Lab Report for CapyMOA

Classification on the Covtype Dataset

Authors:

Ayan Ahmed
ayan.ahmed@polytechnique.edu

Rayyan Ahmed
rayyan.ahmed@polytechnique.edu

Lecturer:

Dr. Mariam Barry

November 4, 2024

Contents

1 Dataset Selection and Task **2**

2 Data Exploration **2**
2.1 Dataset Statistics . . . 2
2.2 Class Label Distribution and Trends in Target Value . . . 3
2.3 Plots of some Numerical Attributes . . . 4
2.3.1 Elevation . . . 4
2.3.2 Vertical Distance to Hydrology . . . 6
2.4 Plots of some Categorical Attributes . . . 7
2.4.1 Distribution of Feature 10 by Class Across Windows . . . 7
2.4.2 Distribution of Feature 11 by Class Across Windows . . . 8

3 Experimental Setting **9**
3.1 Algorithms . . . 9
3.2 Evaluation Metric . . . 11

4 Results and Analysis **11**

5 Concept Drift Analysis **15**
5.1 Analysis of Suspected Concept Drift in the Dataset . . . 15
5.2 Detection of Concept Drift . . . 16

6 Further Analysis with Neural Networks **20**
6.1 Architecture . . . 20
6.2 Training Strategy . . . 22
6.3 Results Obtained using this model . . . 23

Introduction

This report presents an analysis of the Covtype dataset using the CapyMOA framework. The goal is to perform a comprehensive classification task and evaluate model performance under different experimental settings. Additionally, concept drift analysis and further neural network-based classification experiments are conducted.

1 Dataset Selection and Task

The dataset chosen for this lab is the **Covtype** dataset, which is a classification task. This dataset contains various features representing forest cover types and a target class indicating forest type categories.

2 Data Exploration

2.1 Dataset Statistics

The Covtype dataset has multiple features and is structured for streaming data analysis. Key statistics:

1. **Number of Instances**: The dataset contains 581,012 samples.
2. **Size of the Snapshot Used**: We have set the max_instances (the maximum number of instances) to 51000, that is, we are using around one-tenth of the original number of instances.
3. **Number of Features**: There are 54 features in total, including:
    - **Continuous features**: 10 features related to elevation, slope, horizontal and vertical distances to hydrology, hillshade measures at different times of the day, and distances to fire points.
    - **Binary categorical features**: 44 binary features, with 4 representing wilderness areas and 40 representing soil types.
4. **Target Variable (Cover Type)**: The target variable is a multi-class label that represents different forest cover types. There are 7 classes:
    (a) Spruce/Fir
    (b) Lodgepole Pine
    (c) Ponderosa Pine
    (d) Cottonwood/Willow
    (e) Aspen
    (f) Douglas-fir
    (g) Krummholz
5. **Data Type**: Mixed, with both continuous (real-valued) and categorical (binary) features.
6. **Objective**: The main objective is to classify the cover type of each forest area based on the features provided. This is a multi-class classification problem.
7. **Feature Description**:
    - **Elevation**: Elevation in meters.
    - **Aspect**: Compass direction (0–360 degrees).
    - **Slope**: Steepness of the slope in degrees.
    - **Horizontal Distance to Hydrology**: Horizontal distance to nearest surface water feature.
    - **Vertical Distance to Hydrology**: Vertical distance to nearest surface water feature.
    - **Horizontal Distance to Roadways**: Horizontal distance to nearest roadway.
    - **Hillshade**: Measures of sunlight (hillshade) at different times (9 AM, noon, and 3 PM).
    - **Horizontal Distance to Fire Points**: Distance to nearest wildfire ignition point.
    - **Wilderness Areas**: Four binary indicators representing the wilderness area type.
    - **Soil Types**: 40 binary indicators for soil types.

2.2 Class Label Distribution and Trends in Target Value

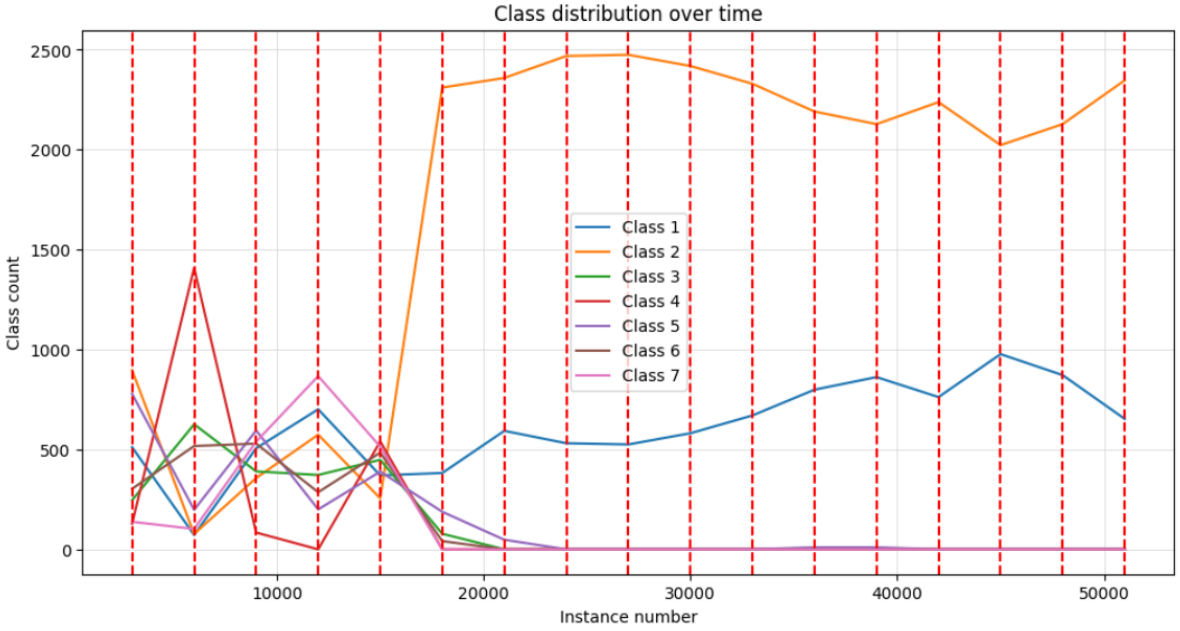


Figure 1: Class Distribution Over Time

Overall Class Label Distribution:

- **Class Dominance**: Class 2 (represented by the orange line) appears to be the most prevalent class overall, with the highest counts across much of the timeline.
- **Other Classes**: Class 1 (blue line) has moderate representation, while Classes 3, 4, 5, 6, and 7 have generally lower counts and fluctuate more irregularly.

Trends in the Target Value Over Time:

- **Class 2**: Initially, Class 2 has a sharp increase in representation around the 20,000 instance mark, reaching over 2,500 counts. It then shows a slight decline but maintains a high count for the remaining instances.
- **Class 1**: Class 1 gradually increases over time, with its count rising more steadily compared to the other classes, although it remains consistently lower than Class 2.
- **Other Classes (3, 4, 5, 6, and 7)**: These classes show significant fluctuations in the early instances (up to around 20,000) but generally stabilize at lower counts afterward, with some instances dropping to near zero.

Observed Patterns:

- **High Variation in Early Instances**: The class distribution is volatile at the beginning, with several classes experiencing rapid increases and decreases. This could indicate changes in the underlying data pattern or the presence of multiple types in the early subset of instances.
- **Stabilization Over Time**: As the instance count progresses, the distribution stabilizes, with Class 2 remaining dominant and Class 1 showing a steady increase, while the other classes have minimal and more stable counts.

Thus, the graph and the pie charts show that Class 2 is the most frequent and stable over time, while Class 1 shows an increasing trend. The other classes are less represented and more volatile in the early stages but stabilize with lower counts as the dataset progresses.

