# EMPLOYING AN EFFICIENT AND SCALABLE IMPLEMENTATION OF THE COST SENSITIVE ALTERNATING DECISION TREE ALGORITHM TO EFFICIENTLY LINK PERSON RECORDS

by

Clark Raymond Phillips, B.S.

A thesis submitted to the Graduate Council of
Texas State University in partial fulfillment
of the requirements for the degree of
Master of Science
with a Major in Computer Science
May 2015

Committee Members:

Anne H.H. Ngu, Chair

Byron J. Gao

Yijuan Lu

## FAIR USE AND AUTHOR’S PERMISSION STATEMENT

### Fair Use

This work is protected by the Copyright Laws of the United States (Public Law 94-553, section 107). Consistent with fair use as defined in the Copyright Laws, brief quotations from this material are allowed with proper acknowledgment. Use of this material for financial gain without the author’s express written permission is not allowed.

### Duplication Permission

As the copyright holder of this work I, Clark Phillips, authorize duplication of this work, in whole or in part, for educational or scholarly purposes only.

## ACKNOWLEDGEMENTS

I acknowledge my advisor Dr. Anne H.H. Ngu. I am grateful for her presence, willingness, and support during my tenure at Texas State University. I also appreciate my thesis committee members Dr. Byron J. Gao and Dr. Yijuan Lu. Their participation and support was integral in completing my Masters thesis as I learned from them directly in their respective graduate classes offered at Texas State University: Data Mining and Machine Learning. I would like to recognize two individuals from the Texas State University Geography department. Ryan Thomas Schuermann and Dr. T. Edwin Chow who graciously provided the datasets used in this thesis. I would like to recognize my fellow student Stan Thornhill as our team projects in various classes were the beginning stages of this thesis. I would like to recognize Esteban Diocares, Meredith Joe, and Nancy Phillips. All three of these individuals were invaluable in formatting and editing this thesis. I stand in gratitude and appreciation to all.

## TABLE OF CONTENTS

**Page**

# LIST OF TABLES

**Table** **Page**

## LIST OF FIGURES

**Figure** **Page**

## LIST OF ABBREVIATIONS

| Abbreviations | Description |
|---|---|
| CS – Computer Science | Field of study concerning computers (theoretical and applied) |
| ML – Machine Learning | Sub-field of Computer Science involving algorithms that learn |
| RL – Reinforcement Learning | Sub-field of Computer Science involving algorithms that learn |
| RL – Record Linkage | Specific task for a ML technique to perform involving linking identical records together |
| ER – Entity Resolution | The task of finding records in a data set that refer to the same entity across different data sources |
| ADT – Alternating Decision Tree | Specific ML technique |
| CSADT – Cost Sensitive Alternating Decision Tree | Specific ML technique |
| NBA – National Basketball Association | The professional basketball league in the USA |

| | |
|---|---|
| UI – User Interface | Refers to the interface between humans and code |
| GUI – Graphical User Interface | Refers to the graphical interface between humans and code |
| ACM – Association of Computing Machinery | World's largest educational and scientific computing society |
| IEEE – Institute of Electrical and Electronics Engineers | World's largest association of technical professionals |
| BMI – Body Mass Index | A numerical computation regarding height and weight |
| EM – Expectation-maximization | Specific ML technique |
| CLS – Concept Learning System | Specific ML technique |
| ACLS – Analog Concept Learning System | Specific ML technique |
| ID3 – Interactive Dichotomizer 3 | Specific ML technique |
| CART – Classification and Regression Trees | Specific ML technique |
| SLIQ – Supervised Learning In Quest | Specific ML technique |
| SPRINT – Scalable Parallelizable Induction of Decision Tree | Specific ML technique |

| | |
|---|---|
| SQL – Structured Query Language | Special-purpose programming language designed for managing data held in a relational database management system |
| STL – Standard Template Library | Software library for the C++ programming language that influenced many parts of the C++ Standard Library |
| TDI – Top Down Induction | A decision tree strategy for constructing nodes from the top to the bottom using induction |

# ABSTRACT

When collecting person records for census, identifying individuals accurately is paramount. Over time, people change their phone numbers, their addresses, even their names. Without a universal identifier such as a social security number or a finger-print, it is difficult to know whether two distinct person records represent the same individual. The Cost Sensitive Alternating Decision Tree (CSADT) algorithm (a supervised learning algorithm) is employed as a Record Linkage solution to the problem of resolving whether two person records are the same individual. A person record consists of several attributes such as a name, a phone number, an address, etc. The number of person-record-pairs grows exponentially as the number of records increase. In order to accommodate this exponential growth, a scalable implementation of the CSADT algorithm was employed. A thorough investigation and evaluation are presented demonstrating the effectiveness of this implementation of the CSADT algorithm on linking person records.

