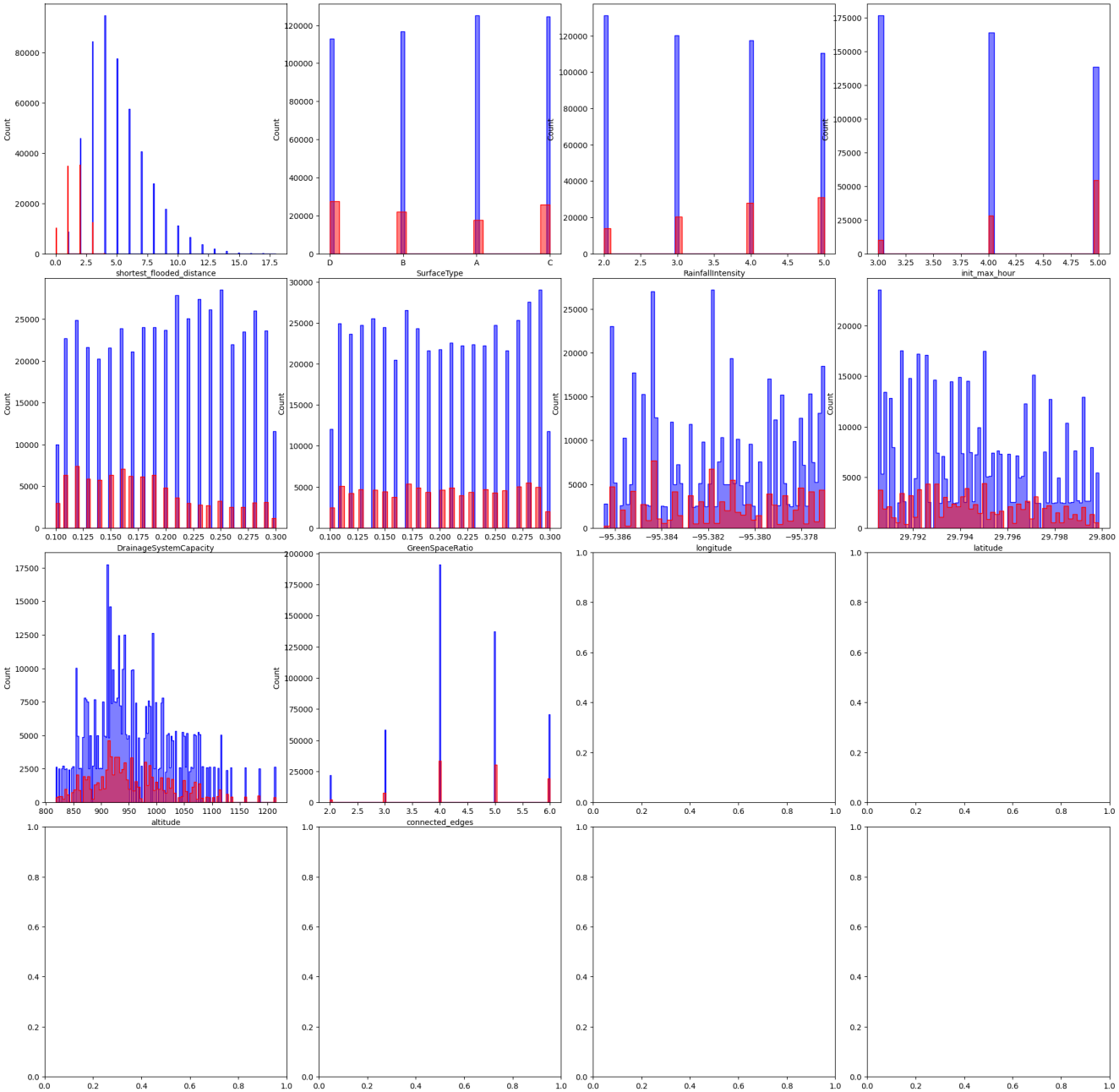
Out[ ]:
```
array([[<Axes: title={'center': 'source_file'}>,
        <Axes: title={'center': 'head_id'}>,
        <Axes: title={'center': 'tail_id'}>,
        <Axes: title={'center': 'shortest_flooded_distance'}>],
       [<Axes: title={'center': 'RainfallIntensity'}>,
        <Axes: title={'center': 'init_max_hour'}>,
        <Axes: title={'center': 'DrainageSystemCapacity'}>,
        <Axes: title={'center': 'GreenSpaceRatio'}>],
       [<Axes: title={'center': 'index'}>,
        <Axes: title={'center': 'longitude'}>,
        <Axes: title={'center': 'latitude'}>,
        <Axes: title={'center': 'altitude'}>],
       [<Axes: title={'center': 'connected_edges'}>, <Axes: >, <Axes: >,
        <Axes: >]], dtype=object)
```



EDA

In [ ]:

```
%matplotlib inline


# dataset created just for these plots
X=df.drop(['source_file','head_id','tail_id','flooded_init','target','index'], axis=1)

X0 = X[df['target']==0]
X1 = X[df['target']==1]

fig, axes = plt.subplots(ncols=4, nrows=4, figsize=(20,20))
fig.tight_layout()

for i, ax in zip(range(X.columns.size), axes.flat):
    sns.histplot(X0.iloc[:,i], color="blue", element="step", ax=ax,  alpha=0.5, discrete=False)
    sns.histplot(X1.iloc[:,i], color="red", element="step", ax=ax,  alpha=0.5,discrete=False)
plt.show()
```



From the plots above it is possible to notice how the number of target = True for the SurfaceType variable, increases going from type A to type D. For this reason, we will map these values from A, B, C, D to 1, 2, 3, 4.