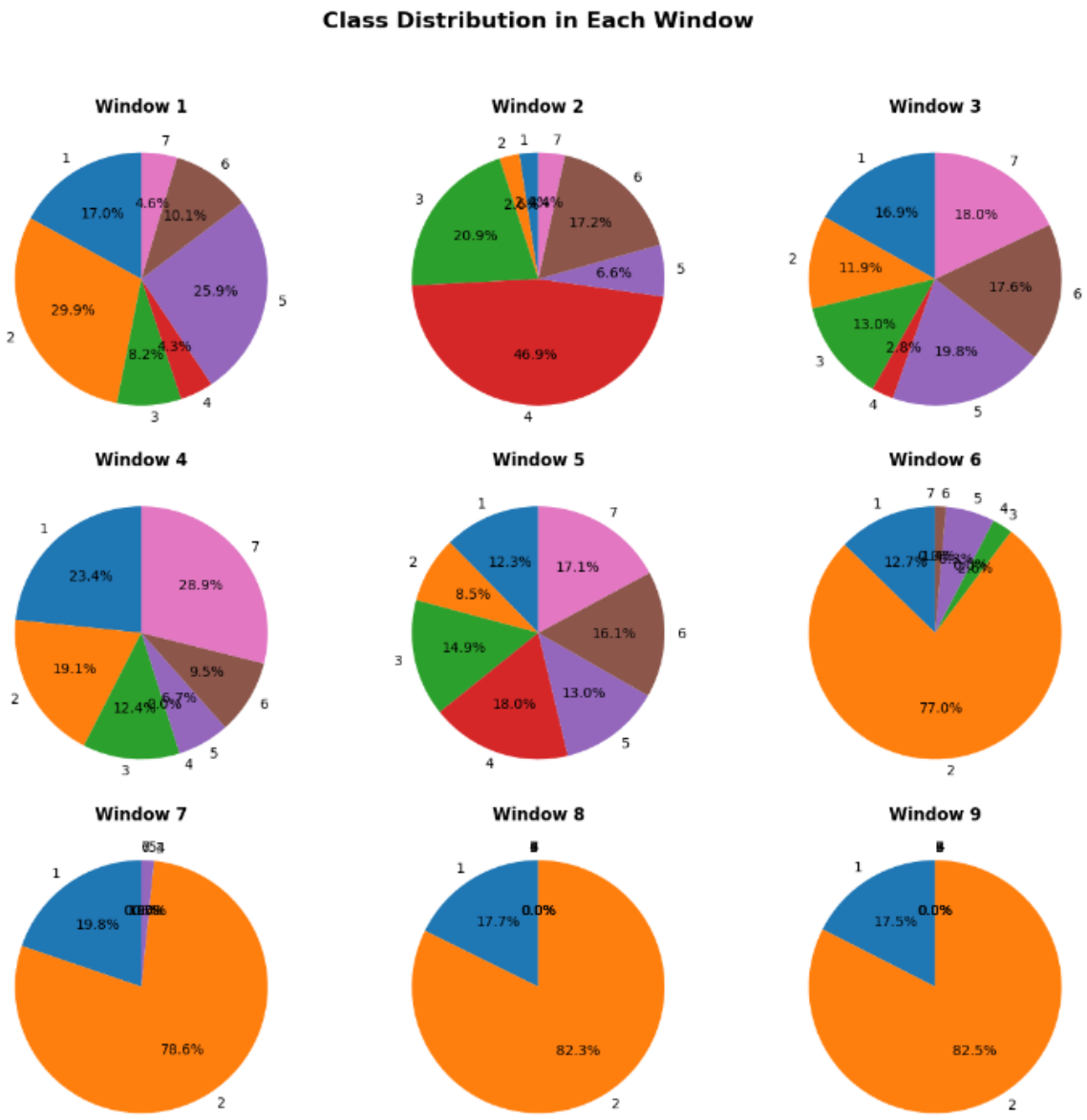


Figure 2: Class Distribution In Windows 1-9

2.3 Plots of some Numerical Attributes

2.3.1 Elevation

Histogram and KDE for Windows 1-8

Density Distribution Each plot represents a distribution of elevation values within eight different windows, as shown by both histograms and KDE (Kernel Density Estimate) curves.

Patterns Observed

- **Window 1 and Window 3**: These windows show bimodal distributions with two distinct peaks around elevations of 2500 and 3000, indicating two dominant elevation ranges in these windows.
- **Window 2**: This window shows a pronounced unimodal peak around 2500, suggesting a consistent elevation range within this window.
- **Windows 6, 7, and 8**: These windows are skewed towards higher elevations, with peaks clustering around 2750-3000 and tapering off. This suggests that these windows are located in generally higher elevation areas.

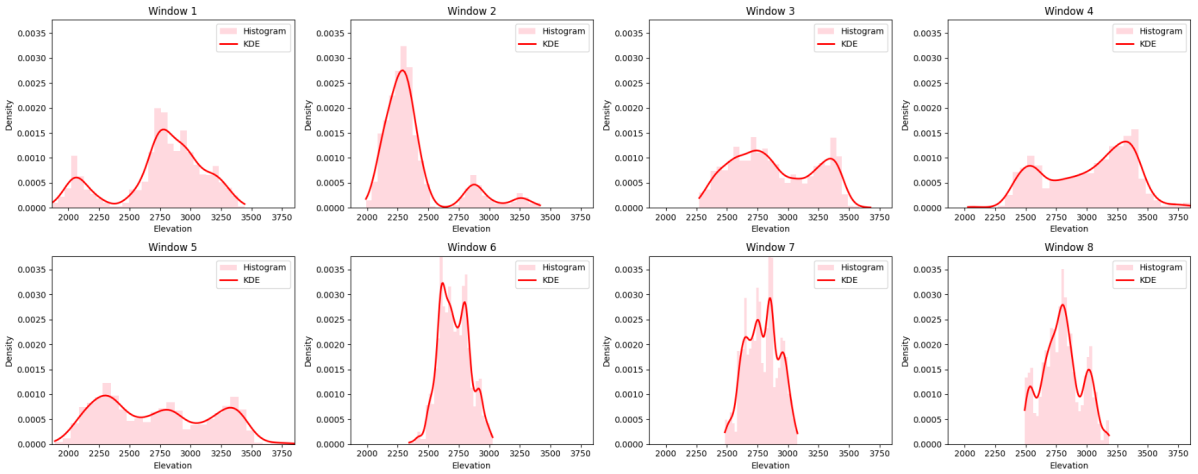


Figure 3: Elevation Histogram

Comparative Density Shifts Variations in the KDE shapes and histogram densities across windows imply spatial variability in elevation. For instance, the shift in dominant peaks between windows suggests that elevation varies significantly across different sections of the dataset.

Second Graph: Rolling Mean and Variance of Elevation Over Time

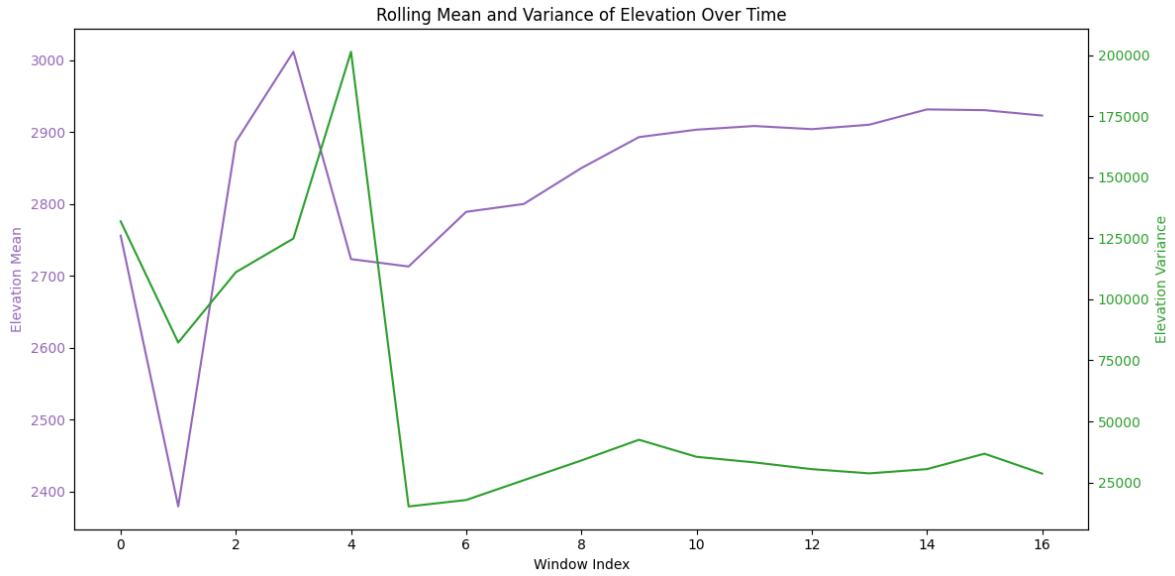


Figure 4: Rolling Mean and Variance

Rolling Mean of Elevation (Purple Line): The elevation mean fluctuates considerably across windows, peaking around Window 4 and then stabilizing in higher windows. After Window 5, the mean gradually increases until Window 8 and then stabilizes. This pattern might suggest a transition from lower to higher elevations in the middle of the dataset and leveling off as it approaches higher indices.

Rolling Variance of Elevation (Green Line) The variance shows a steep drop around Window 5, suggesting a reduction in elevation variability in that region. This could imply more uniform elevations within this window compared to others. Beyond Window 5, the variance remains low, indicating a more consistent elevation trend without much fluctuation within each window.

Interpretation and Insights

Topography Variation The distribution changes and rolling metrics suggest a dataset with varying topography. The regions represented by Windows 1-4 seem to have diverse elevation levels, while Windows

5-8 indicate a shift to more uniform elevation levels, possibly representing a plateau or a more consistent terrain.

Elevation Characteristics The elevation peaks and low variance in later windows suggest that the terrain becomes more homogeneous with increasing window index, potentially indicative of a single type of cover or land use. This observation fits well with our data which shows a huge class imbalance due to concept drifts or the change of the underlying distribution after 20000-25000 instances.

2.3.2 Vertical Distance to Hydrology

Histogram and KDE for Windows 1-8

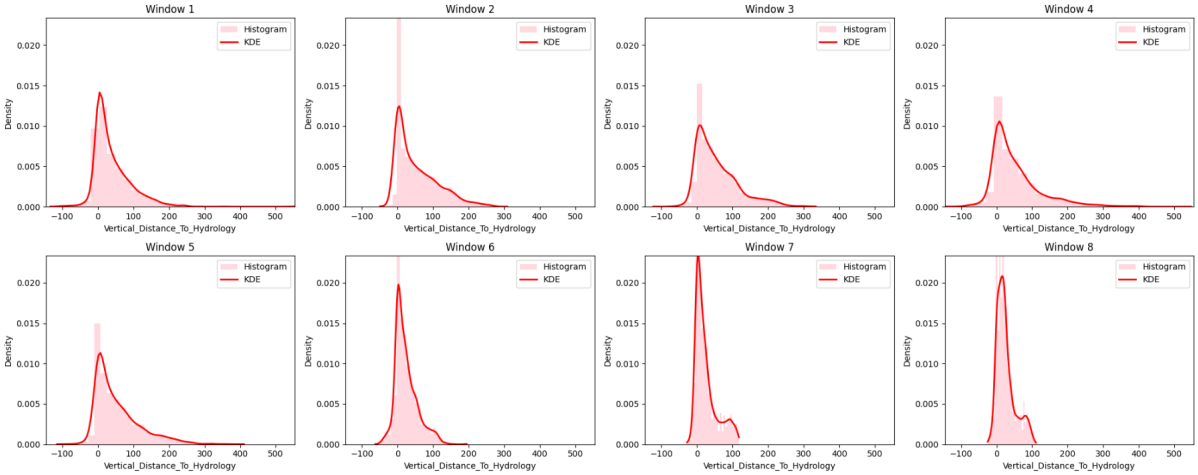


Figure 5: Vertical Distance to Hydrology

Density Distribution Each plot shows the distribution of `Vertical_Distance_To_Hydrology` values across eight different windows, illustrated with histograms and Kernel Density Estimate (KDE) curves. The KDE provides a smooth estimate of the density, showing the underlying distribution shape for each window.

