Data Mining Project

INFO-H423: SNCB DATA CHALLENGE

Adrian Patricio (000603789)
Bryan Harold Ouembe Welaze (000530686)
Filipe Albuquerque Ito Russo (000606886)
Lucia Fernández Sánchez (000606121)

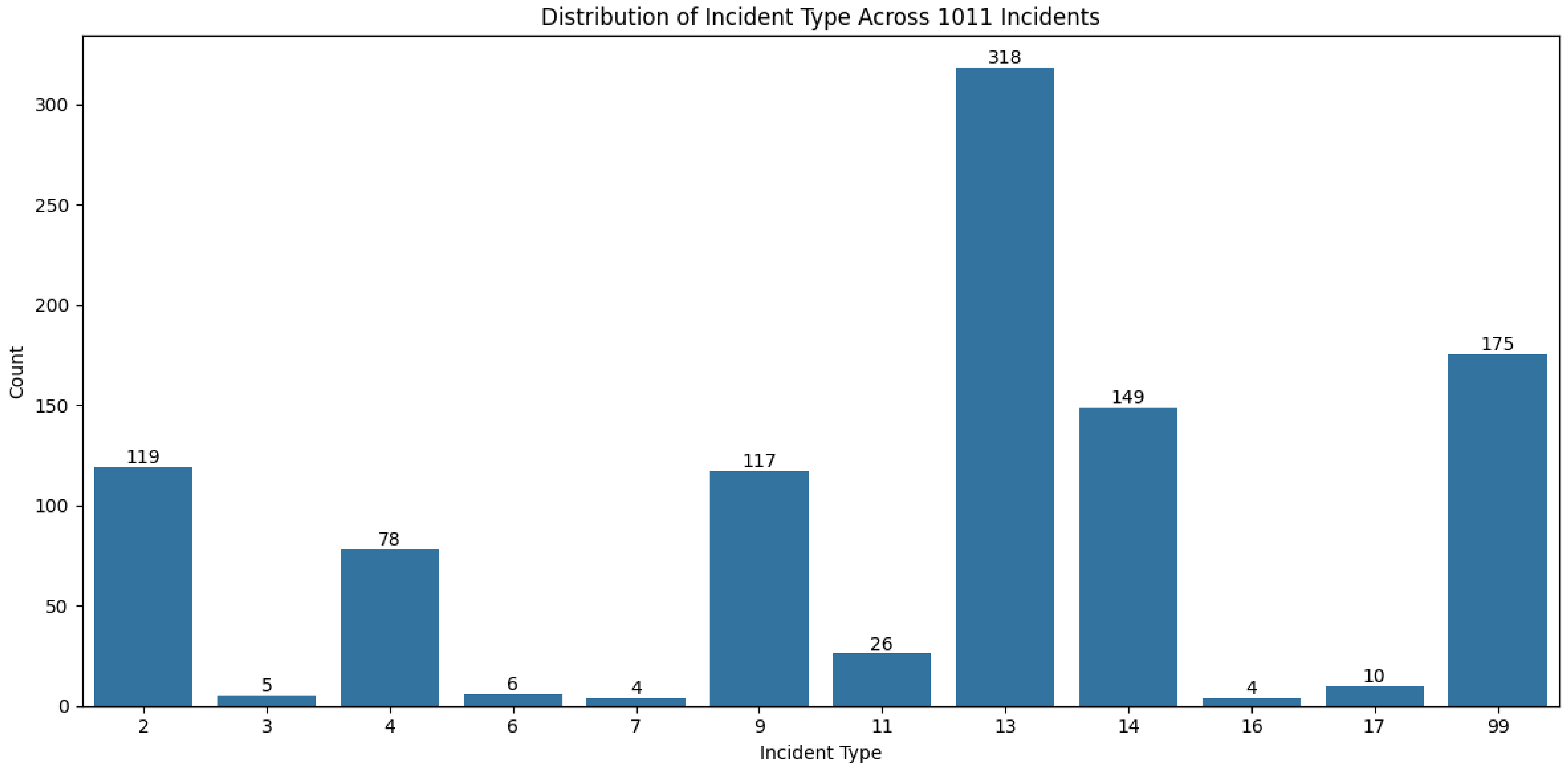
Outline

Data Exploration

SNCB Operational Incident Data

- Describes train incidents with:
  - vehicle sequence
  - **events sequence - main focus; the event codes occurring at most 4 hours before and 1 minute after the incident**
  - **second to incident sequence - time in seconds before or after the incident for each event**
  - approx lat, approx lon
  - train kph sequence
  - dj dc state sequence
  - dj ac state sequence
  - **incident type - target variable**

How many Incident Types are there?



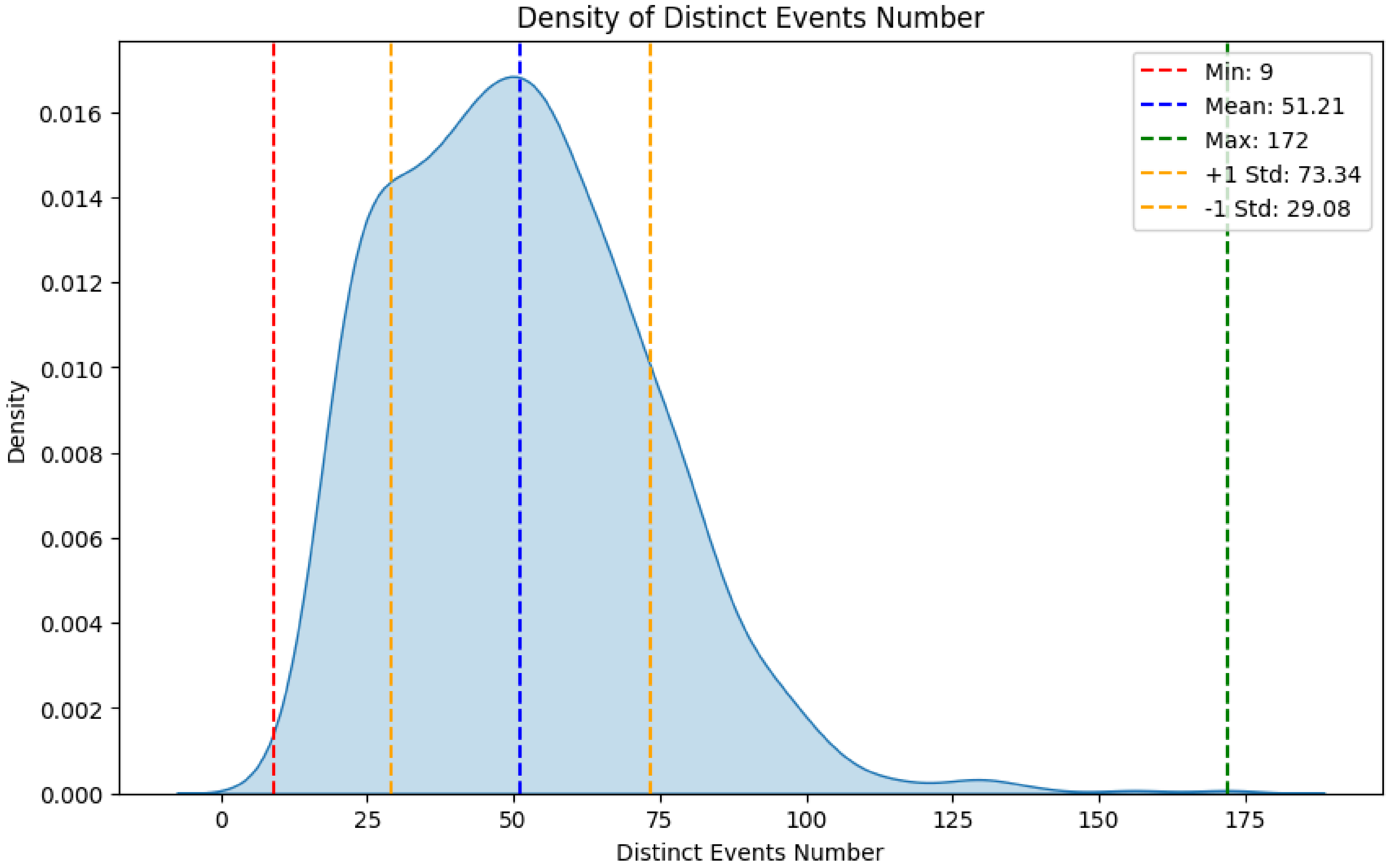
- There are **12 incident types in total** distributed unevenly
- There are 5 incidents types with 10 or less occurrences
- Incident 13 occurs the most with 318 occurrences

What Event Types occur?

- There are **917 unique event types**
- **Event Type 2708** has the highest frequency, occurring during 99.3076% of incidents
- **Event Type 2956** occurs the most at 291975 instances across all incidents
- There are **91 events** which only occurred once across all incidents