In [ ]:
```
mapping = {'A': 1, 'B': 2, 'C': 3, 'D': 4}
df['SurfaceType'] = df['SurfaceType'].map(mapping)
df.describe(include='all')
```

```
C:\Users\lucap\AppData\Local\Temp\ipykernel_24096\992043725.py:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  df['SurfaceType'] = df['SurfaceType'].map(mapping)
```

Out[ ]:

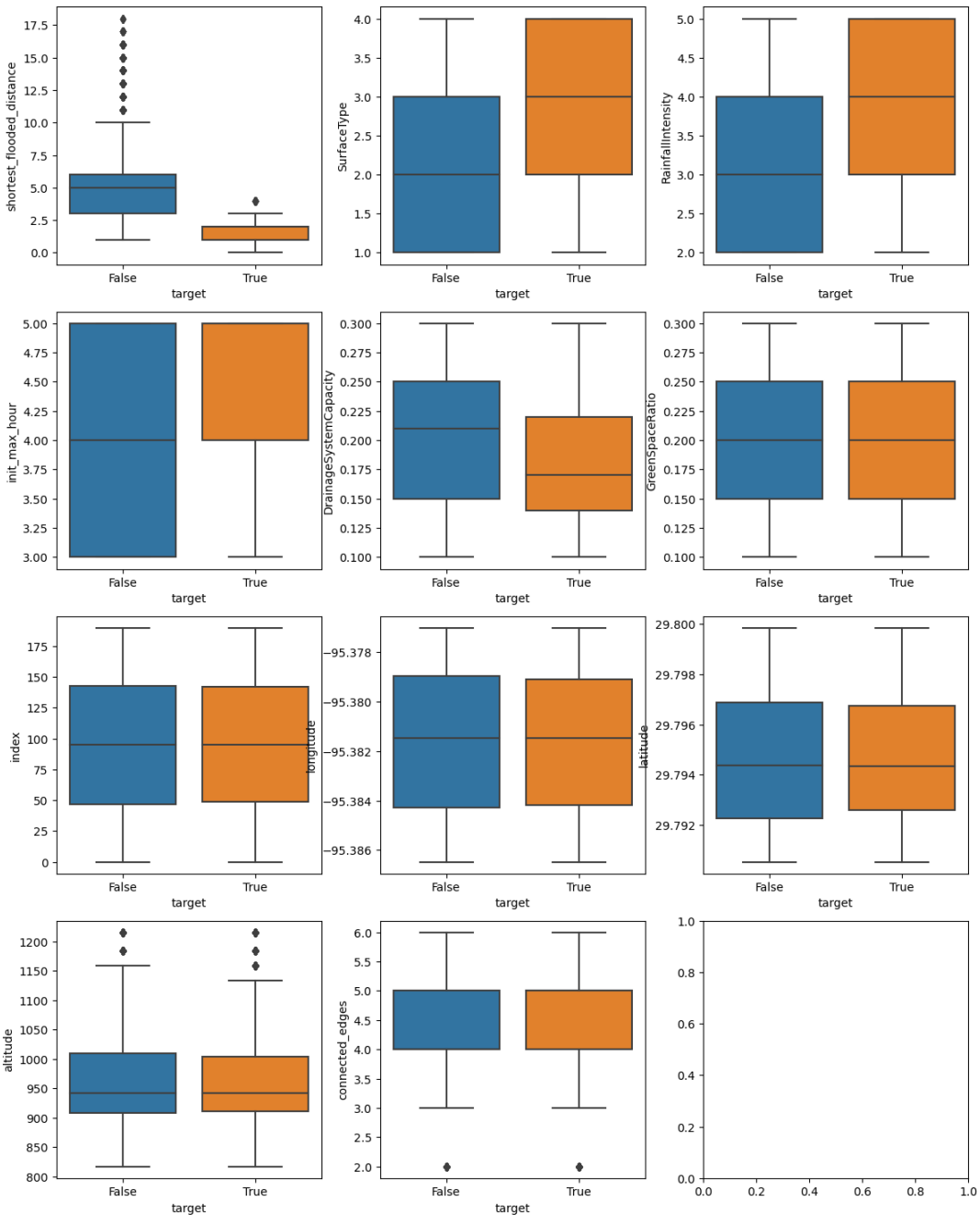
	source_file	head_id	tail_id	shortest_flooded_distance	SurfaceType	RainfallIntensity	init_max_hour	DrainageSystemCapacity	GreenSpaceRatio	index	longitude	latitude	altitude	connected_ed
count	572236.000000	5.722360e+05	5.722360e+05	572236.000000	572236.000000	572236.000000	572236.000000	572236.000000	572236.000000	572236.000000	572236.000000	572236.000000	572236.000000	572236.00
unique	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
top	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
freq	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
mean	1500.268692	1.853797e+08	3.591893e+08	4.451597	2.504005	3.494326	4.011015	0.199089	0.200768	95.000000	-95.381516	29.794657	958.667016	4.40
std	865.837547	3.848152e+08	9.310898e+08	2.545019	1.113241	1.118617	0.814718	0.057180	0.059019	55.136243	0.002809	0.002673	76.164736	1.01
min	0.000000	1.517797e+08	1.521303e+08	0.000000	1.000000	2.000000	3.000000	0.100000	0.100000	0.000000	-95.386469	29.790537	817.500000	2.00
25%	750.750000	1.523614e+08	1.525417e+08	3.000000	2.000000	2.000000	3.000000	0.150000	0.150000	47.000000	-95.384290	29.792273	908.400000	4.00
50%	1499.500000	1.524438e+08	1.529098e+08	4.000000	3.000000	4.000000	4.000000	0.200000	0.200000	95.000000	-95.381482	29.794362	942.100000	4.00
75%	2250.250000	1.526598e+08	1.532331e+08	6.000000	3.000000	4.000000	5.000000	0.250000	0.250000	143.000000	-95.378971	29.796821	1009.400000	5.00
max	2999.000000	5.387962e+09	6.259948e+09	18.000000	4.000000	5.000000	5.000000	0.300000	0.300000	190.000000	-95.377019	29.799847	1215.300000	6.00

