1. MACHINE LEARNING
A data set in machine learning is simply a table representing collections of entries. Each entry, has attributes of which can be thought as the columns in a table. A label describes the class or category of an entry. For binary labels each entry may possess a class label of 0 or 1 of which can be thought of as negative or positive labels. Model and classifier are used synonymously.

Two phases are involved in supervised learning- the training phase and testing phase. The training phase uses a training data set to build a model. The testing phase involves using the model to predict entries from the test data set. After using the model to predict a label, if the predicted label is wrong we describe this as a misclassification. The accuracy rate is the percentage of test set entries that are correctly classified by the model. The accuracy rate is used to summarize the number of correct classifications or misclassifications when comparing different implementations. My work was on using Dafny for the verification of the k-nearest neighbors (kNN) algorithm, one of the most popular machine learning algorithm repository on Github.

2. KNN ALGORITHM
K-nearest neighbor (kNN) algorithm is a lazy learner whereby it learns by new cases as it stores all available cases and classifies new cases based on a similarity measure such as distance functions. A case is classified by a majority vote of its neighbors, with the case being assigned to the class among its kNN measured by a distance function. If \( K = 1 \), then the case is simply assigned to the class of its nearest neighbor. My implementation uses the bfeclidean distance.

Euclidean Distance:
\[
\sqrt{\sum_{i=1}^{k} (x_i - y_i)^2}
\]

Manhattan Distance:
\[
\sum_{i=1}^{k} |x_i - y_i|
\]

Minkowski Distance:
\[
\left[ \sum_{i=1}^{k} (|x_i - y_i|)^q \right]^{\frac{1}{q}}
\]

The Hamming distance is used for categorical variables.

Hamming Distance:
\[
D_H = \sum_{i=1}^{k} |x_i - y_i|
\]

\[
\begin{cases} 
  x = y \Rightarrow D = 0 \\
  x \neq y \Rightarrow D = 1 
\end{cases}
\]

Choosing the optimal value for \( k \) is best done by first inspecting the data. In general, a large \( k \) value is more precise as it reduces the overall noise but there is no guarantee. A good \( k \) can be selected by various heuristic techniques, however, I will not present such techniques as \( k \) is a parameter input from users.

Theorem 1. For each training example \( < x, f(x) > \), add the example to the list of training examples. Given a query instance \( x_q \) to be classified:
- Let \( x_1, x_2, ..., x_k \) denote the \( k \) instances from training examples that are nearest to \( x_q \).
- Return the class that represents the maximum of the \( k \) instances.

As seen in Figure 1, if \( K = 5 \), then query instance \( x_q \) will be classified as negative since three of its nearest neighbors are classified as negative.

3. IMPLEMENTATION IN DAFNY PROGRAMMING LANGUAGE
Dafny relies on high-level annotations to reason about and prove correctness of code. The effect of a piece of code can
be given abstractly, using a natural, high-level expression of the desired behavior, which is easier and less error prone to write. Dafny then generates a proof that the code matches the annotations. Dafny can therefore be used to specify and verify some algorithms. Below are details after using Dafny programming language to implement kNN algorithm. The source code of the kNN implementation using Dafny programming language can be found at [1]. Predicates were used to make the code shorter. A predicate is a function that returns a boolean. Instead of writing out a quantifier multiple times to express a property in a method, I was able to do so by using a predicate. Loop invariant were added were necessary. Loop invariant is an expression that holds upon entering a loop, and after every execution of the loop body. It captures something that is invariant, i.e., does not change, about every step of the loop. As expected methods are used in the program. A method is a piece of imperative, executable code. In other languages, they might be called procedures, or functions. In Dafny, functions are reserved for something even more special. Functions cannot write to memory, and consists solely of one expression. They are required to have a single return value, which is unnamed.

3.1 Initialization and Predicate

Predicate below is used to ensure that the input data is not null and also valid throughout the program. In which case, the train data and test data have the same number of attribute columns. It also ensures that k is at least 1. This is a requirement for the kNN algorithm to successful select at least one neighbor. Another specification is that the numRows variable used within the program is equal to the number of rows in both the testSample and trainSample variable. This is not always the case in real world machine learning data, but it is used to simplify the implementation of the algorithm, so we don’t use different variables to keep track of the number of rows for the testSample and another for the number of rows of the trainSamples. However, the number of attribute columns (i.e numerCols variable) must be equal.

```plaintext
// This predicate express a class invariant:
// All objects of this calls should satisfy this.
predicate Valid()
reads this, this.testSamples, this.trainSamples; 
{ 
  testSamples != null && numRows > 0 &&
  numRows == testSamples.Length0 &&
  numCols > 0 &&
  numCols == testSamples.Length1 &&
  trainSamples != null && numRows > 0 &&
  numRows == trainSamples.Length0 &&
  numCols > 0 &&
  numCols == trainSamples.Length1 &&
  kVal >= 1 && trainClasses != null &&
  numRows == trainClasses.Length
}
```