Patterns Observed

- **Windows 1-4**: These windows demonstrate a positively skewed distribution, with most values concentrated near zero. There is a gradual decline in density as `Vertical_Distance_To_Hydrology` increases. This trend suggests that the majority of data points in these windows are relatively close to hydrological features (e.g., streams or rivers).
- **Windows 5-6**: These windows also show a positively skewed distribution, but the density peak is narrower and more concentrated around zero. This change in distribution shape indicates that data points in these windows tend to be even closer to hydrology features, with less spread across the distance range.
- **Windows 7-8**: The distributions are similarly skewed to the right, with density peaks closer to zero than in previous windows. However, the KDE curves are sharper and more narrowly peaked, suggesting that in these windows, values are even more tightly clustered near hydrological features, with few instances farther away.

Comparative Density Shifts The KDE shifts indicate a gradual change in `Vertical_Distance_To_Hydrology` across windows, with later windows (especially Windows 7 and 8) showing data concentrated near lower values. This trend implies a spatial pattern where instances become increasingly closer to hydrology features as the window index progresses, potentially signifying a consistent proximity to hydrological features in the latter part of the dataset.

Second Graph: Rolling Mean and Variance of Vertical_Distance_To_Hydrology Over Time

Rolling Mean of `Vertical_Distance_To_Hydrology` (Purple Line): The rolling mean shows significant fluctuation, with an initial increase up to Window 4, where it peaks. This peak suggests a relatively

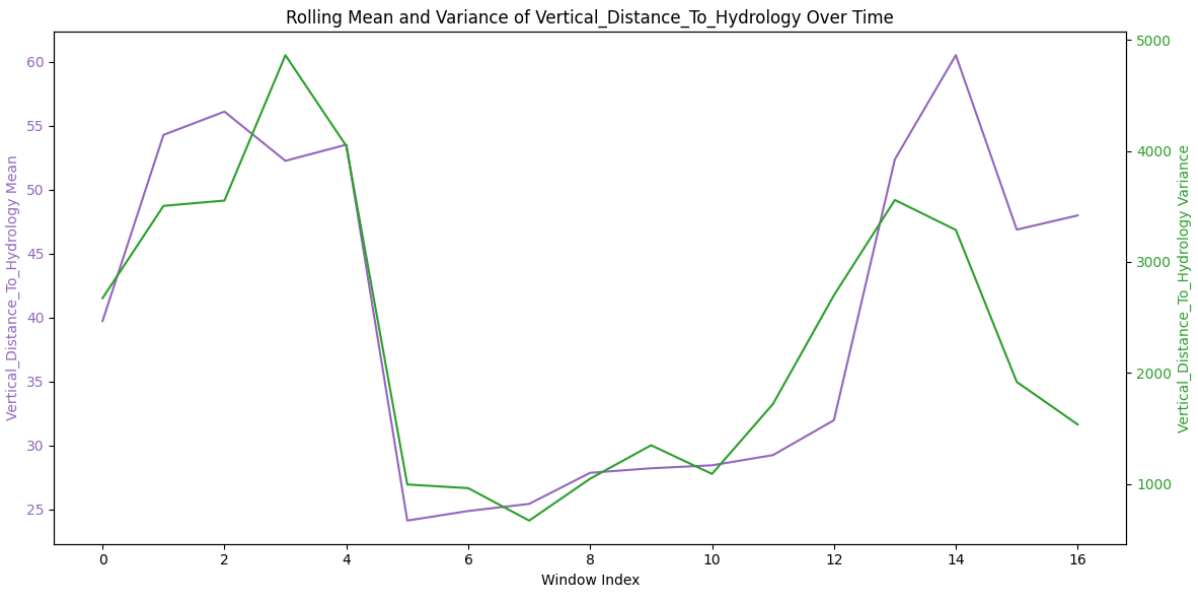


Figure 6: Rolling Mean and Variance

larger distance from hydrology features in the earlier part of the dataset. After Window 4, the mean drops sharply around Window 5, indicating a closer proximity to hydrology features. Beyond Window 5, the mean stabilizes at lower values, suggesting a trend towards a consistent, shorter distance to hydrology features.

Rolling Variance of `Vertical_Distance_To_Hydrology` (Green Line): The variance curve demonstrates a similar pattern, with high variability in earlier windows (especially around Window 4) and a steep decline after Window 5. This drop in variance suggests that, following Window 5, the distances to hydrology features become more consistent and less spread out. This pattern points to more homogeneity in the spatial arrangement with respect to hydrology in the latter part of the dataset.

Interpretation and Insights

Proximity to Hydrology Features The density distributions and rolling metrics indicate that in the initial windows (1-4), instances are relatively farther and more variably spaced from hydrology features, potentially reflecting a diverse or varied terrain in terms of elevation relative to hydrology. Starting from Window 5, there is a notable shift, with distances becoming smaller and more uniform, suggesting a closer and consistent proximity to hydrology in these later windows. This shift could indicate a change in the topography or land use type, where areas become more level and nearer to hydrological features.

Terrain Homogeneity The reduced variance in later windows implies a more homogenous terrain concerning hydrology. This uniformity might suggest the presence of a single, dominant cover type or a consistent landscape feature, aligning with the dataset's characteristic concept drift. The closer, more stable distances to hydrology in later windows may also contribute to a class imbalance as the terrain shifts, possibly due to a shift in cover types or environmental factors associated with proximity to water.

We could have included other plots and how they are varying with time but most of them would produce similar results with a few exceptions. Overall, the report was getting too long so we decided to keep it to two numerical and two categorical attributes.

2.4 Plots of some Categorical Attributes

2.4.1 Distribution of Feature 10 by Class Across Windows

Feature 10 Overview Feature 10 is a categorical variable, possibly a wilderness type, with values 0.0 and 1.0, distributed across different classes (labeled 1 through 7) within each window. The stacked bar charts for each window reveal how each class is associated with the values of Feature 10.

Class Distribution by Windows

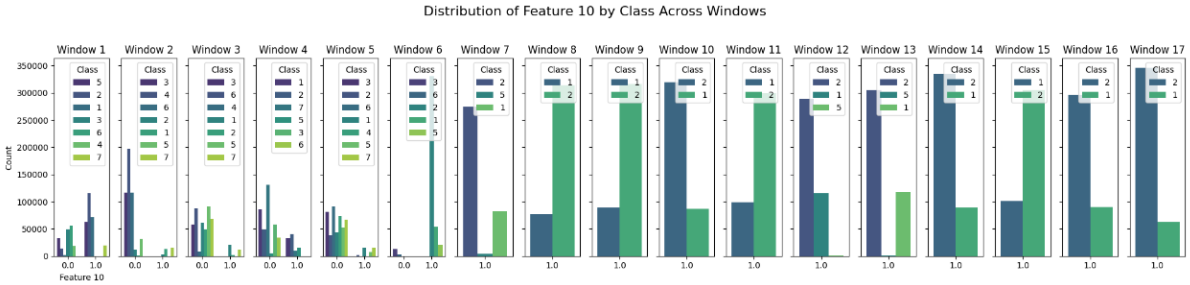


Figure 7: Distribution of Feature 10 across Class Windows

- **Windows 1-6**: In these initial windows, Feature 10 shows a broad distribution across multiple classes, indicating more variability in class composition. Notably:
  - Classes 5, 2, and 1 dominate the counts for Feature 10 in these windows.
  - There is a mix of Feature 10 values (0.0 and 1.0) across most classes, with a visible presence of both values, especially in Classes 5, 2, and 7.
- **Windows 7-17**: A shift occurs in the later windows, where the distribution becomes concentrated in fewer classes:
  - By Window 8 onwards, Classes 2 and 1 dominate, with very high counts for one of the values of Feature 10.
  - Feature 10 shows mostly the value 1.0 in later windows, suggesting a significant reduction in variability across classes.

Class Concentration Trends The progression from Windows 1-6 to Windows 7-17 indicates a transition towards fewer classes (primarily Classes 2 and 1) with concentrated values of Feature 10, mostly at 1.0. This pattern suggests a potential shift in underlying characteristics or an emerging dominance of specific classes in the dataset as it progresses, possibly due to concept drift or class imbalance.

2.4.2 Distribution of Feature 11 by Class Across Windows

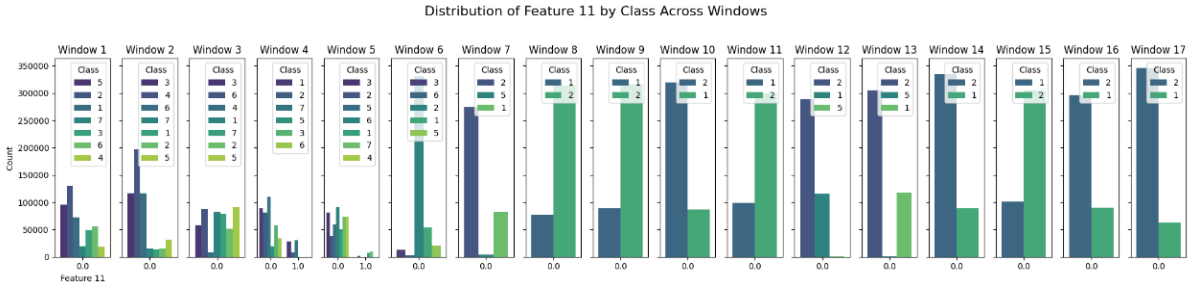


Figure 8: Distribution of Feature 11 across Class Windows

Feature 11 Overview Feature 11 is another categorical variable, possibly another variable for a particular type of wilderness, with values 0.0 and 1.0, distributed across the same set of classes within each window. Similar to Feature 10, the distributions reveal changes in class prominence and feature values across the windows.