In [ ]:
```
# Univariate test
X=df.drop(['source_file','head_id','tail_id','flooded_init','target'], axis=1)
X['longitude']=abs(X['longitude'])
y=df['target']

cs = SelectKBest(score_func=feature_selection.chi2, k=5)
cs.fit(X,y)
feature_score = pd.DataFrame({'Score':cs.scores_, 'p_val':cs.pvalues_}, index=X.columns)
feature_score.nlargest(10, columns='Score')
```

Out[ ]:

	Score	p_val
shortest_flooded_distance	211519.914394	0.000000e+00
init_max_hour	6080.946897	0.000000e+00
RainfallIntensity	3224.367641	0.000000e+00
SurfaceType	1359.410130	1.389271e-297
connected_edges	1041.034034	2.163332e-228
altitude	203.468269	3.656029e-46
DrainageSystemCapacity	182.525104	1.361784e-41
index	45.015877	1.954434e-11
GreenSpaceRatio	0.018634	8.914198e-01
latitude	0.000005	9.982178e-01

In [ ]:
```
%matplotlib inline

fig, axes = plt.subplots(ncols=3, nrows=4, figsize=(12,15))
fig.tight_layout(pad=2)

for i, col in enumerate(X.columns):
    sns.boxplot(y = col, x = "target",data=df, orient='v', ax=axes[int(i/3),i%3])
```



In [ ]:
```
corr = X.corr()
corr.style.background_gradient(cmap='coolwarm', vmin=-1, vmax=1)
```

Out[ ]:

	shortest_flooded_distance	SurfaceType	RainfallIntensity	init_max_hour	DrainageSystemCapacity	GreenSpaceRatio	index	longitude	latitude	altitude	connected_edges
shortest_flooded_distance	1.000000	-0.192000	-0.288076	-0.002148	0.014700	-0.004820	-0.003057	0.015077	0.035947	0.068946	-0.206268
SurfaceType	-0.192000	1.000000	-0.002260	0.001607	0.003597	-0.001012	-0.000000	-0.000000	0.000000	-0.000000	0.000000
RainfallIntensity	-0.288076	-0.002260	1.000000	0.005379	-0.046602	-0.000591	-0.000000	0.000000	0.000000	-0.000000	-0.000000
init_max_hour	-0.002148	0.001607	0.005379	1.000000	0.007667	0.002184	0.000000	0.000000	0.000000	-0.000000	-0.000000
DrainageSystemCapacity	0.014700	0.003597	-0.046602	0.007667	1.000000	-0.000475	-0.000000	0.000000	-0.000000	-0.000000	-0.000000
GreenSpaceRatio	-0.004820	-0.001012	-0.000591	0.002184	-0.000475	1.000000	0.000000	0.000000	-0.000000	0.000000	0.000000
index	-0.003057	-0.000000	-0.000000	0.000000	-0.000000	0.000000	1.000000	-0.244984	-0.255189	0.165398	-0.003449
longitude	0.015077	-0.000000	0.000000	0.000000	0.000000	0.000000	-0.244984	1.000000	0.138150	-0.088626	-0.038031
latitude	0.035947	0.000000	0.000000	0.000000	-0.000000	-0.000000	-0.255189	0.138150	1.000000	0.026118	-0.088107
altitude	0.068946	-0.000000	-0.000000	-0.000000	-0.000000	0.000000	0.165398	-0.088626	0.026118	1.000000	0.034580
connected_edges	-0.206268	0.000000	-0.000000	-0.000000	-0.000000	0.000000	-0.003449	-0.038031	-0.088107	0.034580	1.000000

Creation of variables linking the edge info with the training parameters

1. **Altitued and GreenSpaceRatio**: The product between these two could be useful for examining how the combined effect of altitude and green spaces influences flood risk. For instance, high areas with high green spaces might have a lower risk.

1. **Altitude and RainfallIntensity**: The relationship between rainfall intensity and altitude could be examined through their ratio. Edges situated at lower altitudes might be more susceptible to flooding, particularly when faced with high rainfall intensity.

1. **Altitude and connected_edges**: We used the ratio similarly as before. Edges with low altitude and a high number of adjacent edges, might have a higher likelihood of flooding, since water tends to flow downhill.

In the formulation of these variables we only included *altitude* from the edge_info dataset, since as seen from the EDA, longitude and latitude seems to be irrelevant for the analysis.

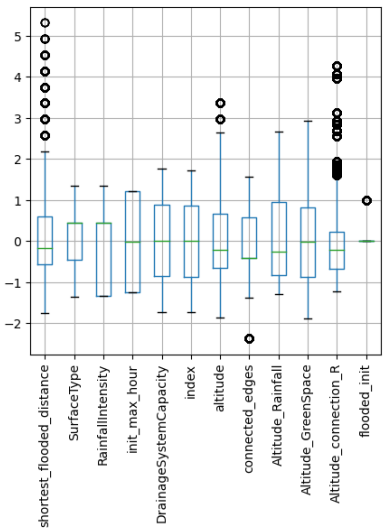
In [ ]:
```
df.loc[:, 'Altitude_Rainfall'] = df['altitude'] / df['RainfallIntensity']
df.loc[:, 'Altitude_GreenSpace'] = df['altitude'] * df['GreenSpaceRatio']
df.loc[:, 'Altitude_connection_R'] = df['altitude'] / df['connected_edges']


df.head()
```

DATA PREPARATION

In [ ]:
```
# Load the master dataset with feature engineering

# Selecting features and target variable
X = df.drop(['target','source_file','head_id','tail_id', 'GreenSpaceRatio','latitude', 'longitude'], axis=1)
y = df['target']
# Splitting the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=88, stratify= y)
```