//Note that Init methods always typically need to state
//that they will modify this object, as their purpose
//is to initialise all the fields.
method Init(k: int, rows: int, cols: int, 
  trainclasses: array < int >, 
  trainsamples : array2 <int>, testsamples : array2 <int> ) modifies this;
//number of rows and columns must be greater than 0
requires rows > 0 && cols > 0 && k > 0;
requires trainclasses != null;
requires trainsamples != null;
requires testsamples != null;
ensures trainSamples != null;
ensures testSamples != null;
ensures trainClasses != null;
requires trainclasses.Length == rows;
requires trainsamples.Length0 == rows;
requires testsamples.Length0 == rows;
requires trainsamples.Length1 == cols;
requires testsamples.Length1 == cols;
//make sure initializes numRows and numCols
//from user input
ensures numRows == rows;
ensures numCols == cols;
//make sure initializes kVal from user input
ensures kVal == k;
//ensure row and column length same as user input
ensures numRows == trainSamples.Length0 &&
  numRows == testSamples.Length0 &&
  numCols == trainSamples.Length1 &&
  numCols == testSamples.Length1;
{ 
  numRows := rows;
  numCols := cols;
  trainClasses := trainclasses;
  trainSamples := trainsamples;
  testSamples := testsamples;
  assert numRows == trainSamples.Length0 &&
  numRows == testSamples.Length0;
  assert numCols == trainSamples.Length1 &&
  numCols == testSamples.Length1;
  kVal := k;
}

3.2 Pre- and Postconditions of Classification Method

The power of Dafny comes from the ability to annotate methods to specify their behavior. The classification method in my implementation combines both train and test method of your typical machine learning programs. The steps to perform classification is illustrated in the code below, the full source code can be obtained here.

```
method classify()returns(assignedTestClass: array < int >)
  requires this.Valid();
  { 
    numRows == trainSamples.Length0 &&
    numCols > 0 &&
    numCols == trainSamples.Length1 &&
    trainSamples != null &&
    numRows == trainClasses.Length
  }
```
ensures assignedTestClass != null;
ensures assignedTestClass.Length == testSamples.Length0;
   // number of rows must be the same
   // for both test and train data
requires numRows == trainSamples.Length0 &&
numRows == testSamples.Length0;
   // number of columns must be the same
   // for both test and train data
requires numCols == trainSamples.Length1 &&
numCols == testSamples.Length1;
   // make sure predicate is still valid
ensures this.Valid();
{
    // initialize distances array to 0.0
    // initialize assignedTestClass array to -1

    // knn algorithm:
    // for each test sample
    // calculate test sample distance to each train sample
    // classify test sample to label of closest
    // . . . train sample
    // sort all distances
    // find the top k train samples
    // classify test sample with majority vote
    // . . . from k train samples
    // assign voted label to assignedTestClass array
    ...
}
3.3 Pre- and Postconditions of Subroutine Calls

Using a functions or subroutine calls can help simplify the specification process for the developers and as such it was frequently used in this program.