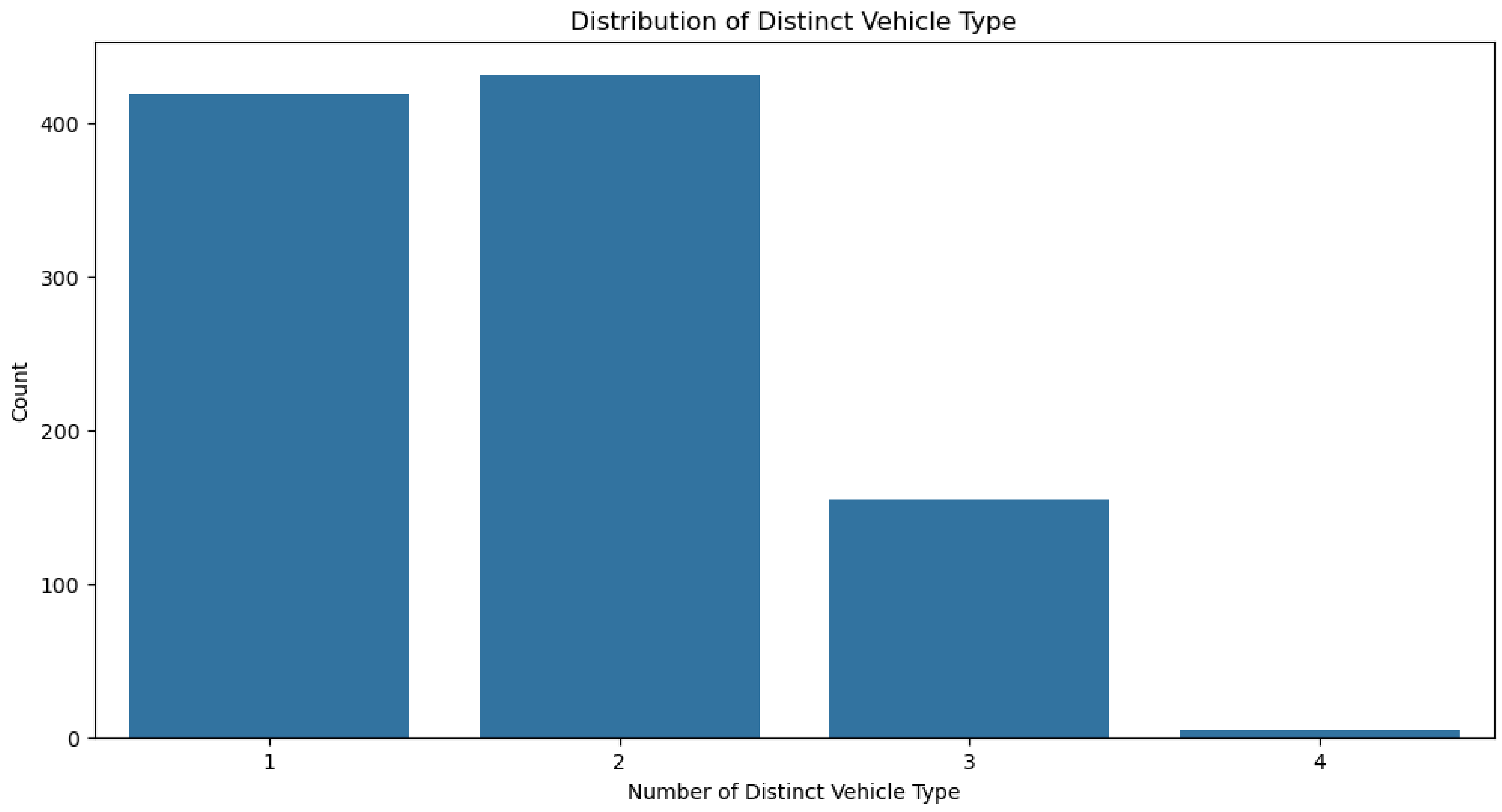
	event_type	count	event_freq
0	2956	291975	0.851632
1	3658	26608	0.880317
2	3636	26491	0.876360
3	4066	23018	0.941642
4	4068	22951	0.934718
...	...	...	...
853	2446	1	0.000989
852	1514	1	0.000989
851	2420	1	0.000989
849	586	1	0.000989
916	4186	1	0.000989

What Event Types occur?



- On average, there are around 51 distinct events per incident, a minimum of 9 events and a maximum of 172 events.

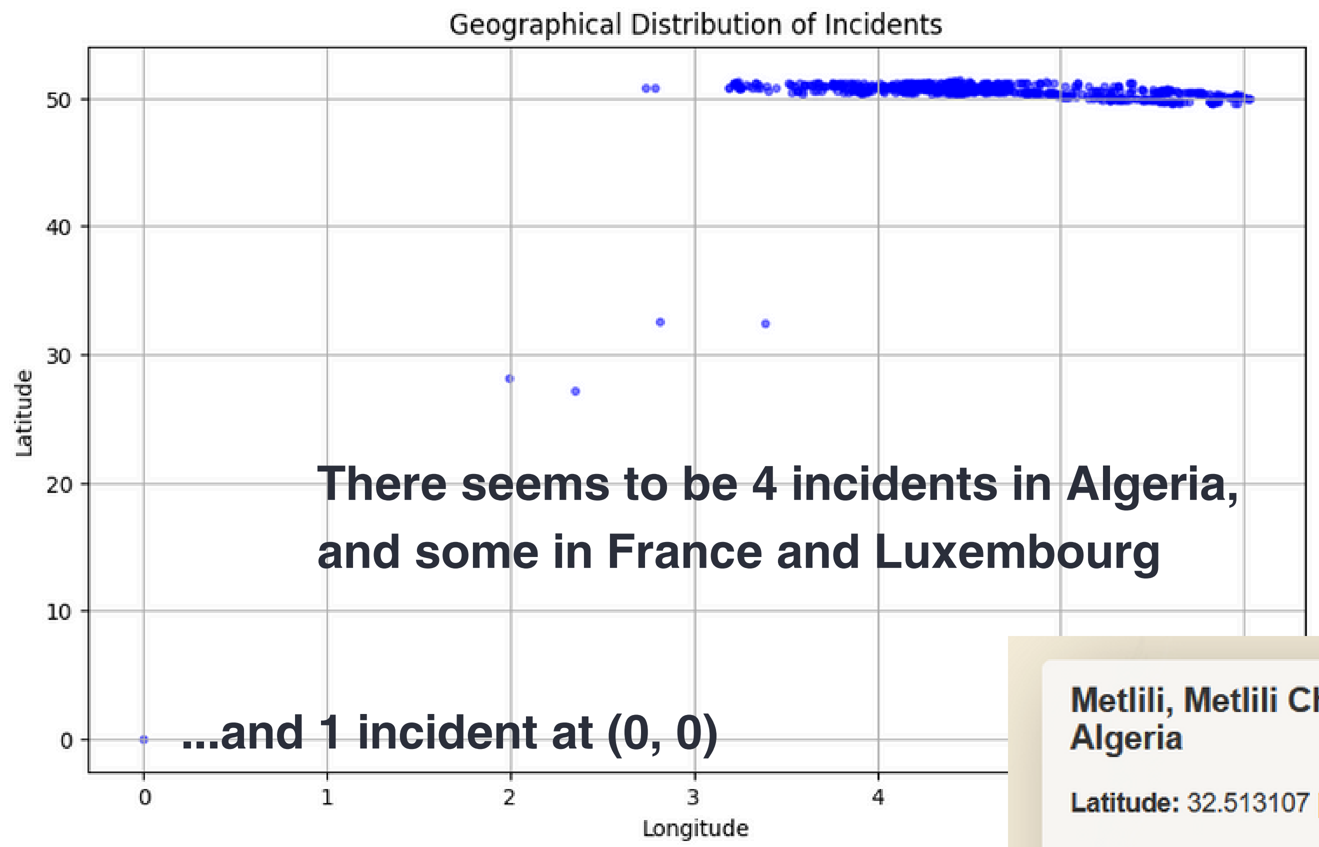
How many vehicles per incident?

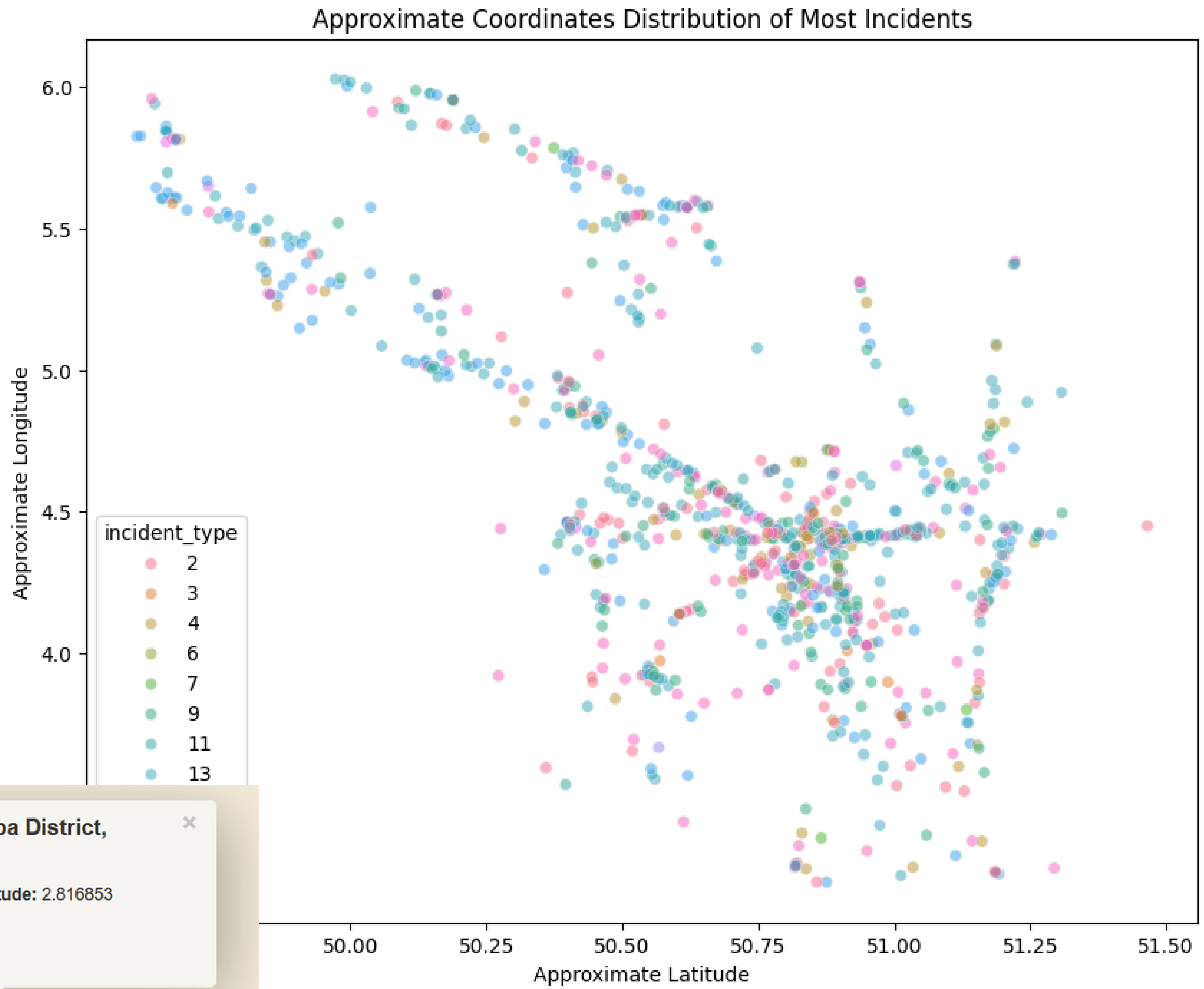


- There are at most 4 distinct vehicle types involved per incident, with the most incidents involving 2 vehicle types.