In [ ]:
```
# Assuming df is your DataFrame with both numerical and dummy columns
colonne_numeriche = X_train.select_dtypes(include=['float64', 'int']).columns  # Select numerical columns
colonne_dummy = X_train.select_dtypes(include=['bool']).columns  # Select dummy/boolean columns

# Standardizing only numerical variables
scaler = StandardScaler().fit(X_train[colonne_numeriche])  # Fit scaler on numerical columns of training data
df_scaled = pd.DataFrame(scaler.transform(X_train[colonne_numeriche]), columns=colonne_numeriche, index=X_train.index)  # Transform the data and create a DataFrame

# Rejoin the scaled numerical variables with the dummy variables
X_train_scaled  = pd.concat([df_scaled, X_train[colonne_dummy]], axis=1)  # Concatenate the scaled numerical and dummy columns

# Standardize the test set as well
X_test_sc_num = pd.DataFrame(scaler.transform(X_test[colonne_numeriche]), columns=colonne_numeriche, index=X_test.index)  # Transform the test set numerical data
X_test_scaled = pd.concat([X_test_sc_num, X_test[colonne_dummy]], axis=1)  # Concatenate the scaled numerical and dummy columns for test data
```

In [ ]:
```
# final boxplot of scaled variables
X_train_scaled.boxplot(rot=90, figsize=(5,5))
plt.show()
```



BALANCING

In [ ]:
```
train_data = pd.concat([X_train_scaled, y_train], axis = 1)
print(train_data.groupby("target").size())
df_majority = train_data[train_data.target == 0]
df_minority = train_data[train_data.target == 1]

df_majority_downsampled = resample(df_majority,
                                   replace = False,
                                   n_samples = 300000,
                                   random_state = 88)
df_downsampled = pd.concat([df_majority_downsampled, df_minority])
print(df_downsampled.groupby("target").size())
y_train = df_downsampled["target"]
X_train = df_downsampled.drop(columns=["target"])


sm = SMOTE(random_state=27, sampling_strategy=0.7)  # This approach will not result in a perfectly balanced dataset, but it is an improvement over the previous state.
X_SMOTE, y_SMOTE = sm.fit_resample(X_train, y_train)

print('Resampled dataset shape %s' % Counter(y_SMOTE))
```

```
target
False    359497
True      69680
dtype: int64
target
False    300000
True      69680
dtype: int64
Resampled dataset shape Counter({False: 300000, True: 210000})
```

CLASSIFICATION MODELS

Initially we fit all the models starting with the complete dataset

LOGISTIC REGRESSION

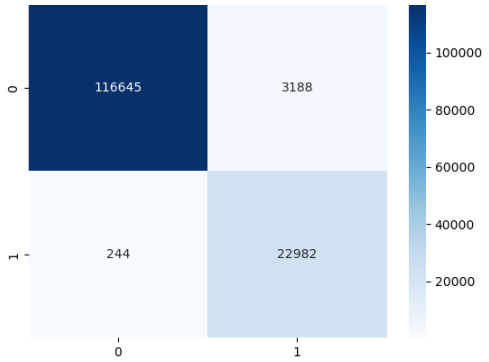
In [ ]:
```
classifier = LogisticRegression()
parameters = {"C":[1e-1,1,10],"penalty":['l2'] , "max_iter":[50000]}

gs = GridSearchCV(classifier, parameters, cv=3, scoring = 'f1', verbose=50, n_jobs=-1, refit=True)

gs = gs.fit(X_SMOTE, y_SMOTE)
print('***GRIDSEARCH RESULTS***')

print("Best score: %f using %s" % (gs.best_score_, gs.best_params_))
```

```
Fitting 3 folds for each of 3 candidates, totalling 9 fits
***GRIDSEARCH RESULTS***
Best score: 0.977027 using {'C': 0.1, 'max_iter': 50000, 'penalty': 'l2'}
```

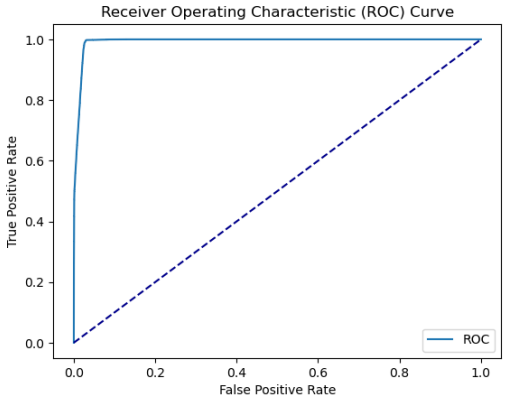
In [ ]:
```
best_model = gs.best_estimator_
y_pred_log = best_model.predict(X_test_scaled)
y_pred_train = best_model.predict(X_SMOTE)
print(gs.best_estimator_.coef_)


print('***RESULTS ON TRAIN SET***')
print("f1_score: ", metrics.f1_score(y_SMOTE, y_pred_train))
print("--")
print('***RESULTS ON TEST SET***')
print("f1_score: ", metrics.f1_score(y_test, y_pred_log))
```

```
[[-1.48919328e+01  1.20459442e-02 -1.05582282e-01  4.51287568e+00
  -2.46823386e+00  1.10361793e-02  1.55468198e-02  5.80258693e-02
  -1.37321604e-01 -3.10954226e-02  2.74987818e-02  2.00517448e+00]]
***RESULTS ON TRAIN SET***
f1_score:  0.9777402758733843
--
***RESULTS ON TEST SET***
f1_score:  0.9305206899344076
```

In [ ]:
```
confusion_matrix(y_test, y_pred_log)

sns.heatmap(confusion_matrix(y_test, y_pred_log), annot=True, fmt='d', cmap="Blues");
```



In [ ]:
```
# ROC curve
y_probs_log = best_model.predict_proba(X_test_scaled) #predict_proba gives the probabilities for the target (0 and 1 in your case)

fpr, tpr, thresholds=metrics.roc_curve(y_test,  y_probs_log[:,1])

plt.plot(fpr, tpr, label='ROC')
plt.plot([0, 1], [0, 1], color='darkblue', linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend()
plt.show()

auc = metrics.roc_auc_score(y_test, y_probs_log[:,1])
print('AUC: %.2f' % auc)
```




```
AUC: 0.99
```