# 1. INTRODUCTION

A first, middle (initial perhaps), and last name typically formulate a unique identifier for an individual. In the digital age, a repository of personal information is available on the Internet. People-finder sites, such as WhitePages.com and Intelius.com, provide easy access to these personal demographics based on simple criteria such as surname and zip code. A simple search at one of these sites results in several person records with similar if not identical names demonstrating just how far a person's name is from unique. Common names, such as Joe Smith, require auxiliary information (address, phone number, date of birth, etc.) in order to ascertain an individual's identity. Such data are not often available while some attributes, such as address and phone number, are subject to change even when they are available. The moving population produces multiple records with identical attributes except their address. If, by some chance, these records were linked together correctly as the same individual, which record would contain the current address? The adoption of an English name, a common practice of cultural assimilation among international immigrants in western countries, often results in records with the first and middle names (or initial) interchanged. Name changes among women according to the marital status is a longstanding challenge in surname analysis. People take on a personal name suffix, such as Sr. or Jr., adding to the difficulty. Some take on a professional suffix, such as Ph.D. or M.D., later in life again increasing the difficulty in identifying the individual correctly.

Clearly, the task of resolving person records is a task that the most sophisticated parsing script could not complete. Some Machine Learning (ML) techniques were specifically designed to accurately solve Record Linkage (RL) problems such as linking people records. RL is the process of matching, resolving, and linking multiple records of the same entity from multiple data sources. It is a common technique in database processing and consolidation. Unfortunately, data often lack a unique, global identifier. The data are neither carefully controlled for quality nor defined in a consistent way across different sources (Elmagarmid 2007). RL can serve an important role in upholding data integrity, maintaining data quality, and enhancing pattern discovery.

In the context of people demographics, RL introduces scale challenges. Suppose a person record *i* containing *k* attributes is one instance in a dataset of size *n*. The person record *i*'s k attributes would need to be compared against all other (*n*-1) person records' *k* attributes in order to ensure minimum duplication. To compare the text attributes of a person record, a text comparing algorithm must be employed such as the Levenshtein distance (Levenshtein 1999).

It is essential to carefully implement an appropriate RL algorithm to detect, identify, and resolve entities in question to remove and consolidate duplicate records across multiple data sources. Several RL techniques were surveyed in the search for an algorithm that could link such records. The most successful combinations have been boosting decision trees (Freund 1999). Boosting uses all instances at each repetition but maintains a weight for each instance in the training set that reflects its importance. Adjusting the weights causes the learner to focus on different instances and leads to pure classifiers (Quinlan 1996). RL models based on Cost Sensitive Alternating Decision Trees (CSADT), an algorithm that uniquely combines boosting and decision tree algorithms creates shorter and easier-to-interpret linking rules. Experiments show that the proposed models significantly outperformed other baselines on the desired industrial operating points, and the improved understanding of the model's decisions led to faster debugging and feature development cycles (Chen 2011).

In this thesis, the CSADT algorithm was selected to explore the challenge of linking person records. The CSADT algorithm carefully examines as many conditions as it's offered and ultimately selects the *best* conditions as splitter nodes in the decision tree that it creates. The CSADT algorithm uses a cost sensitive, boosting, induction step when searching for its next splitter node. Essentially, a CSADT tree is an AND/OR graph. Knowledge contained in the tree is distributed as multiple paths must be traversed to form predictions. Instances that satisfy multiple splitter nodes have the values of prediction nodes that they reach summed to form an overall prediction value (Holmes 2002). A positive sum represents one class and a negative sum the other in the two-class setting. The result is a single interpretable tree with predictive capabilities that rival a committee of boosted C5.0 trees (Freund 1999). While the CSADT algorithm produces telling decision trees which have dissected the training data to illuminate the patterns therein, it is limited by the quality of the conditions available (typically created by the user). A novel approach to solving the generation of telling conditions is explored in this thesis. Conditions are generated automatically from a statistical analysis of the instances' attributes on matches and non-matches from the person record dataset to maximize classification prediction accuracy.