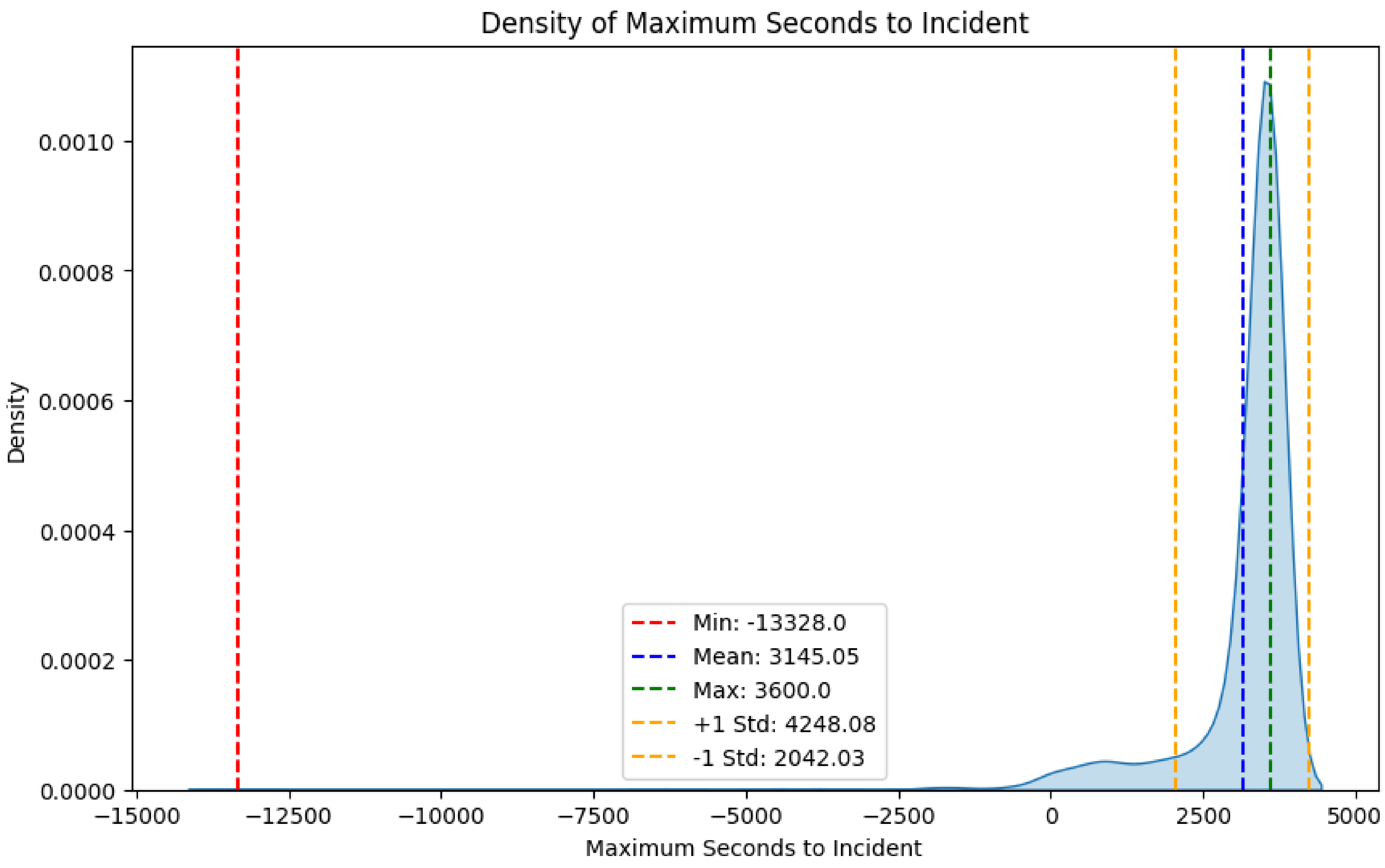
Where do incidents happen?

- There doesn't seem to be clear clusters of incidents occurring at the same coordinates





What time did the last event occur?



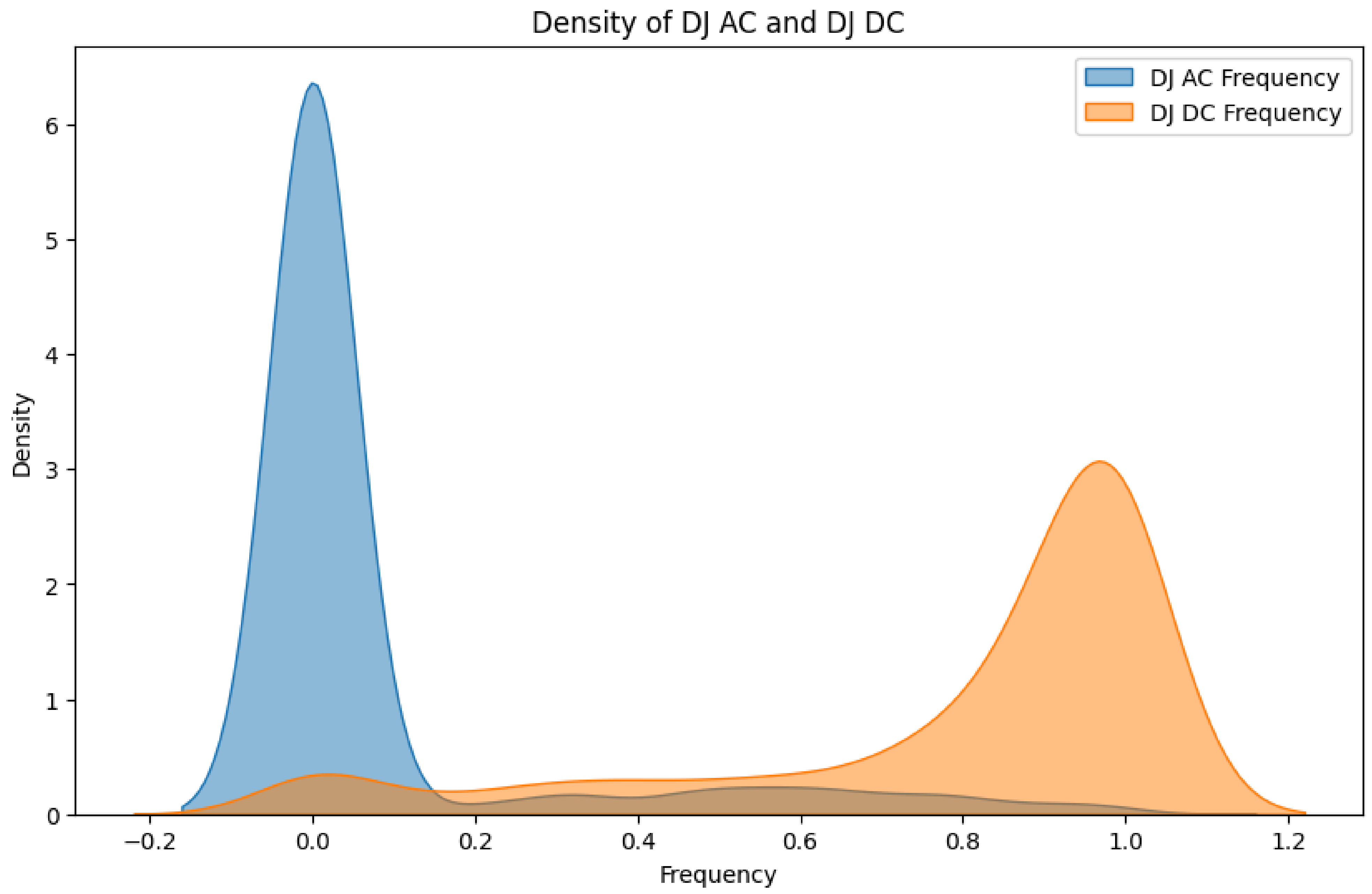
- Interestingly, there is at least one incident where the event sequence occurred entirely before the incident happened.

	incident_id	last time
50	4435091	-634.0
72	4436253	-83.0
155	4440081	-3760.0
290	4446877	-463.0
293	4447345	-4464.0
307	4448103	-1490.0
354	4450417	-1912.0
404	4452459	-652.0
423	4452911	-3762.0
454	4454633	-18.0
482	4455611	-224.0
486	4455925	-26.0
534	4457789	-151.0
543	4458167	-1108.0
561	4459157	-850.0
572	4459483	-1501.0
631	4461939	-1.0
658	4463659	-505.0
681	4464979	-832.0
729	4466973	-13328.0
762	4468421	-120.0
774	4468797	-2010.0
825	4601001	-6305.0
918	4606619	-130.0

Total Count: 24

To be precise, there are 24 incidents that only recorded events before the incident occurred.

How often are trains on AC/DC?



- It seems that trains are more frequently on DC power than on AC power
- On another note, **they are never on at the same time**, but they can be off at the same time (battery-powered)

The vehicles are battery-powered at least once for 724 incidents

	incident_id	dj_neither_freq
0	4432881	0.269327
2	4432955	0.400000
4	4433129	0.410405
5	4433267	0.080766
6	4433287	0.390995
...	...	...
1005	4611931	0.136882
1006	4611953	0.429851
1007	4611991	0.001479
1009	4612321	0.040847
1010	44233933	0.155756

724 rows × 2 columns

Any common subsequences?

```
Most common event subsequences for Incident Type 3 (length 3):
('4066', '3636', '3658'): 59 (100.00%)
('4068', '3636', '3658'): 55 (100.00%)
('2956', '2956', '2956'): 1665 (80.00%)
('3636', '3658', '2956'): 67 (80.00%)
('3658', '2956', '2956'): 62 (80.00%)
```

```
Most common event subsequences for Incident Type 6 (length 2):
('4110', '2854'): 21 (100.00%)
('2708', '2742'): 10 (100.00%)
('2852', '4110'): 16 (83.33%)
('2854', '4026'): 15 (83.33%)
('4168', '4140'): 11 (83.33%)
```

these only consider contiguous events for now

- We search for subsequences that occur **on the least-frequent incidents** to try to find some patterns that can distinguish them
- We got the longest subsequences for **incidents 3, 6, 7, 16, and 17**

Any common subsequences?

```
Most common event subsequences for Incident Type 7 (length 17):
(2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 4068): 20 (100.00%)
(2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 4066, 3636, 3658): 14 (100.00%)
(2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956): 508 (75.00%)
(2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 4068, 3636): 17 (75.00%)
(2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 4066): 15 (75.00%)
```

```
Most common event subsequences for Incident Type 16 (length 11):
(2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956): 255 (100.00%)
(2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 4068, 3636, 3658): 17 (75.00%)
(2956, 2956, 2956, 2956, 2956, 2956, 2956, 2956, 4066, 3636, 3658): 13 (75.00%)
(2956, 2956, 2956, 4068, 3636, 3658, 2956, 2956, 2956, 2956, 2956): 13 (75.00%)
```

```
Most common event subsequences for Incident Type 17 (length 2):
(2708, 2742): 24 (100.00%)
(2956, 2956): 2422 (90.00%)
(2742, 4026): 28 (90.00%)
(4066, 3636): 96 (80.00%)
```