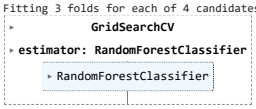
RANDOM FOREST

In [ ]:
```
classifier= RandomForestClassifier()
parameters = {'n_estimators' : [100],
    'criterion' : ['gini'],
              'min_samples_split' : [5,10],
               'min_samples_leaf' : [5,10]
              }

gs = GridSearchCV(classifier, parameters, cv=3, scoring = 'f1', verbose=10, n_jobs=-1, refit=True) #cv era 3 prima
```

```
gs.fit(X_SMOTE,y_SMOTE)
```

```
Fitting 3 folds for each of 4 candidates, totalling 12 fits
```

Out[ ]:



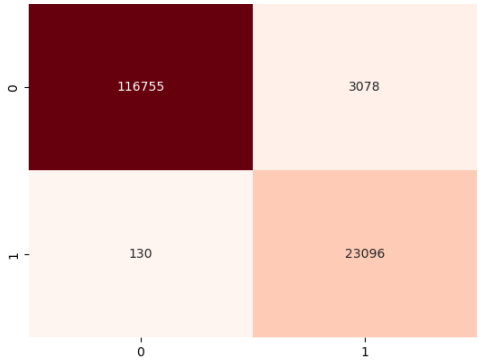
In [ ]:
```
best_parameters = gs.best_params_
best_score = gs.best_score_

print("Best Parameters:", best_parameters)
print("Best F1 Score:", best_score)
```

```
Best Parameters: {'criterion': 'gini', 'min_samples_leaf': 5, 'min_samples_split': 5, 'n_estimators': 100}
Best F1 Score: 0.9791909466861405
```

In [ ]:
```
best_model = gs.best_estimator_
y_pred_RF = best_model.predict(X_test_scaled)
# Plot confusion matrix

sns.heatmap(confusion_matrix(y_test, y_pred_RF), annot=True, fmt='d', cmap="Reds", cbar=False)
```

Out[ ]:

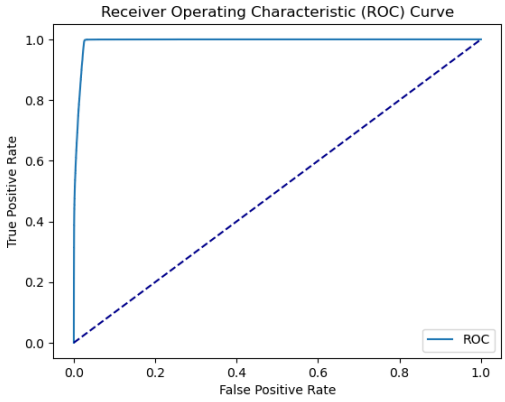
```
<Axes: >
```



In [ ]:
```
print(classification_report(y_test, y_pred_RF))
```

```
              precision    recall  f1-score   support

       False       1.00      0.97      0.99    119833
        True       0.88      0.99      0.94     23226

    accuracy                           0.98    143059
   macro avg       0.94      0.98      0.96    143059
weighted avg       0.98      0.98      0.98    143059
```

In [ ]:
```
# ROC curve
y_probs_RF = best_model.predict_proba(X_test_scaled) #predict_proba gives the probabilities for the target (0 and 1 in your case)

fpr, tpr, thresholds=metrics.roc_curve(y_test,  y_probs_RF[:,1])


plt.plot(fpr, tpr, label='ROC')
plt.plot([0, 1], [0, 1], color='darkblue', linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend()
plt.show()

auc = metrics.roc_auc_score(y_test, y_probs_RF[:,1])
print('AUC: %.2f' % auc)
```




```
AUC: 0.99
```

Conclusions

F-1 score

In order to make our choice of the model, the score taken as reference is the F-1 metric. To sum up the models we tried are showing this best f1 scores:

- KNN: 0.82
- Logistic Regression: 0.93
- Random Forest: 0.93

The results show a predominance of the Logistic Regression and the Random Forest methods, which are basically providing the same scores.

Moreover, we tried some others models such as Bagging and Adaboost using the GridSearch method with different hyperparameters, but the results were not satisfactory. For time reasons we have not included them in the discussion.

It is important to highlight that in predicting flooded streets, **recall** is more important than **precision** because:

- *Safety Priority*: It's crucial to identify as many actual flood events as possible to ensure public safety and effective emergency response, even at the cost of some false alarms.
- *Minimizing Missed Floods*: Missing a flood can have severe consequences, including risks to life and property. High recall minimizes the risk of not detecting actual floods.
- *Risk Management*: The cost of not predicting an actual flood (false negative) is typically much higher than mistakenly predicting a flood (false positive), making it essential to capture all potential floods.
- *Public Trust*: Consistently detecting floods builds trust in the prediction system, ensuring that the public takes necessary precautions seriously.

Model selection

In the process of selecting an appropriate model from those we have trained and tested, we were faced with the decision of choosing between Logistic Regression and Random Forest. After careful consideration, several pivotal factors guided our decision.

On the one hand, a critical variable to consider is the model's **speed**. Logistic regression is notably faster and more efficient; it requires significantly less time to train, even with smaller datasets like ours, thereby enhancing our ability to rapidly update and refine the model as needed.

On the other hand, while Random Forest is acknowledged for its robustness against outliers, this aspect is of less significance in our specific scenario, given that outliers do not pose a substantial challenge in our dataset.

Best model & Parzimony principle

All the models have been trained using all the variables, but looking at the relevant variables only few are really needed for the prediction.