getRowIndex, arrayCopy, getRow, getDistance, distanceTo, and sort

were invoked in the classify method. Below are the pre and post conditions of the subroutine calls:

```java
/**
** getRowIndex(...) - obtains and returns row index from 2d array
** a : array<real> - array to obtain row index from
** key : real - distance value (or key) used to obtain index from row
** returns rowIndex : int - return rowIndex i.e training sample/row location
**
** method getRowIndex(a: array2 < real > , key: real) returns(rowIndex: int)
requires a != null;
//ensures a.Length0 >= rowIndex;
requires a.Length1 == 2;
requires forall k: nat::k < a.Length0 ==> 0 <= a[k, 1].Trunc < a.Length0;
ensures 0 <= rowIndex ==> exists k::0 <= k < a.Length0 && a[k, 0] == key;
ensures 0 > rowIndex ==> forall k::0 <= k < a.Length0 ==> a[k, 0] != key;
ensures rowIndex < a.Length0;
{
...
}
*/

/**
** getRow(...) - obtains and returns row data from either training data (true) or test data (false)
** rowNum : int - row number of obtain from data
** tr : bool - is get row from training data (true) or test data (false)
** returns row : array<real> - return row data from either training data (true) or test data (false)
**
** method getRow(rowNum: int, tr: bool) returns(row: array < real >)
//predicate must be valid
requires this.Valid();
//rowNum must be within range of both test and train data
requires 0 <= rowNum < trainSamples.Length0 && 0 <= rowNum < trainSamples.Length0;
//number of columns must be the same for both test and train data
requires numCols == trainSamples.Length1 && numCols == trainSamples.Length1;
//tr is bool and must be either true or false
requires tr == true || tr == false;
//ensure scope of tr never changes
ensures tr == true || tr == false;
//make sure predicate is still valid
ensures this.Valid();
//make sure returned row is not null
ensures row != null;
//the return row data should have the same length as test and train data's column
ensures row.Length == numCols;
//if row from training data is request, all cell values in training data matches the returned row data
ensures forall k: nat::k < numCols && tr ==> row[k] == real(trainSamples[rowNum, k]);
//if row from test data is request, all cell values in test data matches the returned row data
ensures forall k: nat::k < numCols && !tr ==> row[k] == real(testSamples[rowNum, k]);
{
...
}
*/

/**
** getDistance(...) - calculates and returns square of Euclidean distance between two vectors
** assumes sample1 and sample2 are valid i.e. same length
** sample1, sample2: array<real> - array (or vectors) with cell values
** returns distance : real - eucildean distance result
**
** method getDistance(sample1: array < real > , sample2: array < real > ) returns(distance: real)
```
//sample1 and sample2 must not be null;
requires sample1 != null & sample2 != null;
//sample1 and sample2 must have same length
requires sample1.Length == sample2.Length;
//predicate must be valid
requires this.Valid();
//make sure sample1 and sample2 have same length
ensures sample1.Length == sample2.Length;
//make sure predicate is valid
ensures this.Valid();
//make sure the distance is correct
ensures distance == distanceTo(sample1, sample2, sample1.Length);
{
...
}

/**
 ** sort(..) - sort 2d array based on row as sort-key
 ** creates a new copy of sorted array and returns it
 ** a: array<real> - array to be sorted with key (distance in kNN alg) i.e a[key,index]
 ** res: array2 < real > - sorted array
 ***/
method sort(a: array2 < real > ) returns(res: array2 < real > )
requires a != null;
requires a.Length0 >= 1;
requires a.Length1 == 2;
requires forall k: nat::k < a.Length0 ==> a[k, 1].Trunc <= a.Length0;
ensures res != null;
ensures res.Length1 == 2;
ensures sorted(res, 0, res.Length0);
ensures res.Length0 >= 1;
ensures res.Length0 == a.Length0;
ensures res.Length1 == a.Length1;
{
...
}

Additional predicates were to ensure the result of the distance and sort method are valid.

/**
 ** distanceTo(..) - mathematical, recursive definition of the
 ** distance function from 0 to n-1 over two arrays
 ** it is needed to specify the functionality of the method above
 ** sample1, sample2: array<real> - array (or vectors) with cell values
 ** real - eculidean distance result
 ***/
function distanceTo(sample1: array < real > , sample2: array < real > , n: int): real
//sample1 and sample2 must not be null;
requires sample1 != null & sample2 != null;
//sample1 and sample2 must have same length
requires sample1.Length == sample2.Length;
// index n should be within sample1 and sample2 range
requires 0 <= n && n <= sample1.Length;
requires 0 <= n && n <= sample2.Length;
//make sure sample1 and sample2 have same length
ensures sample1.Length == sample2.Length;
decreases n;
reads sample1, sample2; {
if (n == 0) then 0.0
else distanceTo(sample1, sample2, n - 1) + ((sample1[n - 1] - sample2[n - 1]) * (sample1[n - 1] - sample2[n - 1]))
}
predicate sorted(a: array2 < real > , lo: int, hi: int)
requires a != null;
requires a.Length1 == 2;
//requires 0 <= i <= a.Length0;
reads a; {
forall j, k:0 <= lo <= j < k < hi <= a.Length0 ==> a[j, 0] <= a[k, 0] 
}

4. CONCLUSION
Annotations are written with the Dafny programming language and they are checked at run time to prove a piece of code is correct. This report presented an implementation of the kNN algorithm using Dafny. The full source code can be found here [1].

5. REFERENCES