Any common subsequences?

```
Subsequences:
(4068, 3636, 3658)
(4068, 3636, 3658)
(2742, 4026)
(2708, 2744)
(4068, 2708)
(4066, 3636)

Percentages per incident type:
13: 23.58%
99: 21.14%
14: 31.54%
2: 50.42%
9: 25.64%
4: 34.62%
11: 26.92%
17: 40.00%
6: 16.67%
3: 100.00%
16: 25.00%
7: 0.00%
```

```
Subsequences:
(2708, 2742, 4026)
(4120, 2956, 2956, 2956, 2956)

Percentages per incident type:
13: 33.33%
99: 26.29%
14: 36.91%
2: 45.38%
9: 23.08%
4: 37.18%
11: 34.62%
17: 70.00%
6: 16.67%
3: 40.00%
16: 0.00%
7: 0.00%
```

```
Subsequences:
(4110, 2854)
(2708, 2742)

Percentages per incident type:
13: 40.25%
99: 37.71%
14: 57.05%
2: 47.90%
9: 23.08%
4: 64.10%
11: 50.00%
17: 60.00%
6: 100.00%
3: 20.00%
16: 50.00%
7: 0.00%
```

```
Subsequences:
(2682, 2956, 2956, 2956, 2956)
(4124, 2956, 2956, 2956, 2956)
(4078, 2956, 2956, 2956, 2956)
(2956, 2956, 2956, 2956, 4068)
(2956, 2956, 4066, 3636, 3658)

Percentages per incident type:
13: 11.01%
99: 8.00%
14: 12.08%
2: 16.81%
9: 8.55%
4: 10.26%
11: 19.23%
17: 10.00%
6: 0.00%
3: 40.00%
16: 0.00%
7: 100.00%
```

- While individually, the subsequences might not really account for much, checking for **the presence of certain combinations of subsequences might possibly be indicators of certain incident types**
- Obviously, this was done for a small sample size, but it would still be interesting to explore this further

Data Cleaning and Transformation

Counting the number of events

- Get the frequency and count for the events across all incidents
  - Event 2956 had the highest count at 291975
  - **91 events** happened only once in the entire dataset
- If an event happens only once, **then we consider it as noise and remove it,** because it doesn't provide class information to cluster or classify the data according to it.

Feature Creation

- **Unique Vehicles Number:** u_vehicles_number
  - number of unique vehicles involved per incident
- **Unique Events Numbe**r: u_events_number
  - number of distinct event types per incident
- **Events Sequence Size**: event_sequence_size
  - number of events in the sequence

Feature Creation

- These features cover **variations in seconds_to_incident sequence.**

- **Minimum Seconds:** min_seconds
- **Median Seconds:** median_seconds
- **Max Seconds:** max_seconds
- **Mean Seconds:** mean_seconds
- **Delta Seconds:** delta_seconds
- **Number of events per second:** events_per_second
    - at most one event every 10 seconds

Feature Creation

- **AC Frequency:** dj_ac_freq
  - how often the train is powered by AC
- **DC Frequency:** dj_dc_freq
  - how often the train is powered by DC
- **AC/DC Off Frequency:** dj_neither_freq
  - how often the train is not powered by AC or DC; battery-powered

Feature Creation

- These features cover **variations in train_kph_sequence.**

- **Median Speed:** median_kph
- **Max Speed:** max_kph
- **Mean Speed:** mean_kph

Feature Selection and Engineering using TF-IDF

We are modeling our problem as a text mining classification task.

- remove events in the events_sequence that only happen once in the entire dataset, producing 826 distinct events to be used as features
- vectorize by counting raw occurrences
- apply TF-IDF to the counting
- apply L2 norm, to scale and preserve importance, so all features are in the same scale from 0 to 1 and the importance of the feature value is preserved row/document-wise.

Data Models

Base Models

- We explored several algorithms for classification:
  - **Naive Bayes**
  - **Random Forest**
  - **MLP**
  - **SVM**
  - **Logistic Regression**
  - **Gradient Boosting**

Base Classifier Setup

Data Splitting

- 80% of data (808 rows) used for training and validation
- 20% of data (203 rows) used for testing
- Stratified to maintain the class distribution of incident types

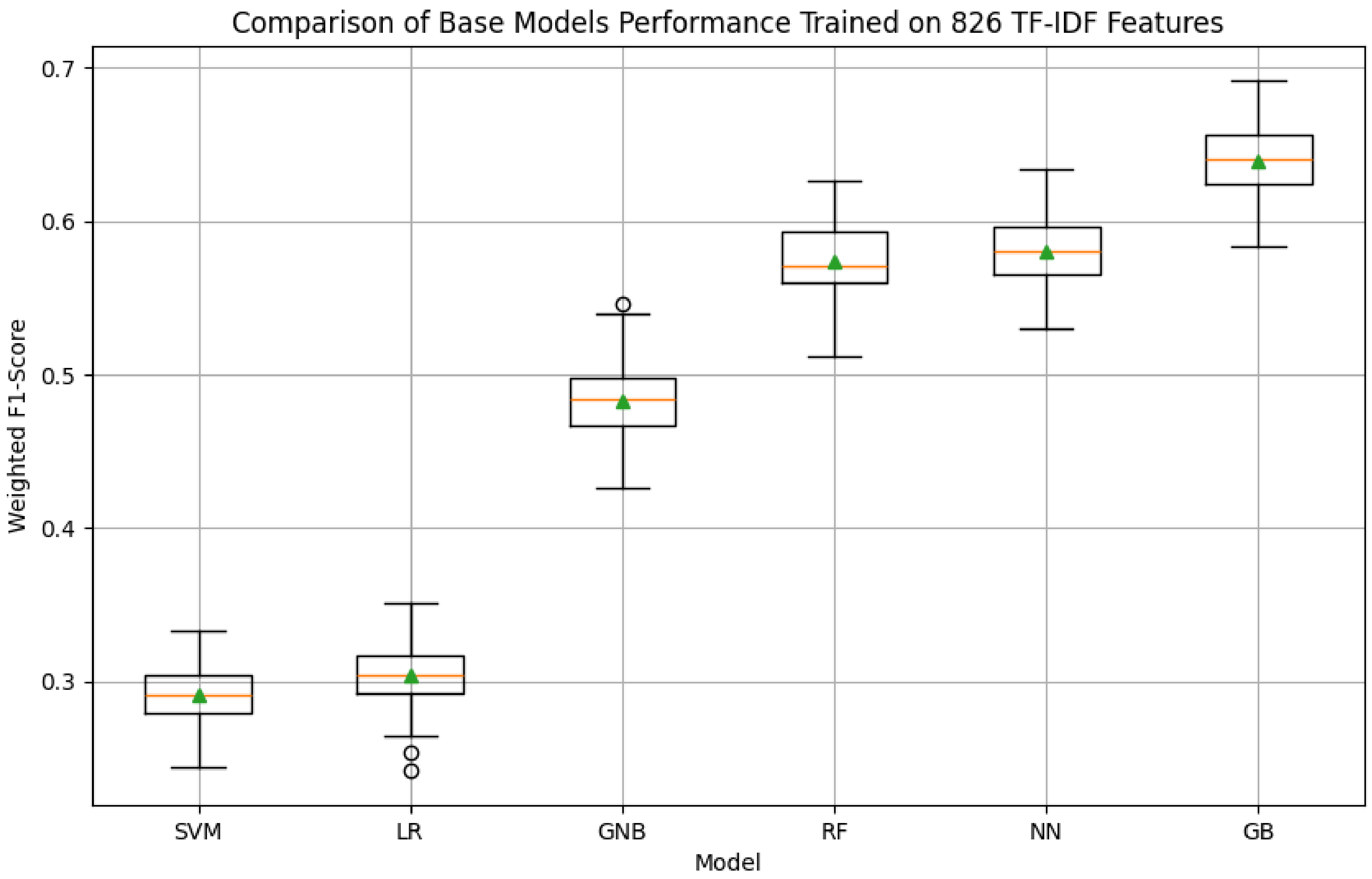
Model Training and Validation

- Trained with repeated stratified K-fold cross-validation
- 102 iterations of cross-validation - 3 folds, 34 repetitions
- Measured accuracy and weighted F1-score on the held-out validation sets

Testing

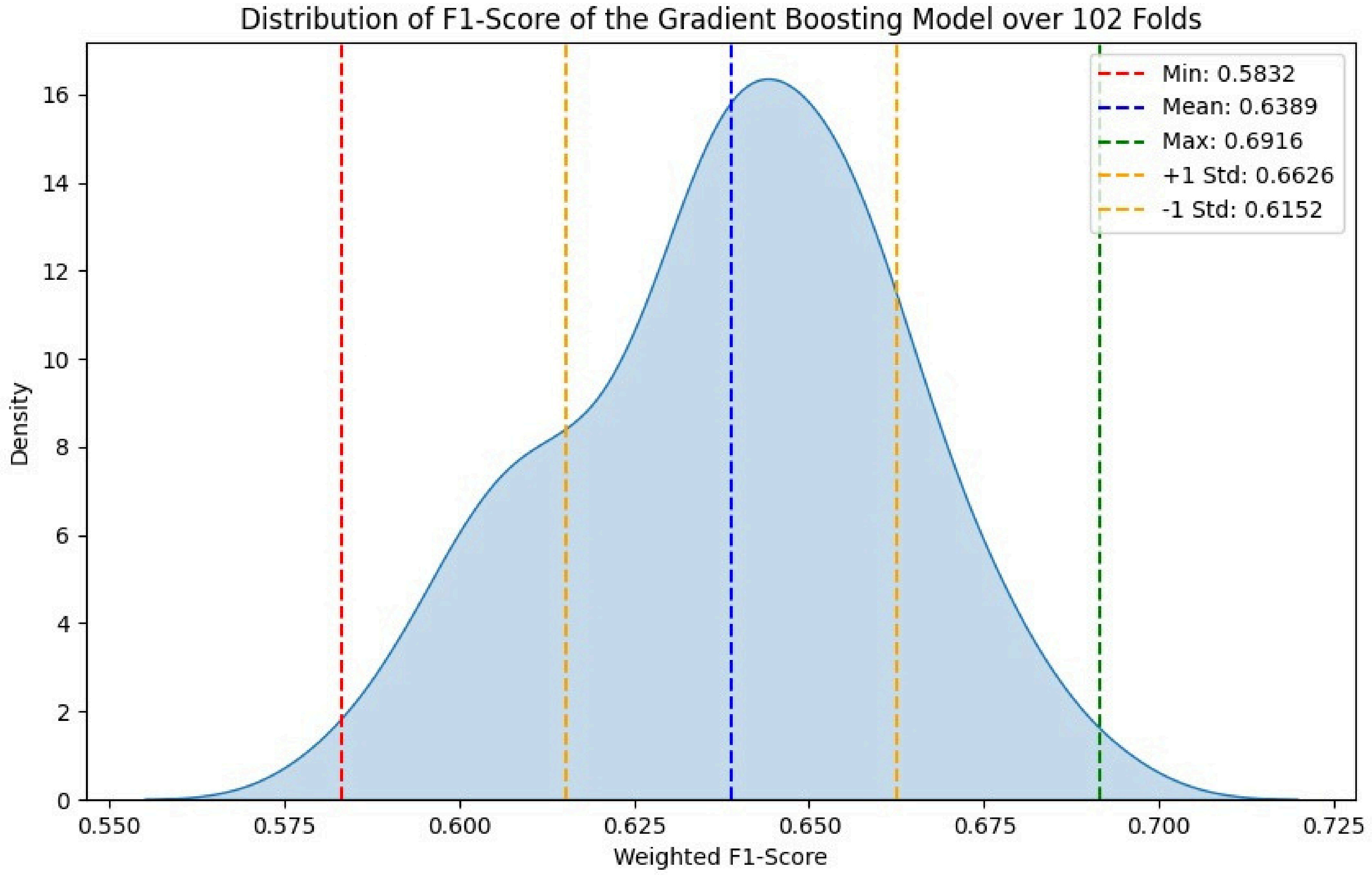
- Reserved test set is kept away and unseen from the model training and validation cycles
- Used to the evaluate performance of the pre-final model on unseen data and as training data increases