In the following fit we will use all the variables:

In [ ]:
```
model = LogisticRegression(C=0.1, max_iter=50000, penalty='l2')

model.fit(X_SMOTE, y_SMOTE)
from sklearn import metrics

y_pred_log1 = model.predict(X_test_scaled)
print("f1_score: ", metrics.f1_score(y_test, y_pred_log1))
```

```
f1_score:  0.9305206899344076
```

In [ ]:
```
from sklearn.metrics import classification_report

print(classification_report(y_test, y_pred_log1))
```

```
              precision    recall  f1-score   support

       False       1.00      0.97      0.99    119833
        True       0.88      0.99      0.93     23226

    accuracy                           0.98    143059
   macro avg       0.94      0.98      0.96    143059
weighted avg       0.98      0.98      0.98    143059
```

While in this we are keeping only the most important ones:

- **shortest_flooded_distance**
- **init_max_hour**
- **DrainageSystemCapacity**
- **Altitude_Rainfall**
- **flooded_init**

In [ ]:
```
X2=X_SMOTE.drop(['connected_edges','SurfaceType', 'index', 'altitude','Altitude_GreenSpace','Altitude_connection_R','RainfallIntensity'],axis=1)
X2_test=X_test_scaled.drop(['connected_edges', 'SurfaceType', 'index', 'altitude','Altitude_GreenSpace','Altitude_connection_R','RainfallIntensity'],axis=1)

best_model = model.fit(X2, y_SMOTE)
from sklearn import metrics

y_pred_log2 = best_model.predict(X2_test)
print("f1_score: ", metrics.f1_score(y_test, y_pred_log2))
```

```
f1_score:  0.9306774519716887
```

The result demonstrates around 1% improvement in the F1 score. It's also important to note that while some variables were initially thought to have high importance, they were eventually found to be negligible, such as GreenSpaceRatio, SurfaceType...

In [ ]:
```
import pickle

# Save the best model from grid search
model_filename = r'C:\Users\lucap\OneDrive\Desktop\ANNO 2\Machine Learning\ASSIGNEMENT\MLAssignment202324\MLAssignment202324\best_model_logistic.pkl'

# Open a file in write-binary mode and save the model
with open(model_filename, 'wb') as file:
    pickle.dump(best_model, file)

print(f"Model saved as {model_filename}")
```

```
Model saved as C:\Users\lucap\OneDrive\Desktop\ANNO 2\Machine Learning\ASSIGNEMENT\MLAssignment202324\MLAssignment202324\best_model_logistic.pkl
```

Predictions on the final test set

In [ ]:
```
test_parameters=pd.read_csv(r"C:\Users\lucap\OneDrive\Desktop\ANNO 2\Machine Learning\ASSIGNEMENT\MLAssignment202324\MLAssignment202324\test_parameters.csv")
```

In [ ]:
```
from pandas import read_csv
folder_path = r"C:\Users\lucap\OneDrive\Desktop\ANNO 2\Machine Learning\ASSIGNEMENT\MLAssignment202324\MLAssignment202324\test"
csv_files = [file for file in os.listdir(folder_path) if file.endswith('.csv')]
# Lista vuota per contenere i dataframes
dfs = []
for file in csv_files:
    file_path = os.path.join(folder_path, file)
    df = pd.read_csv(file_path)

    # Creare un grafo per il DataFrame corrente
    graph = nx.from_pandas_edgelist(df, 'head_id', 'tail_id', ['flooded_init'])
    flooded_nodes = set()
    for u, v, attrs in graph.edges(data=True):
        if attrs['flooded_init']:
            flooded_nodes.update([u, v])

    distances = {}
    for node in graph.nodes():
    # Calculate distances to all 'flooded' nodes
        distances_to_flooded = [nx.shortest_path_length(graph, node, flooded_node) for flooded_node in flooded_nodes if nx.has_path(graph, node, flooded_node)]
        if distances_to_flooded:
            distances[node] = min(distances_to_flooded)
        else:
            distances[node] = float('inf')  # Infinite distance if no 'flooded' nodes are reachable

    df['shortest_flooded_distance'] = df.apply(lambda row: min(distances[row['head_id']], distances[row['tail_id']]), axis=1)

    # Aggiungere il nome del file come colonna
    file_name_without_extension = os.path.splitext(file)[0]
    df['source_file'] = int(file_name_without_extension)

    # Aggiungere il DataFrame modificato alla lista
    dfs.append(df)
# Concatena tutti i dataframes
test = pd.concat(dfs, ignore_index=True)
test_columns=['source_file', 'head_id','tail_id','shortest_flooded_distance','flooded_init']
test.loc[test['flooded_init'] == False, 'shortest_flooded_distance'] += 1
test=test[test_columns]
test.head(10)
```

Out[ ]:

	source_file	head_id	tail_id	shortest_flooded_distance	flooded_init
0	0	151779659	153066427	0	True
1	0	151779659	152426116	1	False
2	0	151779659	152136099	1	False
3	0	151779763	152437059	3	False
4	0	151779763	152541741	4	False
5	0	151779763	152541738	4	False
6	0	152130254	152130257	0	True
7	0	152130254	152677092	1	False
8	0	152130254	153295319	1	False
9	0	152130257	152130265	1	False

In [ ]:
```
test2=test.merge(test_parameters, left_on='source_file',right_on='ObservationIndex', how='left')
test2=test2.drop(['ObservationIndex'], axis=1)
test_final=test2.merge(edge_info, on=['head_id','tail_id'], how='left')
columns=list(test_final.columns[:4]) + list(test_final.columns[6:])
columns2=list(test_final.columns[4:6])
test_final=test_final[columns+columns2]
mapping = {'A': 1, 'B': 2, 'C': 3, 'D': 4}
test_final['SurfaceType'] = test_final['SurfaceType'].map(mapping)
test_final.describe(include='all')

test_final.loc[:, 'Altitude_Rainfall'] = test_final['altitude'] / test_final['RainfallIntensity']
test_final.loc[:, 'Altitude_GreenSpace'] = test_final['altitude'] * test_final['GreenSpaceRatio']
test_final.loc[:, 'Altitude_connection_R'] = test_final['altitude'] / test_final['connected_edges']
test_model=test_final.drop(['head_id','tail_id', 'GreenSpaceRatio','latitude', 'longitude'],axis=1)
#test_model=test_model[X_train_scaled.columns]
#test_model=test_final.drop(['connected_edges','SurfaceType', 'index', 'altitude','tail_id','head_id','GreenSpaceRatio','Altitude_GreenSpace','Altitude_GreenSpace','Altitude_connection_R','RainfallIntensity
```