Class Distribution by Windows

- **Windows 1-6**: The initial windows exhibit a high diversity of classes associated with both values of Feature 11:
  - Classes 5, 2, 3, and 7 are prominent in these windows.
  - Feature 11 values (0.0 and 1.0) are evenly represented across classes, indicating a varied distribution pattern.

- **Windows 7-17**: Similar to Feature 10, there is a shift towards fewer dominant classes with less variability in Feature 11 values:
  - Starting from Window 7, Classes 2 and 1 become increasingly dominant.
  - The value 0.0 for Feature 11 becomes more frequent in these later windows, especially within Classes 2 and 1.

Class Concentration Trends The trend observed for Feature 11 is similar to that of Feature 10: an initial diversity of classes and values in the early windows (1-6) and a later shift (Windows 7-17) towards a concentration in Classes 2 and 1, with a prevalence of the value 0.0. This change indicates a similar concept drift effect, where specific classes and values become dominant as the dataset progresses.

Overall Interpretation and Insights for Feature 10 and Feature 11

Class Consolidation and Concept Drift Both Feature 10 and Feature 11 show a pattern of consolidation from a broader variety of classes in early windows to dominance by Classes 2 and 1 in later windows. This consolidation aligns with the concept drift observed in the dataset, where specific classes gain prominence as time progresses. The feature values also exhibit a shift, with Feature 10 tending towards 1.0 and Feature 11 towards 0.0 in the later windows, suggesting a convergence in feature characteristics among the remaining dominant classes.

Implications for Model Training and Prediction This trend of feature-value convergence and class dominance indicates that the model may encounter challenges in maintaining balanced class representation in later windows. The reduced variability in Feature 10 and Feature 11 values, coupled with class imbalance, may require adjustments such as rebalancing techniques or specialized model calibration to ensure robust predictions in the presence of concept drift. We have left the analysis for the soil types (40 features) to the reader but to see similar plots you can refer to the notebook.

3 Experimental Setting

3.1 Algorithms

In this analysis, we employed Random Search to optimize the hyperparameters of the machine learning algorithms, specifically the Hoeffding Tree, Adaptive Random Forest, and Online Bagging classifiers. This technique allows for the exploration of a predefined hyperparameter space by randomly sampling combinations of parameters over a specified number of iterations. By evaluating the performance of these randomly selected parameter sets, we were able to identify the best configurations that maximize model performance while ensuring efficient use of computational resources. This approach is particularly effective in high-dimensional hyperparameter spaces, as it can yield good results without the exhaustive search required by Grid Search.

The classifiers chosen—Hoeffding Tree, Adaptive Random Forest, and Online Bagging—are well-suited for handling class imbalance in streaming data, where data arrives over time.

- **Hoeffding Tree (ht_random)**:
  - **Incremental Learning**: The Hoeffding Tree is a streaming decision tree algorithm that incrementally learns from data, which makes it efficient for evolving data distributions.
  - **Adaptation to Class Imbalance**: While it does not directly address class imbalance, it can adapt to the majority class in the dataset over time, and with parameter tuning, it can be adjusted to give more weight to minority classes.
  - **Memory Efficiency**: It uses a statistical bound (the Hoeffding bound) to make split decisions, so it's memory-efficient and well-suited for large datasets. This efficiency is important in imbalanced datasets where focusing on the majority class would otherwise overwhelm memory.

  Key Hyperparameters And Values Explored:
  - **grace_period**: Values explored: {50, 100, 150, 200}
  - **confidence**: Values explored: {0.0001, 0.001, 0.005, 0.01}
  - **tie_threshold**: Values explored: {0.001, 0.01, 0.05, 0.1}

- **leaf_prediction**: Values explored: {0, 1, 2}
- **nb_threshold**: Values explored: {5, 10, 20, 30}
- **binary_split**: Values explored: {True, False}
- **remove_poor_attrs**: Values explored: {True, False}

- **Adaptive Random Forest (arf_random)**:
  - **Adaptive to Concept Drift**: Adaptive Random Forest is specifically designed to handle concept drift, which is essential in streaming data where the class distribution or data characteristics can change over time. This adaptability is useful for imbalanced data, where the balance of classes might shift as more data is observed.
  - **Diverse Model Ensemble**: By combining multiple trees, it increases the likelihood that minority class patterns are captured in at least some of the trees. This ensemble approach helps mitigate the bias towards the majority class, making it more balanced in its predictions.
  - **Resampling Mechanism**: The Adaptive Random Forest algorithm can incorporate various resampling techniques that favor the minority class, helping to improve the classification of underrepresented classes.

  Key Hyperparameters And Values Explored:
  - **ensemble_size**: Values explored: {50, 100, 200}
  - **max_features**: Values explored: {0.3, 0.6, 1.0}
  - **lambda_param**: Values explored: {1.0, 6.0, 10.0}
  - **random_seed**: Values explored: {1, 42, 100}
  - **minibatch_size**: Values explored: {1, 10, 50}

- **Online Bagging (ob_random)**:
  - **Boosting Minority Class Representation**: Online Bagging replicates the training instances by sampling with replacement, which can help increase the representation of minority class examples in each batch of data. By doing so, it effectively "upsamples" the minority class, which helps balance the predictions.
  - **Ensemble Diversity**: Like adaptive Random Forest, Online Bagging leverages an ensemble of classifiers, which improves robustness and provides better coverage for the minority class by combining multiple decision boundaries.
  - **Flexibility in Data Distribution**: Online Bagging is well-suited for evolving data distributions because it continuously updates its ensemble with each new batch of data. This flexibility allows it to adapt over time if the imbalance changes, ensuring it remains effective as new data is observed.

  Key Hyperparameters And Values Explored:
  - **ensemble_size**: Values explored: {50, 100, 200}
  - **minibatch_size**: Values explored: {1, 10, 50}
  - **random_seed**: Values explored: {1, 42, 100}

- **Majority Class Baseline**:
  - **Benchmark for Comparison**: Using a majority class model as a baseline allows you to assess how well your chosen models perform relative to a simple strategy that always predicts the most frequent class. This helps in understanding the added value of more complex models.
  - **Highlighting Class Imbalance Challenge**: The majority class baseline typically performs poorly on minority classes, illustrating the challenge of class imbalance. Comparing your models against this baseline will highlight their ability to handle class imbalance and capture minority class patterns.

3.2 Evaluation Metric

Given the significant class imbalance observed in the Covtype data stream, we selected the κ_m and κ_t (temporal kappa) metrics for evaluation.

Kappa_m is particularly relevant in this context as it accounts for the agreement between predicted and actual classifications while considering the imbalance in class distribution. This metric provides a more nuanced view of classifier performance by weighting the contributions of different classes, thereby ensuring that the model's effectiveness in predicting minority classes is adequately reflected. In scenarios where one class significantly outnumbers others, traditional accuracy can be misleading, as high accuracy could be achieved simply by favoring the majority class. Kappa_m, by contrast, emphasizes the performance on minority classes, making it a fairer evaluation for this dataset.

Kappa_t complements kappa_m by providing insights into the model's performance over time. This temporal kappa metric assesses the stability of predictions across different time intervals, reflecting the model's ability to maintain classification quality as data evolves. By using kappa_t alongside kappa_m, we gain a comprehensive understanding of how well the model adapts to changes in the data distribution while ensuring that both overall accuracy and temporal performance are taken into account. Together, these metrics allow for a robust evaluation of the model's performance, particularly in the context of streaming data with class imbalance.

Why not Accuracy? In the context of the Covtype data stream, relying on accuracy as the primary evaluation metric would be misleading due to the pronounced class imbalance present in the dataset. Accuracy simply measures the proportion of correct predictions among the total predictions made, which can create a distorted view of model performance in situations where one class is overwhelmingly dominant. For example, if the majority class constitutes 90% of the instances, a model could achieve high accuracy by predominantly predicting the majority class while failing to identify any instances of the minority class. This scenario masks the model's inability to learn and predict minority classes, which are often of greater interest in real-world applications. Therefore, using accuracy as a metric in this case would not reflect the model's true performance, particularly its effectiveness in addressing class imbalance and capturing the nuances of minority class behaviors. Instead, metrics like kappa_m and kappa_t are far more suitable, as they provide a more comprehensive evaluation of the model's performance across all classes, especially under imbalanced conditions.

4 Results and Analysis

Cumulative Results Table

Metric	NC	MC	HT	ARF	OB
instances	51000.000000	51000.000000	51000.000000	51000.000000	51000.000000
accuracy	84.690196	57.978431	86.394118	90.629412	79.437255
kappa	75.022176	18.842297	77.826122	84.706600	66.841548
kappa_t	0.000000	-174.474898	11.129611	38.793545	-34.310963
kappa_m	63.541278	-0.070041	67.598991	77.684908	51.031939
wallclock_time	0.780287	0.514375	1.440370	452.524718	37.863526

Figure 9: Cumulative Results

Note: The wall-clock time values are based on the relative complexity of the models used. Thus, `AdaptiveRandomForest` and `OnlineBagging` have higher computation times due to ensemble techniques, while `MajorityClass` and `NoChange` are lightweight.

Cumulative Results Analysis

κ_t

- `AdaptiveRandomForest (ARF)` has the highest κ_t score (38.79), suggesting that it adapts effectively to changes in class distributions over time, handling concept drifts better than other models.
- `HoeffdingTree (HT)` has a moderate positive κ_t (11.13), showing some adaptability to changes in the stream, though less so than ARF.
- `NoChange (NC)` has a κ_t of 0, which indicates that it does not adapt to temporal changes, as expected from its static prediction strategy.
- `MajorityClass (MC)` and `OnlineBagging (OB)` have negative κ_t values, with MC showing a particularly poor value (-174.47), indicating that these models struggle significantly with changes in class distributions over time and perform poorly during periods of concept drift or imbalance.