Base Model Performance



- Support Vector Machine (SVM) model exhibited the lowest mean weighted F1-score among the base models. In contrast, the **Gradient Boosting (GB) model achieved the best performance.**
- While performance is good for GB, it takes the longest among the 6 to train and validate. We will discuss time in detail further ahead.

GB Model Performance

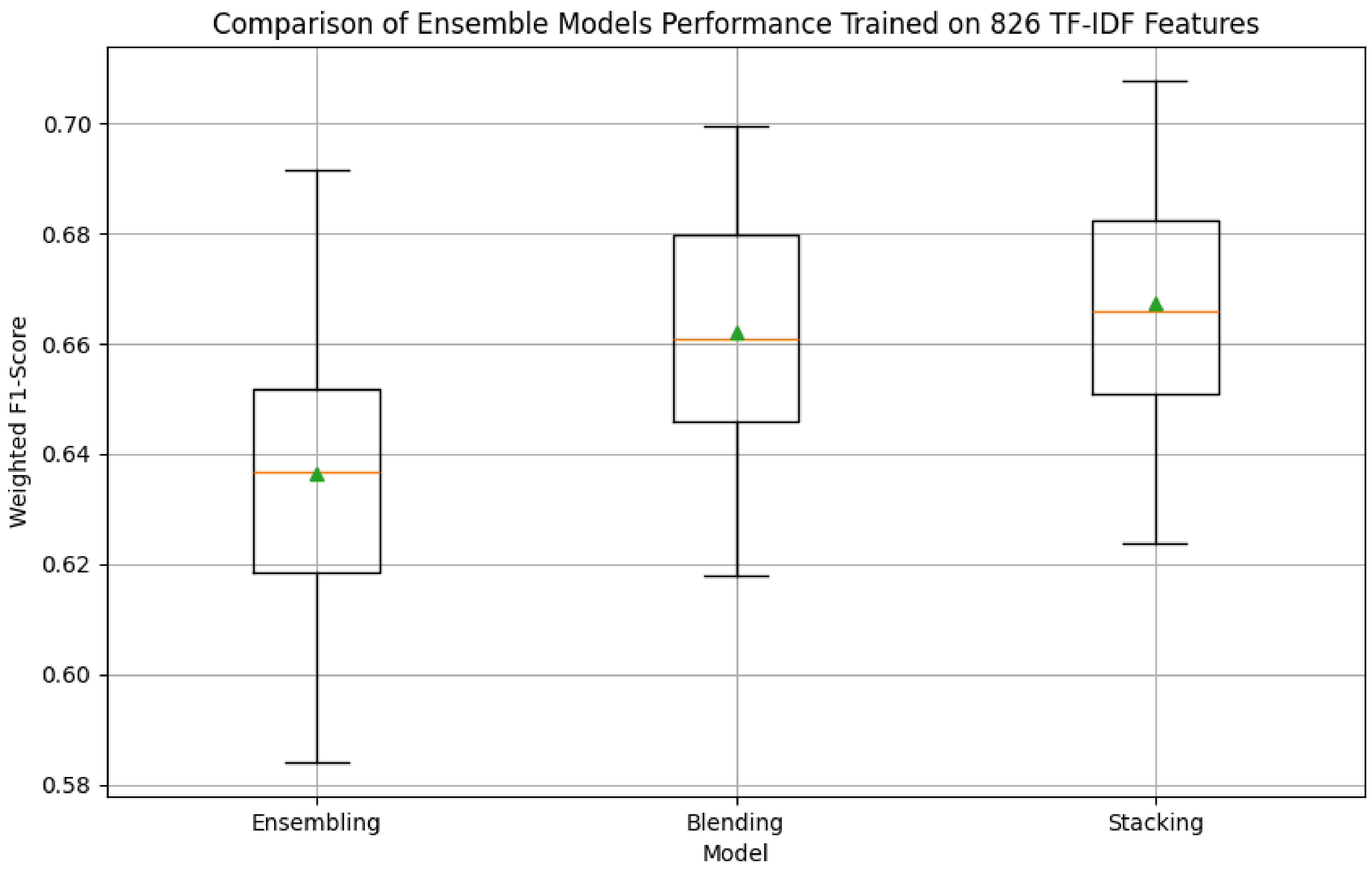


- Gradient Boosting showed moderate performance with a **mean F1-score of 0.6389** across 102 repetitions

Ensemble Models

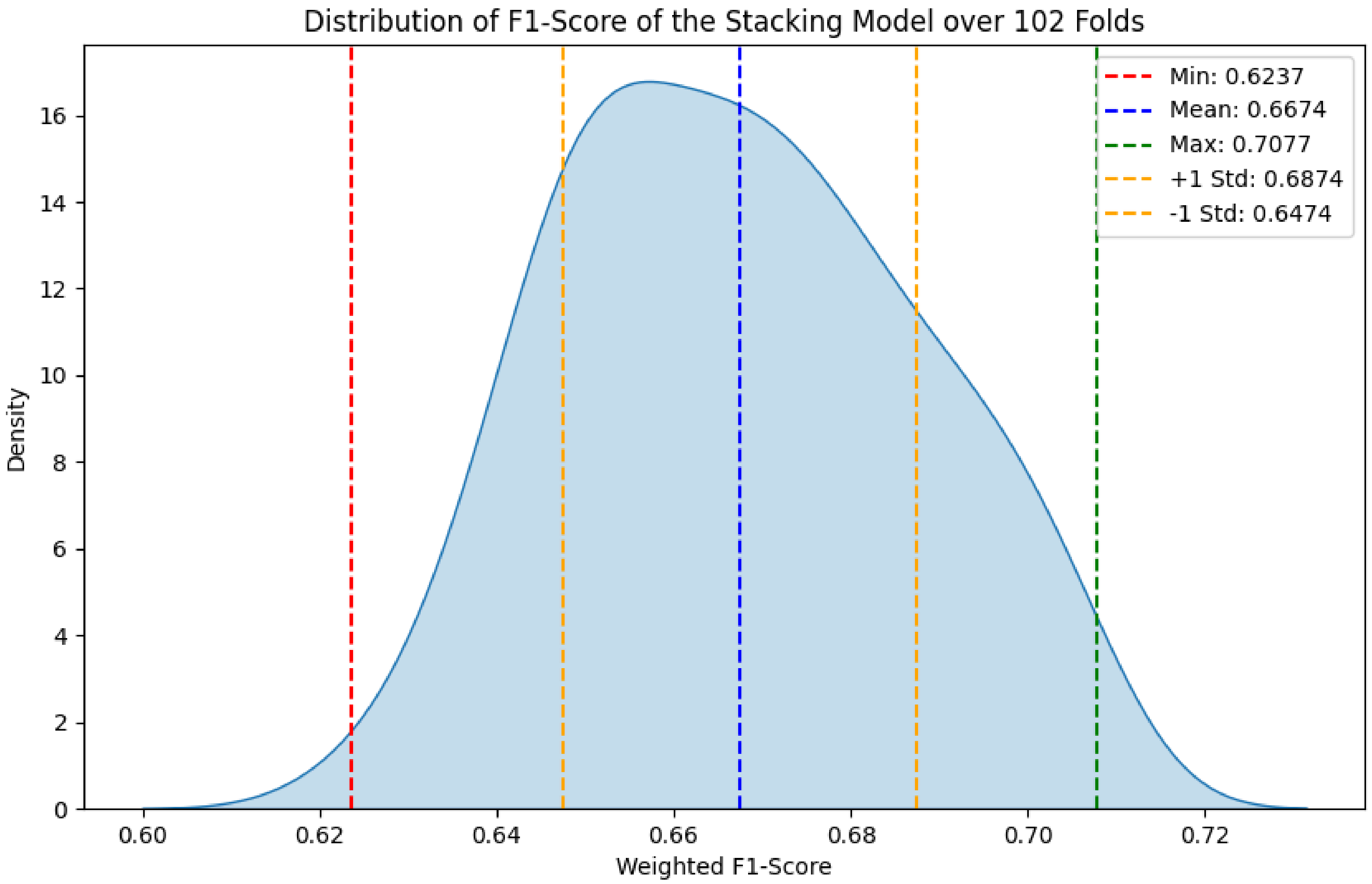
- The discovery of the best base models allowed us to ensemble them using different methods:
  - Ensembling (hard voting)
  - Blending (soft voting)
  - Stacking (meta-model)

Ensemble Model Performance



- The ensembling model exhibited the lowest mean weighted F1-score among the ensemble models. In contrast, **the stacking model achieved the best performance.**
- While performance is good for stacking, it takes the longest among the 3 to train and validate. We will discuss time in detail further ahead.