In [ ]:
```
#Let's standardize also the test set
test_num_sc=pd.DataFrame(scaler.transform(test_model[df_scaled.columns]),columns=df_scaled.columns,index=test_model.index)
test_scal=pd.concat([test_num_sc,test_model[['source_file','flooded_init']]],axis=1)
test_scal=test_scal.drop(['connected_edges','SurfaceType', 'index', 'altitude','Altitude_GreenSpace','Altitude_connection_R','RainfallIntensity'],axis=1)
```

In [ ]:
```
X2=X_SMOTE.drop(['connected_edges','SurfaceType', 'index', 'altitude','Altitude_GreenSpace','Altitude_connection_R','RainfallIntensity'],axis=1)
best_model=best_model.fit(X2, y_SMOTE)

for i in range(200):
    subset = test_scal[test_scal['source_file'] == i] #subset used to find the predictions
    subset = subset.drop(columns=['source_file'])

    predictions = best_model.predict(subset)
    subset2= test_final[test_final['source_file'] == i]  #subset to build the prediction datasets
    subset2['flooded_final'] = predictions

    percorso_file = os.path.join(r"C:\Users\lucap\OneDrive\Desktop\ANNO 2\Machine Learning\ASSIGNEMENT\MLAssignment202324\MLAssignment202324\Test_predictions", f'pred_{i}.csv')

    subset2[['head_id','tail_id','flooded_init','flooded_final']].to_csv(percorso_file, index=False)
```

Appendix

It worths to mention an interesting variant of the variable **Closeness to the initial flooded edge** that has been tried.

In fact, this proposed variable is even more accurate in terms of distance between the edge and the clostset flooded one, because it is taking into account the distance in meters between the two edges.

The passages followed to create this variable are the following:

1. Creation from the edge_info file a graph taking into account the coordinates of every edge (so every node of the graph rapresents the coordinates of the edge)

1. Linking with arcs only the adjacent edges (nodes), where adjeacent meaning sharing at least a head or a tail id.
   The Manahattan distance was chosen because more rapresentative of the real map streets layout.

2. Calculating a distance matrix using the Manahttan distance for every node.

3. Assigning to every arch a weight equal to the Manahattan distance on meters separating the two nodes. So the higher the weight the higher the distance between the two nodes.

1. All the distances between a node and the flooded initial nodes are computed and the minimum one is taken.

However when fitting the models with this variable the results were worst than the other one.

In [ ]:
```
output_directory = r"C:\Users\lucap\OneDrive\Desktop\ANNO 2\Machine Learning\ASSIGNEMENT\MLAssignment202324\MLAssignment202324\training_with_distance_meters"
```

Crating the distance matrix

In [ ]:
```
def create_planar_graph(file_path_coordinates):
    # Load the distance matrix from an Excel file
    #distance_matrix = pd.read_excel(file_path_distance_matrix, index_col=0).to_numpy()

    # Load coordinates and IDs from the CSV file
    edge_data = pd.read_csv(file_path_coordinates)
    coordinates = edge_data[['latitude', 'longitude']].values

    # Create Delaunay triangulation
    tri = Delaunay(coordinates)

    # Create graph from triangulation and initialize distance matrix
    G = nx.Graph()
    removed_edges = []  # List to keep track of removed edges
    num_nodes = len(coordinates)
    manhattan_distance_matrix = np.zeros((num_nodes, num_nodes))  # Initialize the distance matrix

    # Approximate lengths of degrees
    meters_per_degree_lat = 111132.92  # varies from 110.57 km at the equator to 111.70 km at the poles
    meters_per_degree_lon = 111412.84  # at the equator and smaller towards the poles

    # Adding nodes to graph
    for index, row in edge_data.iterrows():
        G.add_node(index, head_id=row['head_id'], tail_id=row['tail_id'])

    for simplex in tri.simplices:
        for i in range(3):
            for j in range(i + 1, 3):
                node1, node2 = simplex[i], simplex[j]

                lat1, lon1 = coordinates[node1]
                lat2, lon2 = coordinates[node2]

                # Calculate the differences in latitude and longitude
                lat_diff = np.abs(lat1 - lat2)
                lon_diff = np.abs(lon1 - lon2)

                # Convert degree differences to meters
                lat_diff_meters = lat_diff * meters_per_degree_lat
                lon_diff_meters = lon_diff * meters_per_degree_lon * np.cos(np.radians((lat1 + lat2) / 2))  # Adjusting for latitude

                # Calculate Manhattan distance in meters
                manhattan_distance = lat_diff_meters + lon_diff_meters

                # Add edge to graph with weight
                G.add_edge(node1, node2, weight=manhattan_distance)

                # Store in the distance matrix
                manhattan_distance_matrix[node1][node2] = manhattan_distance
                manhattan_distance_matrix[node2][node1] = manhattan_distance  # Ensure symmetry

    # Remove edges that don't share a head_id or tail_id
    for u, v in list(G.edges()):
```

```
        node_u = G.nodes[u]
        node_v = G.nodes[v]

        if not (node_u['head_id'] == node_v['head_id'] or node_u['tail_id'] == node_v['tail_id'] or node_u['head_id'] == node_v['tail_id'] or node_u['tail_id'] == node_v['head_id']):
            removed_edges.append((u, v))
            G.remove_edge(u, v)

    # Convert the distance matrix to DataFrame
    manhattan_distance_df = pd.DataFrame(manhattan_distance_matrix, index=edge_data.index, columns=edge_data.index)

    return G, removed_edges, coordinates, manhattan_distance_df

# File paths and create the planar graph with the updated condition
graph, removed_edges, coordinates, manhattan_distance_df = create_planar_graph(edge_info_path)

# Print or return the DataFrame as needed
manhattan_distance_df
```

