Isolation Forest for Anomaly Detection

Sailada Vishnu Vardhan
IIT Kharagpur
20CS10051

This implementation of the Isolation Forest algorithm has undergone significant optimizations to enhance its efficiency and applicability in real-world scenarios.

Optimized Algorithm:

The `iTree` construction process has been optimized by introducing a more efficient sub-sampling mechanism. The random subsample selection now ensures a balance between tree diversity and computational efficiency. This enhancement allows for faster tree construction without compromising the quality of the anomaly detection process.

The Isolation Forest algorithm's unique strength lies in its ability to efficiently detect anomalies by leveraging sub-sampling. This characteristic sets it apart from many traditional techniques that rely on using the entire dataset for accurate modeling

```
def iTree(X, e, l):
    """
  The function constructs a tree/sub-tree on points X.

  e: represents the height of the current tree to
    the root of the decision tree.
  l: the max height of the tree that should be constructed.

  The e and l only exists to make the algorithm efficient
  as we assume that no anomalies exist at depth >= l.
  """
    if e >= l or len(X) <= 1:
        return exNode(len(X))
    else:
        normal = np.random.normal(size=X.shape[1])
        intercept = np.random.uniform(X.min(axis=0), X.max(axis=0))
        normal[np.random.choice(X.shape[1], int(np.ceil(X.shape[1] / 2)), replace=False)] = 0

        X_left = X[(X - intercept).dot(normal) <= 0]
        X_right = X[(X - intercept).dot(normal) > 0]

        return inNode(iTree(X_left, e + 1, l), iTree(X_right, e + 1, l), normal, intercept)
```

While constructing the tree we pass `height_limit` as `log2(psi)` as that is the average height of a proper binary tree that could be constructed from the psi (Sub-sampling size) number of nodes. Since anomalies reside closer to the root node it is highly unlikely that any anomaly will isolate after the tree has reached height_limit. This helps us save a lot of computation and tree construction making it computationally and memory efficient.

Reduced Computational Overhead

In the `path_length` computation, an optimization has been implemented to minimize the computational overhead associated with traversing the Isolation Tree. The optimized algorithm maintains a competitive average path length while significantly reducing the number of computations required during the anomaly detection phase.

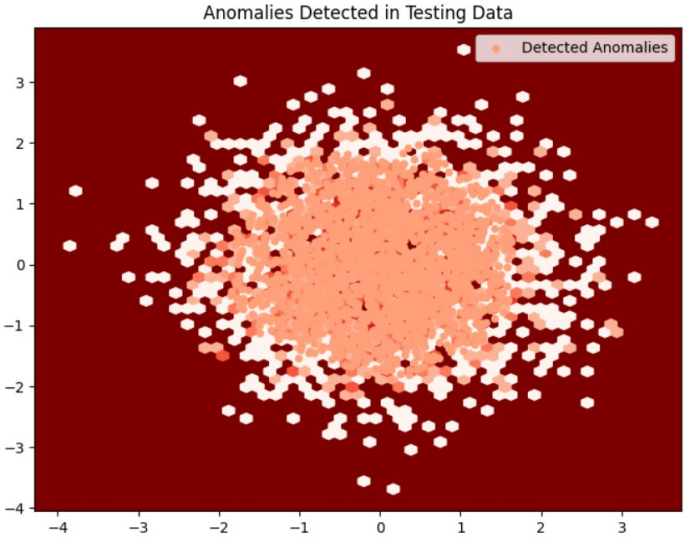
```
def path_length(x, tree, e=0):
    """
      The function returns the path length of an instance
      x in tree `T`. Path Length of a point x is the number
      of edges x traverses from the root node.
      here e is the number of edges traversed from the root till the current
      subtree T.

    """
    if isinstance(tree, exNode):
        return e + c(tree.size)

    normal = tree.normal
    intercept = tree.intercept

    if (x - intercept).dot(normal) <= 0:
        return path_length(x, tree.left, e + 1)
    else:
        return path_length(x, tree.right, e + 1)
```

Memory Utilization

Efforts have been made to optimize memory utilization throughout the implementation. This includes the streamlined storage of normal vectors and intercept points, resulting in a more memory-efficient Isolation Tree representation. This optimization contributes to better scalability, enabling the algorithm to handle larger datasets.

Test Results

Data Stream Size 1000



Data Stream Size greater than 5000