Stacking Model Performance



- Stacking showed improved performance with a **mean F1-score of 0.6674** across 102 repetitions

Can we improve?
Hyperparameter Tuning

- First, the three best-performing base models were used for hyperparameter tuning:
  - **Random Forest**
  - **Neural Networks**
  - **Gradient Boosting**
- At last, the improved behavior of the tuned base models was added to the ensemble models:
  - **Ensembling**
  - **Blending**
  - **Stacking**

Hyperparameter Tuning

Random Forest default parameters

```
class sklearn.ensemble.RandomForestClassifier(n_estimators=100, *,
criterion='gini', max_depth=None, min_samples_split=2, min_samples_leaf=1,
min_weight_fraction_leaf=0.0, max_features='sqrt', max_leaf_nodes=None,
min_impurity_decrease=0.0, bootstrap=True, oob_score=False, n_jobs=None,
random_state=None, verbose=0, warm_start=False, class_weight=None,
ccp_alpha=0.0, max_samples=None, monotonic_cst=None)
```

[source]

A random forest classifier.

A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. Trees in the forest use the best split strategy, i.e. equivalent to passing `splitter="best"` to the underlying `DecisionTreeRegressor`. The sub-sample size is controlled with the `max_samples` parameter if `bootstrap=True` (default), otherwise the whole dataset is used to build each tree.

Hyperparameter Tuning

Random Forest tested and set parameters

```
# Define the base model
model_rf = RandomForestClassifier(random_state = r_state,
                                  n_jobs = -1,
                                  criterion = "gini", # tested ["gini", "entropy", "log_loss"]
                                  max_features = 0.12, # tested ["sqrt", "log2", 10, 100, 0.12]
                                  class_weight = None # tested ["balanced", "balanced_subsample"]
                                 )
```

Hyperparameter Tuning

Neural Networks default parameters

```
class sklearn.neural_network.MLPClassifier(hidden_layer_sizes=(100,),
activation='relu', *, solver='adam', alpha=0.0001, batch_size='auto',
learning_rate='constant', learning_rate_init=0.001, power_t=0.5, max_iter=200,
shuffle=True, random_state=None, tol=0.0001, verbose=False, warm_start=False,
momentum=0.9, nesterovs_momentum=True, early_stopping=False,
validation_fraction=0.1, beta_1=0.9, beta_2=0.999, epsilon=1e-08,
n_iter_no_change=10, max_fun=15000)
```
[source]

Multi-layer Perceptron classifier.

This model optimizes the log-loss function using LBFGS or stochastic gradient descent.

Note: The default solver 'adam' works pretty well on relatively large datasets (with thousands of training samples or more) in terms of both training time and validation score. For small datasets, however, 'lbfgs' can converge faster and perform better.

Hyperparameter Tuning

Neural Networks tested and set parameters

```
# Define the base model
model_nn = MLPClassifier(random_state = r_state,
                          solver = "lbfgs",
                          hidden_layer_sizes=(100,) # tested [None, 50, 100, 150, 200], (100, 50)
                                                    # (200, 100), (100, 100), (100, 200), (50, 100), (668,)
                         )
```

Hyperparameter Tuning

Gradient Boosting default parameters

```
class sklearn.ensemble.GradientBoostingClassifier(*, loss='log_loss',
learning_rate=0.1, n_estimators=100, subsample=1.0, criterion='friedman_mse',
min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0,
max_depth=3, min_impurity_decrease=0.0, init=None, random_state=None,
max_features=None, verbose=0, max_leaf_nodes=None, warm_start=False,
validation_fraction=0.1, n_iter_no_change=None, tol=0.0001, ccp_alpha=0.0)
```

Gradient Boosting for classification. [source]

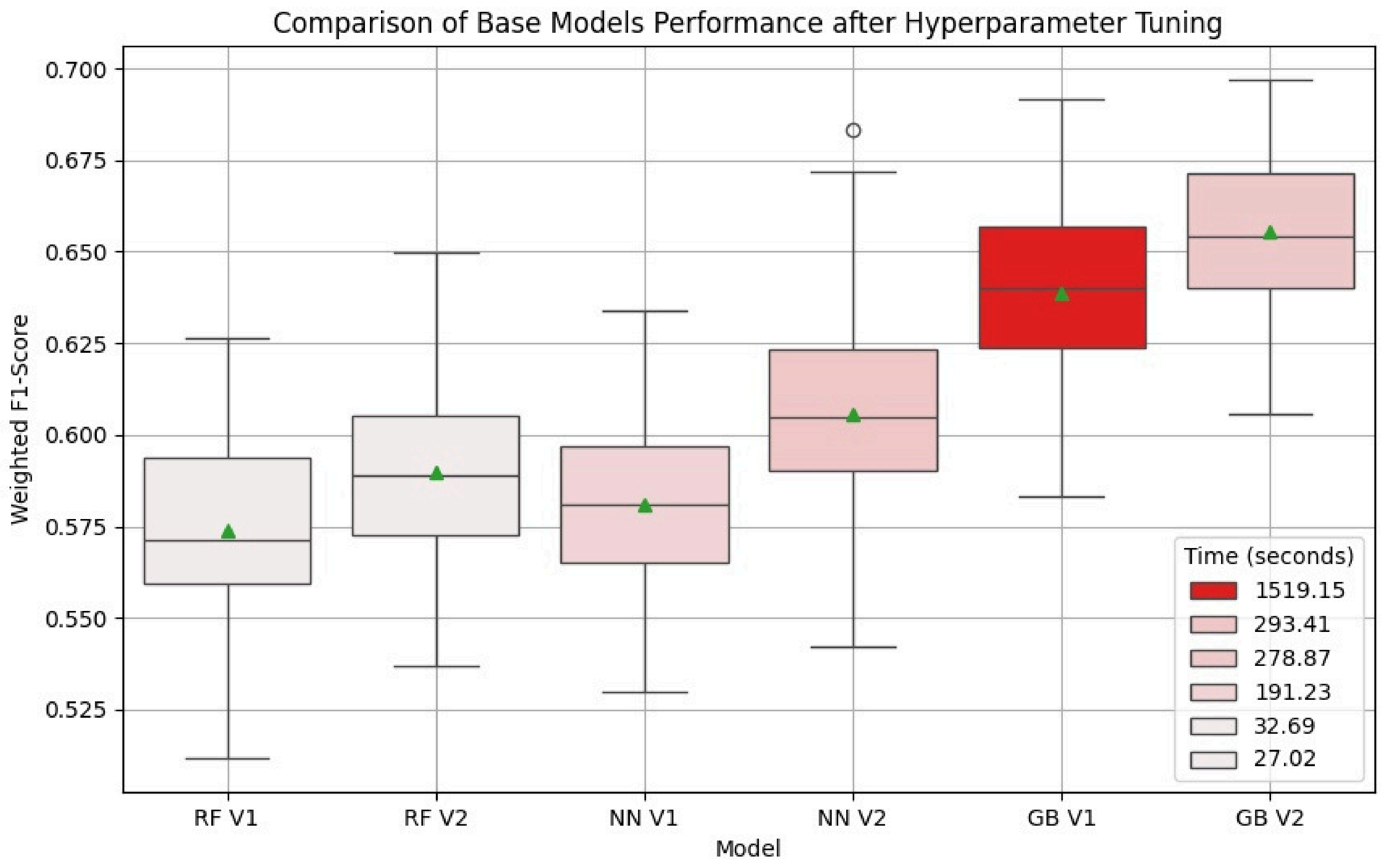
This algorithm builds an additive model in a forward stage-wise fashion; it allows for the optimization of arbitrary differentiable loss functions. In each stage `n_classes_` regression trees are fit on the negative gradient of the loss function, e.g. binary or multiclass log loss. Binary classification is a special case where only a single regression tree is induced.

Hyperparameter Tuning

Gradient Boosting tested and set parameters

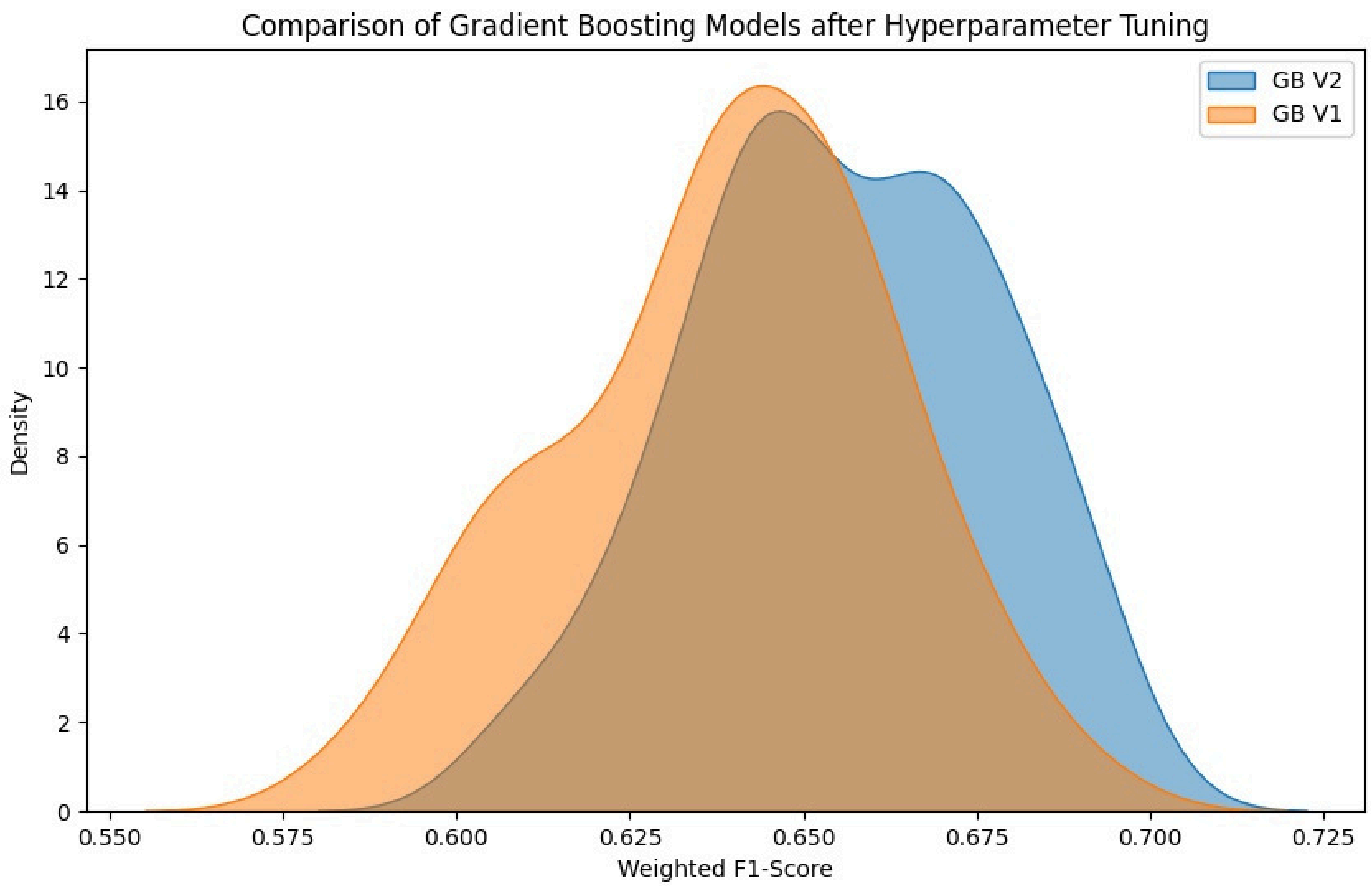
```
# Define the base model
model_gb = GradientBoostingClassifier(random_state = r_state,
                                       loss = "log_loss", # exponential works only for binary classification
                                       criterion = "friedman_mse", # tested ["squared_error"]
                                       max_features = 0.12 # tested [100, 0.15, 0.20]
                                      )
```