Out[ ]:

	0	1	2	3	4	5	6	7	8	9	...	181	182	183	184	185	186	187	188	189	190
0	0.000000	73.003746	153.616179	0.000000	0.000000	0.000000	0.0	0.0	0.000000	0.0	...	0.0	0.0	0.000000	0.00000	0.000000	0.0	0.0	0.0	0.0	0.0
1	73.003746	0.000000	161.051765	0.000000	0.000000	0.000000	0.0	0.0	0.000000	0.0	...	0.0	0.0	0.000000	0.00000	0.000000	0.0	0.0	0.0	0.0	0.0
2	153.616179	161.051765	0.000000	0.000000	0.000000	0.000000	0.0	0.0	0.000000	0.0	...	0.0	0.0	0.000000	0.00000	0.000000	0.0	0.0	0.0	0.0	0.0
3	0.000000	0.000000	0.000000	0.000000	149.188862	101.702658	0.0	0.0	0.000000	0.0	...	0.0	0.0	79.177271	0.00000	0.000000	0.0	0.0	0.0	0.0	0.0
4	0.000000	0.000000	0.000000	149.188862	0.000000	172.209026	0.0	0.0	0.000000	0.0	...	0.0	0.0	168.034761	176.37821	0.000000	0.0	0.0	0.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
186	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.0	0.0	79.296648	0.0	...	0.0	0.0	0.000000	0.00000	0.000000	0.0	0.0	0.0	0.0	0.0
187	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.0	0.0	0.000000	0.0	...	0.0	0.0	0.000000	0.00000	0.000000	0.0	0.0	0.0	0.0	0.0
188	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.0	0.0	0.000000	0.0	...	0.0	0.0	0.000000	0.00000	60.148283	0.0	0.0	0.0	0.0	0.0
189	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.0	0.0	0.000000	0.0	...	0.0	0.0	0.000000	0.00000	0.000000	0.0	0.0	0.0	0.0	0.0
190	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.0	0.0	0.000000	0.0	...	0.0	0.0	0.000000	0.00000	0.000000	0.0	0.0	0.0	0.0	0.0

191 rows × 191 columns

compute the shortest path to the closest flooded_init

In [ ]:
```
# Load edge information just once as it's common for all node data files
edge_info = pd.read_csv(edge_info_path)
node_id_mapping = {(row['tail_id'], row['head_id']): idx for idx, row in edge_info.iterrows()}

# List all .csv files in the directory
csv_files = glob.glob(os.path.join(training_folder_path, "*.csv"))

# Dictionary to hold all processed DataFrames
processed_dataframes = {}

# Loop through each file
for file_path in csv_files:
    # Load the data
    nodes_data = pd.read_csv(file_path)

    # Apply the mapping to create a new 'node_id' column
    nodes_data['node_id'] = nodes_data.apply(lambda row: node_id_mapping.get((row['tail_id'], row['head_id'])), axis=1)

    # Step 1: Check if each node is flooded_init or not
    # For this, you'll need a DataFrame or some structure that associates nodes with their flooded_init status
    # Assuming 'nodes_data' contains 'node_id' and 'flooded_init' status for each node
    nodes_data['distance from closest flooded_init'] = float('inf')  # Initialize with infinity

    # Identifying all initially flooded nodes
    flooded_nodes = nodes_data[nodes_data['flooded_init'] == True]['node_id'].tolist()

    # Step 2: Assign distance zero to initially flooded nodes
    for node in flooded_nodes:
        nodes_data.loc[nodes_data['node_id'] == node, 'distance from closest flooded_init'] = 0

    # Step 3: Calculate shortest path to the nearest flooded_init node for other nodes
    for index, row in nodes_data.iterrows():
        if not row['flooded_init']:  # If the node is not initially flooded
            shortest_distance = float('inf')
            # Find the shortest path to any of the flooded nodes
            for flooded_node in flooded_nodes:
                try:
                    # Calculate the path using the network graph G and update the shortest distance if it's lower
                    distance = nx.shortest_path_length(graph, source=row['node_id'], target=flooded_node, weight='weight')
                    if distance < shortest_distance:
                        shortest_distance = distance
                except nx.NetworkXNoPath:
                    # If no path exists, continue to the next flooded node
                    continue
            # Update the nodes_data with the shortest distance found
            nodes_data.at[index, 'distance from closest flooded_init'] = shortest_distance

        # Construct new file name and path
    # Get the new filename without the .csv extension for use as a key
    filename_without_extension = os.path.splitext(os.path.basename(file_path))[0]
    new_filename = filename_without_extension + '_with_distance_meters'

    # Save the updated DataFrame in the dictionary
    processed_dataframes[new_filename] = nodes_data
```

In [ ]:
```
# Assuming you want to display the first dataset in the dictionary
# Extracting the first key from the dictionary
first_dataset_key = list(processed_dataframes.keys())[0]

# Retrieve the DataFrame associated with the first key
first_dataset = processed_dataframes[first_dataset_key]

# Display the DataFrame
first_dataset
```

Out[ ]:

	head_id	tail_id	flooded_init	flooded_final	node_id	distance from closest flooded_init
0	151779659	153066427	False	False	0	784.212097
1	151779659	152426116	False	False	1	791.647683
2	151779659	152136099	False	False	2	630.595919
3	151779763	152437059	False	False	3	313.209560
4	151779763	152541741	False	False	4	459.567685
...	...	...	...	...	...	...
186	153295319	153341528	False	False	186	600.279463
187	153444946	153444951	False	False	187	212.096293
188	153456725	153456737	False	False	188	262.220705
189	1191806314	1191806355	False	False	189	563.864116
190	5387962358	5387962359	False	False	190	103.826152

191 rows × 6 columns