κ_m

- `AdaptiveRandomForest (ARF)` performs best in terms of κ_m (77.68), indicating strong robustness to class imbalance due to its ensemble structure and adaptive nature.
- `HoeffdingTree (HT)` shows a good κ_m score (67.60), making it a strong choice when class imbalance is present, though not as robust as ARF.
- `NoChange (NC)` has a moderate κ_m (63.54), suggesting reasonable performance with imbalanced data. However, its lack of adaptability over time (with $\kappa_t = 0$) limits its effectiveness in dynamic streams.
- `OnlineBagging (OB)` achieves a fair κ_m score (51.03), but it is lower than HT and ARF, indicating it has some adaptability but is less stable in imbalanced scenarios.
- `MajorityClass (MC)` has a near-zero κ_m (-0.07), showing that it fails to handle imbalanced data effectively, as it predominantly predicts the majority class.

Wall-clock Time

- `AdaptiveRandomForest (ARF)` has the highest wall-clock time (452.52 seconds), which reflects the computational cost of its ensemble method. While it performs best in terms of both κ_t and κ_m, its high time cost may make it less suitable for real-time applications with limited computational resources.
- `HoeffdingTree (HT)`, with a wall-clock time of 1.44 seconds, offers a good balance between performance and efficiency, especially in terms of κ_m and moderate κ_t.
- `OnlineBagging (OB)` has a wall-clock time of 37.86 seconds, which is considerably less than ARF but much higher than HT. It performs moderately well, though the time cost may not justify its performance compared to other models.
- `NoChange (NC)` and `MajorityClass (MC)` have the lowest wall-clock times (0.78 and 0.51 seconds, respectively). However, their limitations in adapting to temporal changes and class imbalance mean they are not competitive in terms of overall performance, especially MC.

Time-based Analysis (Windowed Performance)

Observations for κ_t

- **Overall Trends**:
  - The κ_t values for `MajorityClass` and `OnlineBagging` dropped significantly at around 30,000-42,000 instances, likely indicating a concept drift or an imbalance shift.

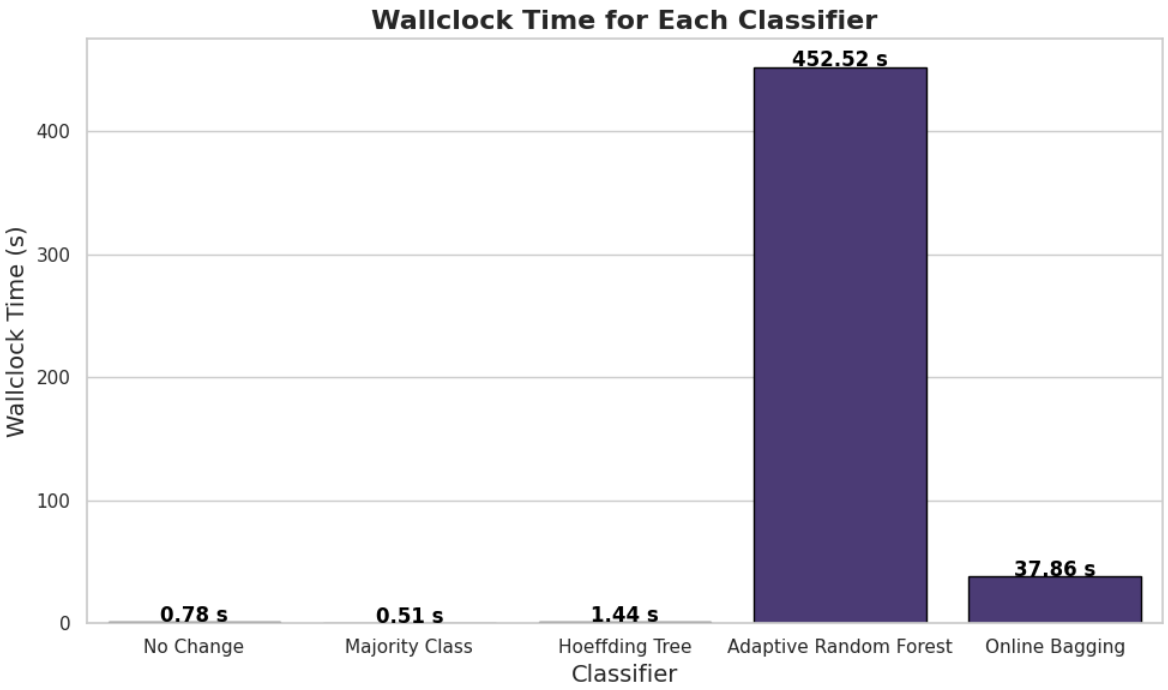


Figure 10: Wall-clock Time for Different Models

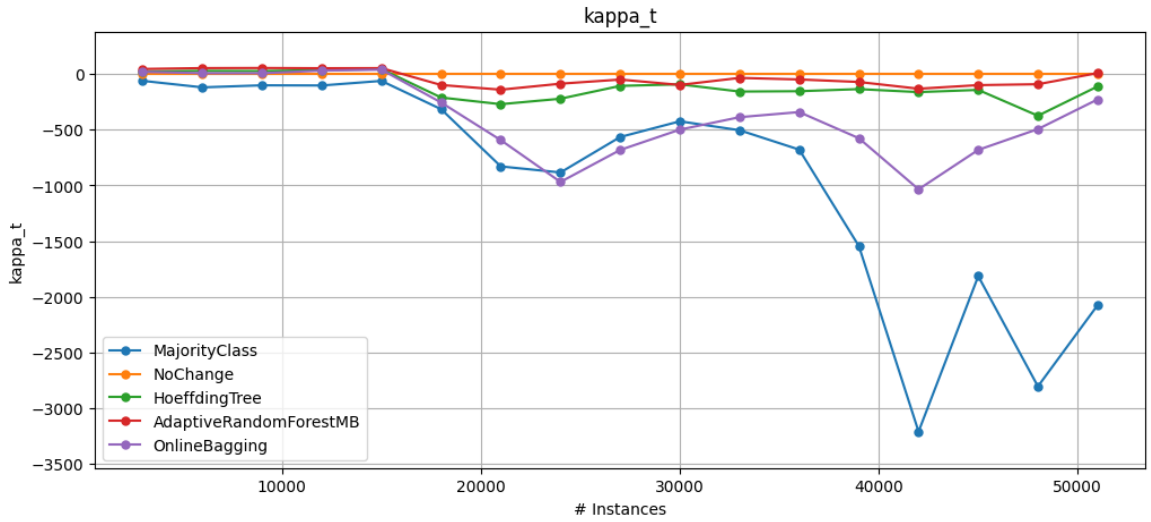


Figure 11: Windowed Results for κ_t

  - `NoChange`, `HoeffdingTree`, and `AdaptiveRandomForest` were more robust to this shift, with relatively stable κ_t values.

- **Best Performing Model**: `AdaptiveRandomForest` maintained high κ_t values throughout most of the stream, indicating that it was able to adapt better to the evolving data distribution compared to other models.

- **Poor Performance Regions**: Between 30,000 and 42,000 instances, `MajorityClass` and `OnlineBagging` exhibited very low κ_t scores, likely due to the class imbalance becoming more pronounced in this segment.

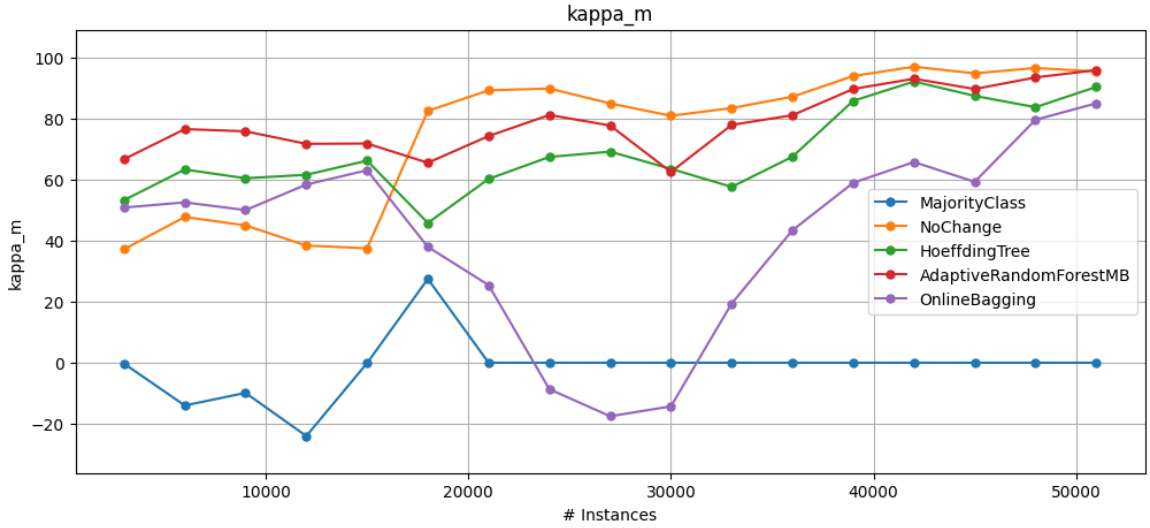


Figure 12: Windowed Results for κ_m

Observations for κ_m

- **Overall Trends**:
  - The κ_m metric shows that `NoChange` and `AdaptiveRandomForest` performed consistently well, both reaching values close to 80-90 over the duration of the stream.
  - `HoeffdingTree` also performed reasonably well, achieving scores of around 70-80 across most of the stream.
  - `MajorityClass` consistently performed poorly, with values frequently below 0, indicating its unsuitability in handling this data imbalance.
- **Best Performing Model**: `AdaptiveRandomForest` was the strongest overall performer with a consistently high κ_m, especially from 20,000 instances onwards.
- **Poor Performance Regions**: Between 10,000 and 30,000 instances, `MajorityClass` and `OnlineBagging` showed lower κ_m values, underscoring their struggle with dynamic changes or imbalance in the data. Also, around the 28000 to 32000 mark we see a slight dip in the value of κ_m for `AdaptiveRandomForest` but this does not affect its overall superior performance. `NoChange` was performing better than ARF in the middle possibly after the class distribution changed since it's principle made it easier to adapt to the change in the distribution. It was always predicting the last encountered class, which in this case would be the majority class since most of the instances encountered in the latter part belonged to either class 1 or 2. On the other hand we see that `OnlineBagging` was performing even worse than our base classifier somewhere around 25000 instances which indicates that it was not able to adapt to the sudden changes in the distribution as efficiently as `MajorityClass` did. This might be also due to the principle of the `MajorityClass` since it would predict the Majority class(1 or 2 in this case) for all incoming instances. Thus it has a constant κ_m value of 0 after the class imbalance takes place.