Base Model Performance



- **All three models improved** after hyperparameter tuning.
- **Gradient boosting still got the best performance overall**, with random forest (RF) being the worst out of the three.
- With the exception of the neural networks model (NN), which took longer to run, **the main performance improvement wasn't in the scoring, but in training and validation time.**

Model Performance



- Gradient Boosting improved from a mean F1-score of about **0.6389** to **0.6651.**
- Such improvement was solely due to **max_features = 0.12**, meaning that at step of the training the model would use only 12% of the total available features.

Ensemble Models

- In an attempt to improve the performance, we utilize several ensemble methods to combine the base classifiers.
- The three best-performing models are used: **Random Forest, Neural Networks,** and **Gradient Boosting**
- We attempted this on the **base models before and after hyperparameter tuning** for comparison.
- We explore three ensemble algorithms:
    - **Ensembling -** hard voting
    - **Blending** - soft voting
    - **Stacking** - using another classifier (i.e. logistic regression) as the final estimator, a meta model to be trained on the base models predictions

Voting Classifiers

- Ensembling (hard voting)
- Blending (soft voting)

```
# Define Ensembling (Hard Voting)
model_ensembling = VotingClassifier(estimators = base_models,
                                    voting = 'hard',
                                    weights = norm_weights)
```

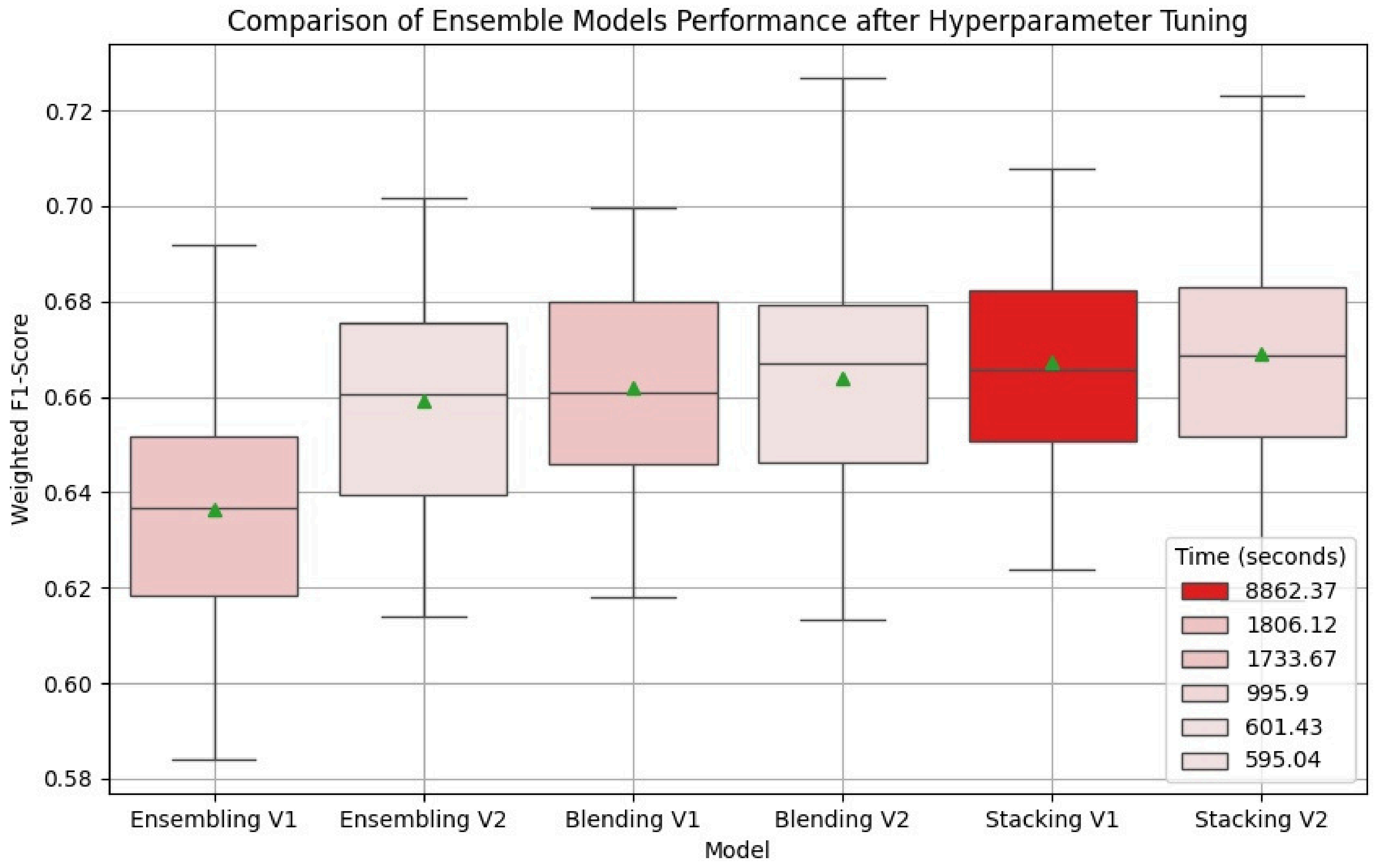
Stacking Model

```
# Define array of base models
base_models = [
    ('Random Forest', RandomForestClassifier(random_state = r_state, n_jobs = -1,
                                             criterion = "gini",
                                             max_features = 0.12,
                                             class_weight = None)),
    ('Neural Networks', MLPClassifier(random_state = r_state,
                                      solver = "lbfgs",
                                      hidden_layer_sizes=(100,))),
    ('Gradient Boosting', GradientBoostingClassifier(random_state = r_state,
                                                     loss = "log_loss",
                                                     criterion = "friedman_mse",
                                                     max_features = 0.12))
]

# Set up inner cross-validation of model_stacking
skf = StratifiedKFold(n_splits=3, shuffle=True, random_state=r_state)

# Define Stacking
model_stacking = StackingClassifier(estimators = base_models,
                                    final_estimator = LogisticRegression(),
                                    cv = skf,
                                    n_jobs = -1,
                                    verbose = 1
                                   )
```

Model Performance



- **Stacking the tuned models got the highest mean F1-score at 0.6689.**
- On the other hand, ensembling exhibited the worst performance out of all methods.
- Again, **the most noticeable improvement was on training and validation time.**