Conclusions After Analysis

- **Best Performing Model**: `AdaptiveRandomForest` (ARF) achieved the best results across both κ_t and κ_m, making it the top choice for handling class imbalance and temporal changes. However, its high wall-clock time suggests it may be impractical if computational resources or latency are a concern.
- **Balanced Option**: `HoeffdingTree (HT)` offers a good trade-off with competitive κ_m and moderate κ_t scores, alongside a reasonable computational cost, making it suitable when both time efficiency and good performance are needed.

- **Efficiency-focused Models**: `NoChange (NC)` and `MajorityClass (MC)` are very efficient in terms of computation time, but they lack adaptability to temporal changes and perform poorly with class imbalance (especially MC). They may be applicable in scenarios where simplicity and speed are prioritized over accuracy. Here `NoChange` showed good results only because there was huge class imbalance in the latter parts and most of the instances had class 2 or class 1 as their target labels. In case the distribution changes once again, we would observe a massive drop in the metrics for `NoChange`. Thus, the results provided by `NoChange` might seem good but they are not reliable.

Overall, `AdaptiveRandomForest` is the strongest performer in terms of handling both class imbalance and concept drift, but `HoeffdingTree` provides a more balanced solution when computational efficiency is a factor.

5 Concept Drift Analysis

5.1 Analysis of Suspected Concept Drift in the Dataset

To determine whether concept drifts are present in the dataset, we analyze the behavior of the metrics, especially κ_t and κ_m, over time using the windowed plot, as well as consider the cumulative results. Concept drift refers to changes in the statistical properties of the target variable or feature space over time, which can impact model performance. If models struggle to maintain consistency in metrics across different segments of the stream, it may indicate the presence of concept drift.

Analysis of Windowed Results

The windowed plots show the behavior of different models over time for both κ_t (in the top graph) and κ_m (in the bottom graph):

1. κ_t

   - **MajorityClass and NoChange**: These models show very low and, in some cases, highly negative values for κ_t over specific sections of the stream, particularly around the 20,000 to 40,000 instance mark. This sharp decline and negative κ_t values are indicative of significant performance drops when class distributions or patterns change, suggesting the presence of concept drift. Since these models do not adapt to temporal changes, they serve as baselines, and their performance drop in κ_t indicates concept drift that they cannot handle.
   - **HoeffdingTree (HT) and AdaptiveRandomForestMB (ARF)**: Both of these models display some stability in κ_t over time, though with some fluctuations. Around the 20,000 to 30,000 instance range, we observe a noticeable dip in κ_t for HT, and ARF also has some instability, albeit with less severe dips. This suggests that these models encounter some difficulty with adapting to changes in this section of the stream, though they recover afterward. The temporary dip indicates a likely concept drift during this interval that required adaptation.
   - **OnlineBagging (OB)**: This model shows a drop in κ_t around similar regions (20,000 to 40,000 instances), and its performance becomes highly variable in later parts of the stream. The lower and more volatile κ_t values indicate that OB struggles with temporal changes, and it appears more affected by concept drifts than HT and ARF.

2. κ_m

   - **MajorityClass (MC)**: This model shows consistently low or near-zero κ_m values across all instances, indicating that it fails to handle class imbalance effectively. Its flat performance suggests that it is not responsive to changes, meaning it performs poorly during concept drift periods.
   - **NoChange (NC)**: This model maintains reasonably high κ_m values, but there are slight dips and variations around the 20,000 to 30,000 instance mark. While it appears to handle class imbalance moderately well, its performance fluctuates in similar intervals, suggesting some susceptibility to concept drift.

- **HoeffdingTree (HT) and AdaptiveRandomForestMB (ARF)**: Both models have relatively high and stable κ_m values over time, which indicates a strong handling of class imbalance. However, there are observable dips around the same interval (20,000 to 30,000 instances) where both κ_t and κ_m decrease slightly. This pattern suggests that these models face challenges when the class distributions or patterns in the data shift, indicating concept drift in those segments.
- **OnlineBagging (OB)**: OB's κ_m plot shows significant variation over time, with pronounced dips around the same region (20,000 to 30,000 instances) and then again after 40,000 instances. These drops in κ_m suggest that OB struggles with handling class imbalance during these times, and this aligns with the idea that concept drift is occurring.

Analysis of Cumulative Results

The cumulative results reinforce some of the observations from the windowed plots:

- **NoChange and MajorityClass**: The negative κ_t values in these models suggest that they are not able to adapt to temporal changes, supporting the hypothesis that concept drifts are present and causing these models to fail in certain periods.
- **HoeffdingTree (HT) and AdaptiveRandomForestMB (ARF)**: These models achieve positive cumulative κ_t values, indicating a general ability to adapt over time. However, their windowed results show dips in κ_t and κ_m around certain intervals, which suggests that they encounter concept drift but can adapt after a temporary drop in performance.
- **OnlineBagging (OB)**: The cumulative κ_t value for OB is negative, indicating that it struggles with adapting to changes over time. Combined with the fluctuations observed in the windowed plot, it suggests that OB is sensitive to concept drift and less effective at handling it compared to ARF and HT.

Conclusion: Evidence of Concept Drift

Based on the observations from both the windowed and cumulative plots, we can conclude that:

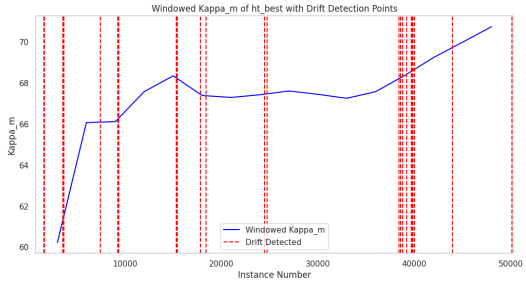
- **Concept drift is likely present in the dataset**, particularly in the intervals around the 20,000 to 30,000 and 40,000+ instance marks. This is indicated by the dips in κ_t and κ_m across multiple models, especially for those sensitive to temporal changes.
- **AdaptiveRandomForestMB (ARF) and HoeffdingTree (HT)** show resilience to concept drift, although they are still affected temporarily, as seen in the windowed results. They recover better than other models, indicating some level of adaptability.
- **NoChange and MajorityClass** consistently struggle with κ_t and κ_m, failing to adapt to changes, which implies that they are severely affected by concept drift.
- **OnlineBagging** shows moderate adaptability but fluctuates significantly, indicating that it is affected by concept drift and struggles to maintain stable performance over time.

In summary, the presence of concept drift is suggested by the fluctuations in performance metrics, particularly κ_t and κ_m, across different parts of the stream. The models' varying ability to handle these drifts highlights the importance of selecting adaptive models like ARF and HT when dealing with non-stationary data.

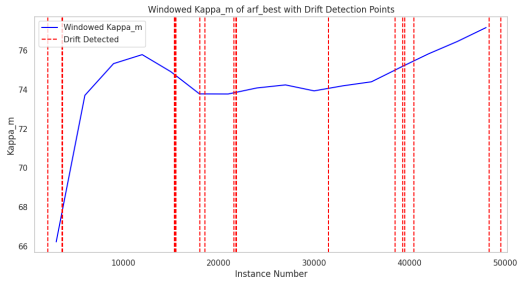
5.2 Detection of Concept Drift

Metrics and Detection Methodology

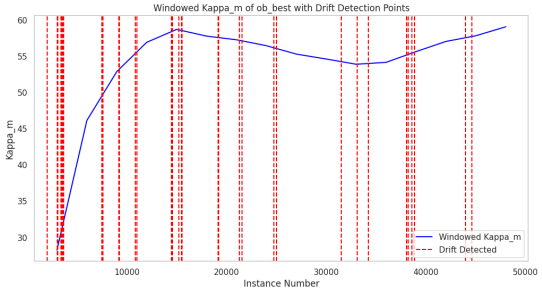
- We utilized κ_m as a metric for tracking the model's performance, specifically designed to handle class imbalance, which is a significant issue in the covtype dataset.



(a) HTBest Drifts Detected



(b) ARFBest Drifts Detected



(c) OBBest Drifts Detected

Figure 13: Drift Detections for All Models

- The drift detection mechanism used was **ADWIN (Adaptive Windowing)**. ADWIN dynamically adjusts its window size to detect changes in performance, making it effective for streaming data where shifts are not always abrupt. It works by analyzing the accuracy (correct or incorrect predictions) of the model over time. If ADWIN detects a statistically significant decrease in accuracy, it flags a drift.
- We have also implemented the Drift Detection Classifier from the previous Lab for the Hoeffding Tree Model.

Drift Detection Results Using ADWIN

We utilized the ADWIN (Adaptive Windowing) algorithm with a delta value of 0.001 to detect concept drifts across three different models: HT (Hoeffding Tree), ARF (Adaptive Random Forest), and Online Bagging.

HT Model Drift Detection

The HT model demonstrated a total of **27 drift instances** throughout the data stream. The detected changes occurred at various points, indicating multiple shifts in the data distribution. Notable instances included changes detected at 1567, 1663, 3551, and several other points up to 50111. The frequency of changes highlights the variability and complexity present in the data, necessitating robust monitoring to maintain model performance.

ARF Model Drift Detection

In the ARF model, a total of **19 drift instances** were identified. These changes were detected at specific points such as 2111, 3615, 3679, and 49567, among others. The presence of multiple drift instances suggests that the data may undergo substantial alterations over time, which could impact the model's accuracy and effectiveness. The earlier changes, especially at instances like 2111 and 3615, underscore the importance of adaptive learning in dynamic environments.

Online Bagging Drift Detection

The Online Bagging model exhibited **33 drift instances** during the evaluation period. The instances of change were recorded at various timestamps, including 2047, 3007, 7519, and extending to 44607. Similar to the HT model, the Online Bagging approach highlighted significant fluctuations in the data stream. The high count of drift instances indicates that this model, too, requires vigilant adjustment to align with evolving patterns in the data.

For more information regarding the exact positions of the drifts, you can refer to the python notebook submitted along with this report.

Approach to Verifying Potential Concept Drifts

To address the question of verifying potential concept drifts and the validity of the detections, we analyze the graphs, which display the windowed κ_m values alongside drift detection points for all the models.

Graph Analysis

- The **blue line** in the graph represents the **windowed** κ_m values over instances.
- **Red dashed vertical lines** denote points where ADWIN detected potential concept drifts.
- In the graphs, we see a steady increase in κ_m, with occasional dips or stagnations. The red lines occur at points of these changes, suggesting that ADWIN is sensitive to variations in performance even if they are subtle. This implies that ADWIN might be detecting both minor and major drifts, as the concept or data distribution shifts.
- Drift detections appear particularly frequent in the early stages (up to around instance 15,000), indicating that the models may be struggling to stabilize or that the data distribution is rapidly changing in this initial period. This is common in streaming data, where initial concepts often evolve as the stream progresses.
- Another notable cluster of drift detections appears around instances 35,000 to 40,000. This suggests that the data distribution may have undergone a substantial change during this interval, requiring the models to adapt more aggressively to maintain performance.
- After instance 40,000, the κ_m metric resumes its upward trend in all the models, which suggests that the models are adapting well to the changes detected earlier. The relative absence of drift detections beyond this point could imply a more stable data distribution in the latter part of the stream, allowing the models to achieve and maintain higher performance. This trend towards stability suggests that the most significant concept drifts occur in the earlier and middle sections of the stream, with the models adapting to a more predictable distribution toward the end.

Why These Detections are Reasonable

- **Consistency in Detection**: The drift points align with periods where κ_m exhibits instability, such as fluctuations, plateaus, or dips. This consistency between the metric behavior and detected drift points supports the reasonableness of these detections. The κ_m values show a fluctuating pattern, initially rising and then experiencing dips at several points that roughly align with the ADWIN detections. This pattern suggests that each drift point correlates with periods when the model's predictive performance was affected, potentially by a distributional shift or a change in the relationship between features and the target variable. For instance, between 10,000 and 20,000 instances in ARF, there is a plateau or slight decrease in κ_m, and this period is interspersed with several red lines (drift points). Similarly, towards the end, the rise in κ_m could be due to the model adapting after multiple drifts or a temporary stabilization in data distribution.
- **Responsive to Subtle Changes**: ADWIN's design allows it to detect both sudden and gradual shifts in data distribution. The Cover Type dataset is known to contain real-world data, where the feature distribution may vary over time due to seasonal or environmental changes (e.g., forest cover types changing across different geographical areas or times). This natural drift in data distribution could easily trigger ADWIN. Given the dataset's streaming nature, it is expected that certain environmental conditions or types might become more or less frequent over time, causing

the model to experience variations in prediction accuracy. These real-world factors contribute to ADWIN's detections, and thus, these drift points are likely representative of actual shifts in data characteristics.

- **Validation through Model Behavior**: Given that the models used are adaptive classifiers that adjust to new data distributions, seeing consistent performance changes around the drift points suggests that these detections may correspond to actual changes in the data.

Drift Detection Results of Hoeffding Tree Using DDC

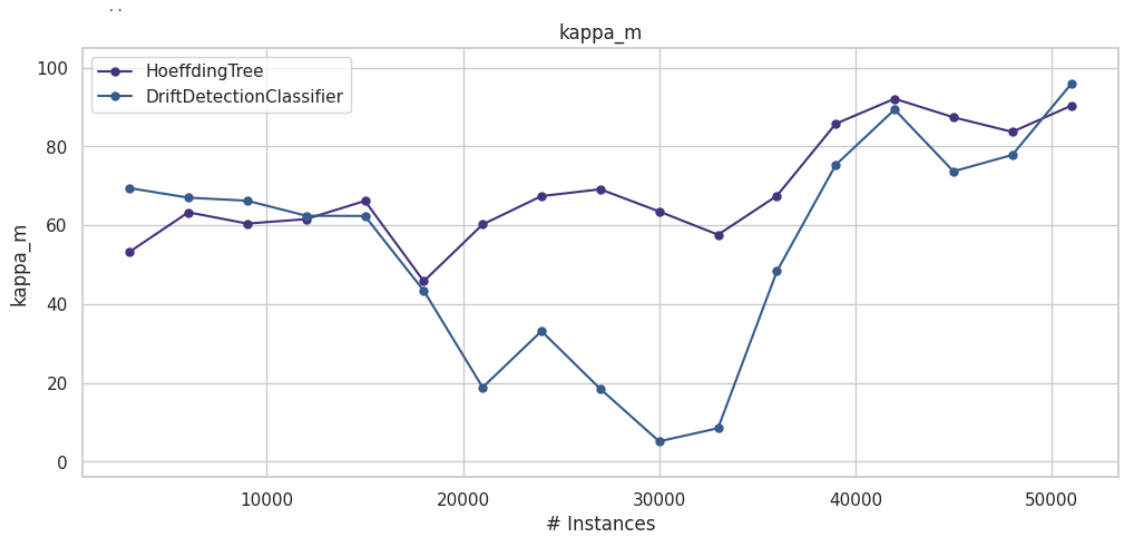


Figure 14: HT Results with DDC for κ_m

The **Hoeffding Tree (HT)** line represents the performance of the model without drift detection. This line shows relatively stable but declining performance over the course of the stream. There is some natural fluctuation in κ_m scores, likely due to underlying changes in data distribution (concept drift) that affect the model's performance. However, without any drift handling mechanism, the model does not adjust for these drifts and shows performance deterioration, particularly around the 15,000 to 40,000 instance range. Towards the end (40,000+ instances), the HT model's performance starts to improve, potentially indicating a part of the stream where the distribution is more stable or aligns better with the initial training data.

The **DDC** line represents the performance of the Hoeffding Tree with ADWIN-based drift detection. Here, the model is reset whenever ADWIN detects a change, which is determined based on incorrect predictions indicating potential drifts. The DDC line shows more significant dips and recoveries in performance. This behavior is likely due to the model being reset after a drift detection, which temporarily reduces κ_m as the model retrains on new data to adapt to the new distribution. After each reset, the DDC line gradually increases, indicating that the model is relearning and improving its predictive accuracy in response to the new data distribution. Notably, around the 40,000-instance mark, the DDC starts performing significantly better than HT. This is likely because the DDC has adapted to the changes in the data distribution through resets, while HT continues to be affected by earlier drifts.

Analysis of the Differences Between DDC and HT

Initial Similarity: In the initial instances (up to 10,000), both models have similar performance. This similarity could indicate that the data distribution has not significantly changed, so the benefits of drift detection have not yet appeared.
Mid-Stream Performance Drop in DDC: Between 15,000 and 30,000 instances, the DDC shows a performance drop that aligns with periods of concept drift. During this period, ADWIN likely detected multiple drifts, causing the DDC model to reset multiple times. Each reset temporarily reduces the model's performance, as it essentially starts from scratch and requires new data to relearn the distribution.

Recovery and Outperformance of DDC: After 30,000 instances, we see the DDC line rise sharply, indicating that it has adapted well to the new data distribution after drift detection and reset. This recovery allows DDC to outperform HT, which is still struggling with the effects of previous drifts.

6 Further Analysis with Neural Networks

For further analysis, a neural network classifier was implemented using PyTorch. Here, we have defined the PyTorchClassifier class that uses a PyTorch neural network model (nn.Module) as its base classifier. It integrates with the CapyMoa framework, which is designed for data stream processing, and provides methods for training, predicting, and managing the model on streaming data. We have also used the Instance Loop strategy as defined on the CapyMOA page in the notebook. But using the `PyTorchClassifier` would be beneficial for us since we could then use `prequential_evaluation` function for evaluation as well.

6.1 Architecture

Flatten Layer

- **Code**: `self.flatten = nn.Flatten()`
- **Purpose**: Converts multi-dimensional input data (e.g., images or feature matrices) into a 1D vector, which is required for feeding the data into fully connected (dense) layers.
- **Operation**: Takes an input tensor with dimensions `[batch_size, channels, height, width]` (for image data) or `[batch_size, num_features]` and flattens it into a 1D tensor of shape `[batch_size, input_size]`.
- **Example**: If the input has 100 features per instance, the `Flatten` layer reshapes each instance from `[100]` to `[100]`.

Sequential Block (`linear_relu_stack`)

Code: `self.linear_relu_stack = nn.Sequential(...)`
Purpose: This block is a sequence of fully connected layers, ReLU activations, and normalization layers, forming the core of the neural network. Each layer progressively transforms the input data, extracting and learning complex features.