Performance Comparison

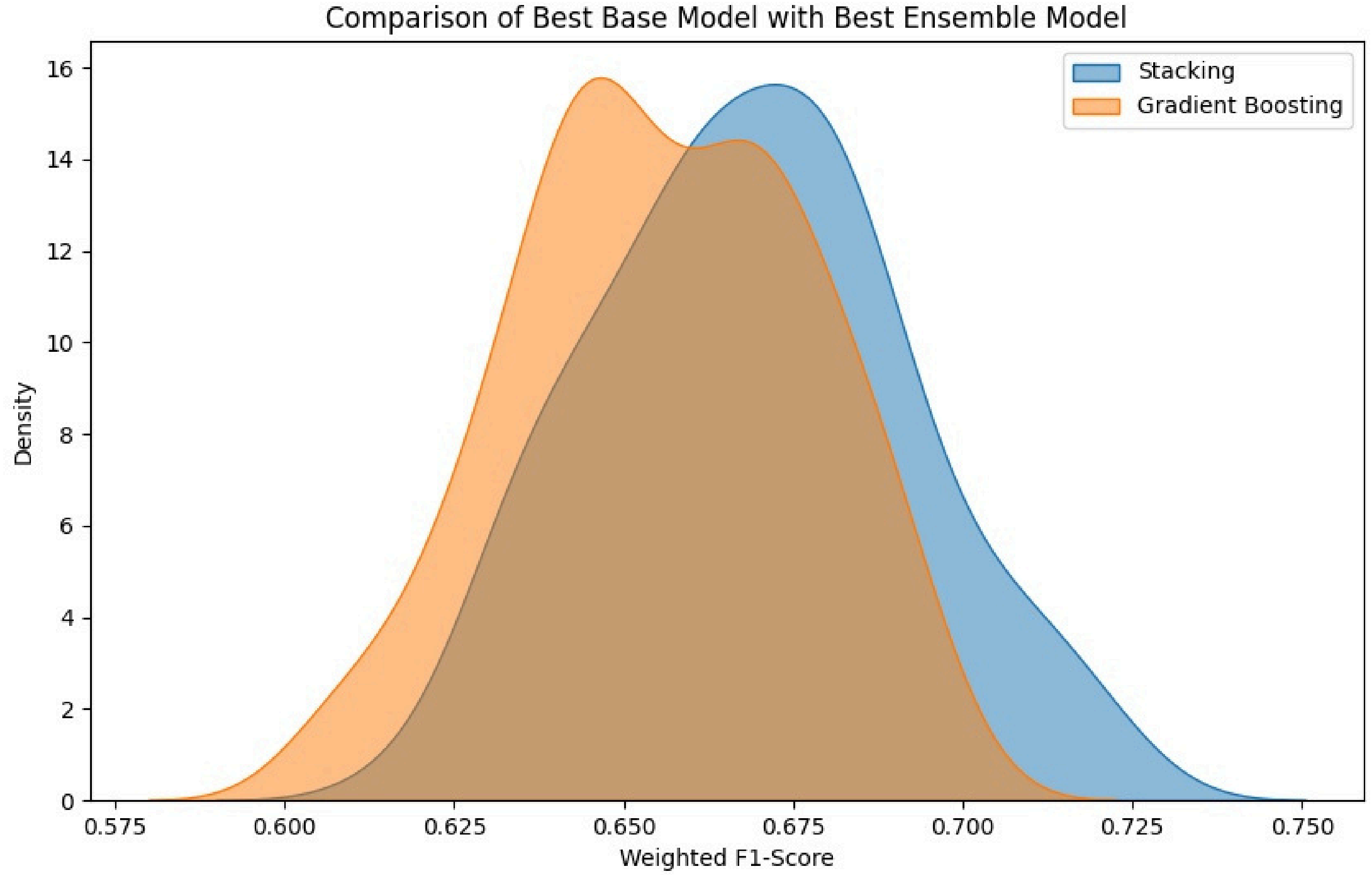


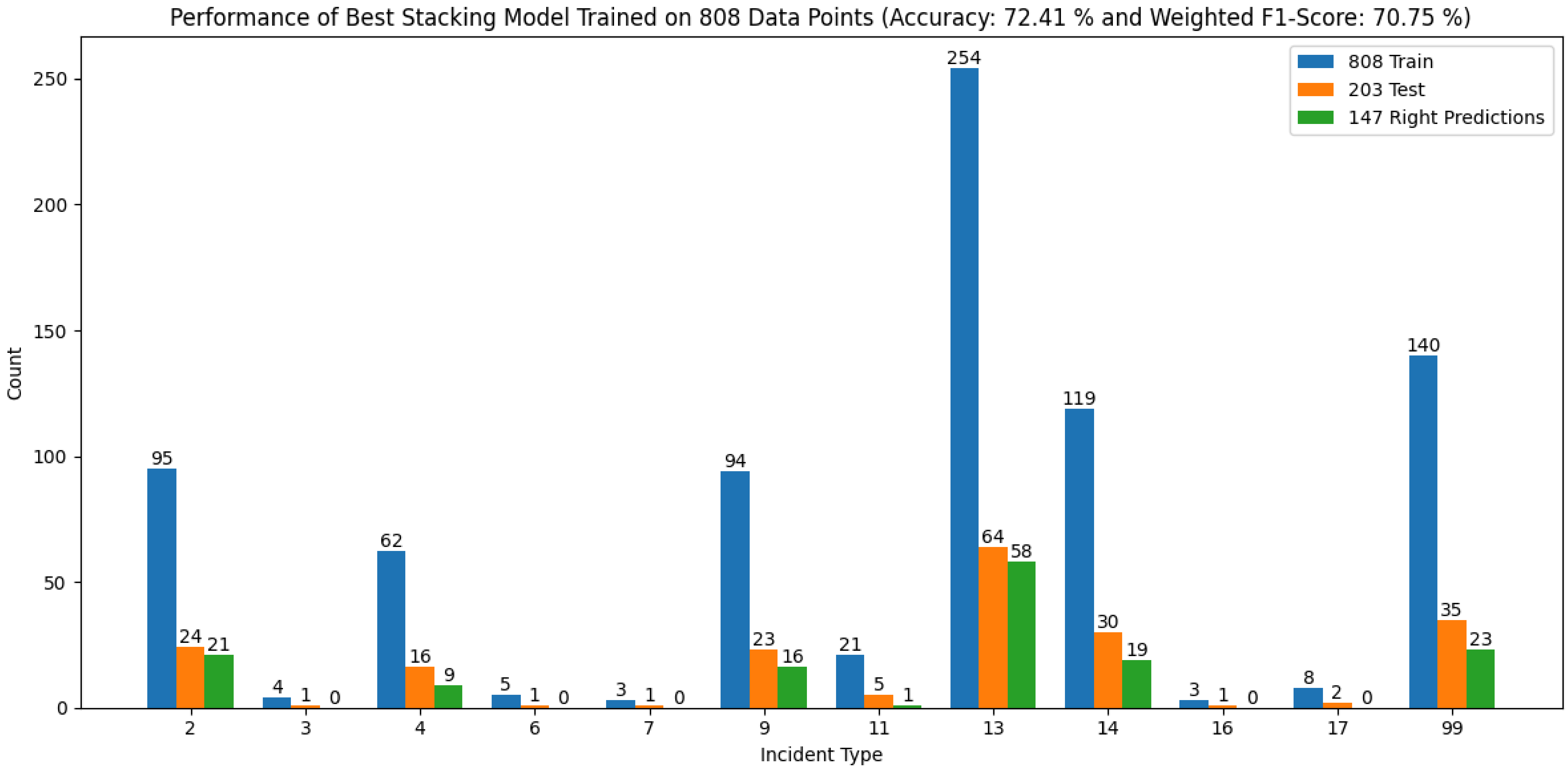
Figure 5: Learning machine performance: descriptive (red) and predictive (blue) performances across three different fleets (AM08, HLE18 and M7): typically the F_1-score is high. Nevertheless some incidents are poorly classified even during training. The M7 is a very recent fleet which explains why there is less data.

motivated by two factors: (1) staying away from Covid-19 low intensity period of operations and (2) choosing a period of low variations of vehicle on-board software versions. In terms of computational cost, the resulting model requires little power[6] and runs fast enough to open the potential for edge computing in the future.

By computing the F_1-score across all classes on out of training datasets, the predictive performance is typically around 80%, see figure 5. It is also clear that some incidents are poorly classified by the models even during training. On figure 6 the confusion matrix on the

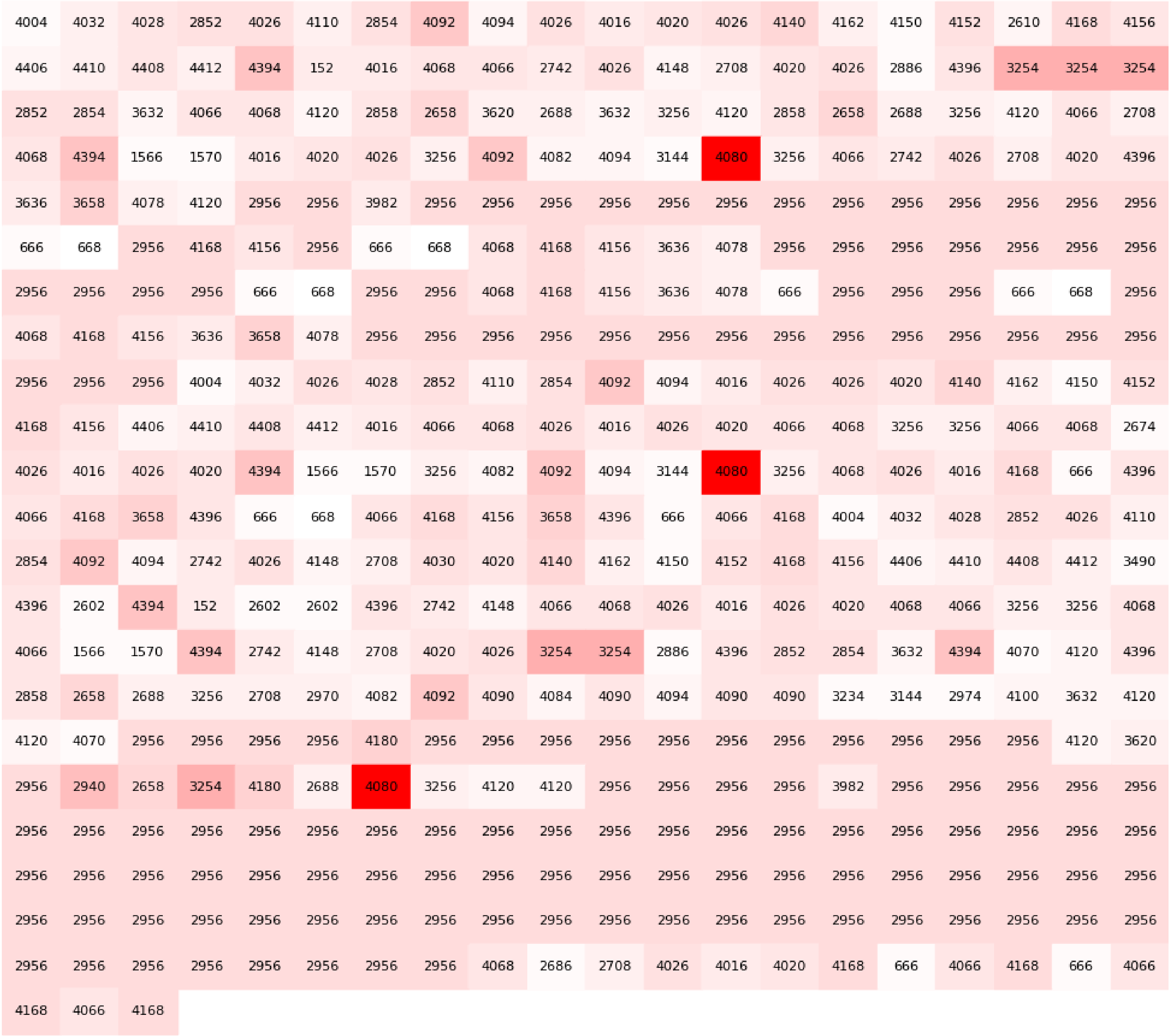
- As opposed to the GB model which got a mean F1-score of 0.6651, the **stacked ensemble model got a mean F1-score of about 0.6689.**
- In comparison, **the SNCB classifier got an F1-score of about ~0.80 across all classes.**

Performance per Class on Test Set



To note, the best performance per class was observed for incident types 13 and 2.

Events Heatmap from GB Feature Importance



Data from incident 44233933

Given an incident, we are able to generate a heatmap of its events sequence based on the importance given to them in the training of the gradient boosting model. This could be further developed to find and analyze important sub-sequences.

Conclusion and Next Steps

Limitations

- Increasing the number of data points, especially for underrepresented classes, could improve the classifier performance.
- There is a lack of contextual information about the incidents and events, even from the business side of the operation. What is the cost of incident type identification? In terms of time, money and personnel. Do they vary by incident type? Should we prioritize accuracy instead of weighted F1-score? What would be the financial impact of doing so? Is there an economic imbalance in the misclassification of one incident type in relation to the others?

Can we do better?

- Extract other features from the original dataset.
- Use time windows to select more interesting events or balance them better. Order and closeness to incident occurrence was not used.
- Explore other techniques to handle imbalanced classes.
- Engineer dedicated classifiers for more frequent incidents and less frequent incidents or for macro-classes of incident type. Can we join the incident types somehow?
- In the presence of more data, we could better explore approaches such as LSTM or RNN, which would improve pattern recognition and give a better generalization considering unseen data.