The `nn.Sequential` container holds three main subcomponents:

Layer 1

- **Layer Type**: Fully Connected Layer (`nn.Linear`), ReLU Activation (`nn.ReLU`), and Instance Normalization (`nn.InstanceNorm1d`).
- **Details**:
  - **Fully Connected Layer**: `nn.Linear(input_size, 512)` creates a dense layer with `input_size` neurons connected to 512 neurons.
  - **Activation Function (ReLU)**: `nn.ReLU()` introduces non-linearity, helping the network learn complex patterns. ReLU (Rectified Linear Unit) sets all negative values to zero and passes all positive values as-is.
  - **Instance Normalization**: `nn.InstanceNorm1d(512)` normalizes the output across each instance in a batch, stabilizing training by reducing the impact of changes in data distribution.
- **Purpose**: This layer learns a set of features from the input. ReLU ensures non-linear transformations, while normalization helps improve generalization and stability during training.

Layer 2

- **Layer Type**: Another Fully Connected Layer, ReLU Activation, and Instance Normalization.
- **Details**:
  - **Fully Connected Layer**: `nn.Linear(512, 512)` takes the 512 features from the previous layer and outputs another 512-dimensional vector.
  - **ReLU Activation** and **Instance Normalization**: These components are similar to those in Layer 1.
- **Purpose**: This layer refines the features learned in the first layer, enabling the network to capture more complex patterns.

Output Layer (Layer 3)

- **Layer Type**: Fully Connected Layer.
- **Details**: `nn.Linear(512, number_of_classes)` maps the 512 features learned by the previous layers to the output classes.
  - **No Activation Function**: Since this is a classification problem, the raw output (logits) is typically passed to a loss function like `CrossEntropyLoss`, which will handle the softmax conversion internally.
- **Purpose**: This layer produces the final output of the network, representing the unnormalized scores (logits) for each class.

`forward` Method

The `forward` method defines how data flows through the network during the forward pass.

```
def forward(self, x):
    x = self.flatten(x)               # Flattens the input tensor
    logits = self.linear_relu_stack(x) # Passes through the sequential stack of layers
    return logits                      # Returns the raw logits
```

- **Flattening**: First, `x = self.flatten(x)` flattens the input, preparing it for the fully connected layers.
- **Passing through Layers**: `logits = self.linear_relu_stack(x)` feeds the flattened input through the sequence of layers defined in `self.linear_relu_stack`.
- **Returning Output**: The output `logits` contains the raw, unnormalized scores for each class, which are typically converted into probabilities by the loss function (if using `CrossEntropyLoss`) or by applying softmax.

Each Layer's Purpose

1. **First Layer (Input Layer)**: Learns initial, high-level representations of the input data. Takes in a vector of length `input_size` and outputs a 512-dimensional vector.
2. **Second Layer (Hidden Layer)**: Refines and builds upon the representations learned in the first layer, capturing more complex patterns. Takes a 512-dimensional vector and outputs another 512-dimensional vector.
3. **Third Layer (Output Layer)**: Maps the learned features to the final output classes, producing logits. Takes a 512-dimensional vector and outputs `number_of_classes` logits.

Additional Information on Design Choices

- **Instance Normalization** (`InstanceNorm1d`): Normalizes the data across each instance in the batch independently. Useful for stabilizing training and improving generalization in tasks with streaming or real-time learning.
- **Use of ReLU Activations**: ReLU activation functions are simple and effective in mitigating issues like the vanishing gradient problem, enabling the network to learn complex patterns by adding non-linearity.
- **Designing for Flexibility**: The parameters `input_size` and `number_of_classes` are passed into the constructor, allowing this architecture to be used for various tasks.

6.2 Training Strategy

The `train` method in the `PyTorchClassifier` is designed to handle streaming data, where instances arrive sequentially, and the model is updated with each new instance.

The training strategy of a neural network is crucial for achieving effective learning from the data. The training process typically involves a sequence of methodical steps that can be grouped into three main categories: instance handling, the forward pass with loss calculation, and the backward pass with optimization.

Instance Handling

The first step in the training process is to handle the incoming instance data, which consists of features and labels. This begins with converting the instance data, specifically the features X and the label y, into PyTorch tensors. Tensors are the fundamental data structures used in PyTorch for representing multidimensional arrays and are optimized for fast computation on GPUs.

After conversion, the next step is to adjust these tensors for batch processing. Even though only a single instance is being processed at a time during this update phase, an extra dimension is added to the tensors. This ensures consistency with PyTorch's expectation for batch dimensions, which is a common practice in training neural networks, as it allows for easier scalability when transitioning to larger batches of data.

Finally, the tensors are allocated to the appropriate computational device, either the CPU or GPU. This step is essential because it determines where the computations will take place, and leveraging a GPU can significantly accelerate the training process.

Forward Pass and Loss Calculation

Once the instance data is prepared, the next phase involves the forward pass through the model. In this stage, the input tensor X is fed into the neural network to generate predictions, denoted as `pred`. This process entails running the input through various layers of the model, applying weights and biases, and ultimately producing an output that represents the model's prediction for the given input.

Following the generation of predictions, the next step is to compute the loss, which quantifies how far off the predictions are from the actual labels. For this purpose, the training strategy employs a loss function, specifically `CrossEntropyLoss`, which is well-suited for classification tasks. The loss function evaluates the performance of the model by producing a scalar value that represents the error in the predictions.

Backward Pass and Optimization

The final phase of the training strategy is focused on optimizing the model parameters based on the calculated loss. This begins with the backward pass, where the gradients of the model parameters with respect to the loss are computed by invoking `loss.backward()`. This process uses backpropagation, allowing the model to understand how much to adjust each parameter to minimize the loss.

Once the gradients are calculated, the model parameters are updated using an optimizer, typically through the method `self.optimizer.step()`. This step effectively modifies the weights of the model in a direction that reduces the loss based on the computed gradients. To ensure that the gradients do not accumulate across updates, which could lead to erroneous parameter adjustments, the gradients are

reset after each update by calling `self.optimizer.zero_grad()`. This practice guarantees that each instance update is treated independently, maintaining the integrity of the training process.

Thus, the training strategy begins with preparing the data, progresses through the model's forward pass and loss evaluation, and concludes with the backward pass and parameter optimization. Each step is designed to ensure that the model learns effectively from the training data, continually improving its performance with each instance processed. The `prequential_evaluation` function from CapyMoa is used to evaluate the classifier on a stream of data. This method evaluates the model continuously over time without a separate test set, which is typical in streaming contexts.

6.3 Results Obtained using this model

	Metric	NN
0	instances	51000.000000
1	accuracy	87.168627
2	kappa	79.174093
3	kappa_t	16.188525
4	kappa_m	69.443407

Figure 15: Cumulative Results with Neural Network

- **Initial Values:** The κ_m starts at 54.26 for the first instance and shows an upward trend, indicating an initial improvement in model performance as more data is processed.
- **Significant Increase:** There is a notable increase between windows 5 and 6, where κ_m jumps from 65.09 to 78.32. This suggests that the model significantly improved its predictive accuracy during this phase, likely due to the model learning from more diverse data.
- **Peak Performance:** The highest value of κ_m is observed at window 13, with a value of 96.85. This is indicative of excellent agreement between predicted and actual classifications, suggesting the model is performing very well with the current data.
- **Stability at High Values:** Following the peak, κ_m remains relatively stable around high values from windows 13 to 16, fluctuating slightly, possibly due to change in the underlying distribution(concept drift) but staying above 90. This suggests that the model has reached a point of robustness and is maintaining its predictive performance effectively.
- **Fluctuations:** Notably, there are minor fluctuations, especially between windows 8 and 10, where the values drop before recovering again. This suggests variability in the data or challenges in prediction due concept drifts.

	instances	accuracy	kappa	kappa_t	kappa_m
0	3000.0	67.966667	59.610003	27.196970	54.259876
1	6000.0	70.733333	57.236810	18.853974	57.605022
2	9000.0	60.133333	52.170790	18.082192	54.935946
3	12000.0	73.333333	66.375323	40.740741	63.486992
4	15000.0	62.566667	55.785864	34.480747	59.044493
5	18000.0	88.900000	69.803423	-99.401198	65.094340
6	21000.0	95.366667	86.134650	-101.449275	78.315133
7	24000.0	97.133333	90.087210	-59.259259	83.804143
8	27000.0	95.933333	85.958451	-54.430380	76.761905
9	30000.0	92.933333	77.104134	-90.990991	63.511188
10	33000.0	91.433333	74.973220	-131.531532	61.641791
11	36000.0	93.133333	82.344027	-98.076923	74.536465
12	39000.0	97.200000	93.165286	-58.490566	90.366972
13	42000.0	99.200000	97.885343	-4.347826	96.850394
14	45000.0	97.933333	95.297133	-21.568627	93.654043
15	48000.0	99.000000	97.578208	0.000000	96.559633
16	51000.0	98.966667	96.947365	-3.333333	95.252680

Figure 16: Windowed Results with Neural Network

References

[1] CapyMoa, "capymoa.datasets.Covtype," Available: `https://capymoa.org/api/modules/capymoa.datasets.Covtype.html`. [Accessed: Nov. 3, 2024].

[2] CapyMoa, "Using PyTorch with CapyMoa," Available: `https://capymoa.org/notebooks/03_pytorch.html`. [Accessed: Nov. 3, 2024].

[3] A. Bifet, G. de Francisci Morales, J. Read, G. Holmes, and B. Pfahringer, "Efficient Online Evaluation of Big Data Stream Classifiers," in *Proceedings of the 21st ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '15)*, Sydney, NSW, Australia, 2015, pp. 59–68. Available: `https://doi.org/10.1145/2783258.2783372`.

[4] CapyMoa, "Parallel Ensembles in CapyMoa," Available: `https://capymoa.org/notebooks/parallel_ensembles.html`. [Accessed: Nov. 3, 2024].