As the number of person records used in training the CSADT algorithm increases, the number of record-pairs (unique person-to-person pairs) grows exponentially. In tuning the CSADT algorithm, it is pertinent that the creation of the CSADT tree be created sufficiently quickly (minutes to hours rather than days to weeks). Therefore, the implementation of the CSADT algorithm needs to be able scale linearly in order to effectively provide decision trees.

The CSADT algorithm was originally implemented in Python and later implemented in C++. The most computationally expensive task the CSADT algorithm performs is the search for the best condition. Fortunately, this task is a parallelizable. Consequently, C++ boost threads (an element in the most predominant third party library) were ultimately employed to perform this task. This thesis contains a thorough investigation of the effects on computation time of the CSADT algorithm using different programming languages CPU architecture.

Finally, the CSADT algorithm was applied to another unique dataset (from a different domain), specifically NBA team/player statistics asking questions such as, “What is the single most important facet of the game of basketball that a team X should pursue to increase their chances of securing spot in the playoffs?” The CSADT algorithm performed exceptionally as reported in the Evaluation section.

Lastly, a detailed comprehensive plan is included on what further improvements might be pursued in an effort to decrease CPU cycles while the CSADT algorithm is underway and to improve the classification prediction accuracy of the CSADT tree.

In summary, the main contribution of the thesis is the demonstration of the CSADT algorithm as an efficient, scalable, and accurate machine learning technique for linking person records when implemented using threads in C++. Ultimately, the training dataset resulting in the most accurate CSADT tree was built from 100 training records (4,950 record-pairs), considering 171 conditions, and generated 30 nodes. The pre-processing of the training data in conjunction with the automatically generated 171 high quality conditions were paramount in achieving 99.95% prediction accuracy (89.40% precision and 87.39% recall).

## 2. BACKGROUND

Computer Science (CS) is the study of the principles and use of computers. It is the study of automating algorithmic processes that scale. It is the science that deals with the theory and methods of processing information in digital computers, the design of computer hardware and software, and the applications of computers. CS spans a range of topics from theoretical studies of algorithms and the limits of computation to the practical issues of implementing computing systems in hardware and software.

The Association for Computing Machinery (ACM) and the IEEE Computer Society (IEEE-CS) have specified the following four areas of computer sciences as crucial:

1. Theory of computation
2. Algorithms and data structures
3. Programming methodology and languages
4. Computer elements and architecture

Although the above list has been deemed as “crucial by the ACM and IEEE-CS, there are other important subfields of computer science that remain:

- Software engineering
- Artificial intelligence
    - Data mining
    - Machine learning
- Computer networking and telecommunications
- Database systems
- Parallel computation

- Distributed computation
- Computer-human interaction
- Computer graphics
- Operating systems numerical
- Symbolic computation

## 2.1 Machine Learning

Machine learning (ML) falls under the artificial intelligence umbrella of computer science and is the study of algorithms that can literally learn. The word “learning” is typically associated with people and animals, however, some machines are equipped with the capacity to learn as well. Learning comes in different forms including the ability to acquire new information; modify and reinforce knowledge that already exists; and changes behaviors, skills, values, and/or preferences. ML algorithms typically fall into one of three sub-categories:

- Supervised Learning
- Unsurprised Learning (also sometimes called clustering)
- Reinforced Learning

The CSADT algorithm is a supervised learning technique that has been thoroughly explored, implemented, and analyzed in this thesis. When a classification question containing attribute-rich data exists, a supervised learning technique is needed. For example, a survey is given to multiple individuals resulting in the following attributes:

- Age

- Height
- Gender
- Political Affiliation
- BMI
- Salary

Given these attributes, a supervised learning technique can be applied to build a decision tree that would aid in the prediction of a classification. In order for the ML algorithm to build its tree, it first needs to model the attributes in the dataset and then consume the dataset. Trends and patterns are illuminated in the decision tree as the ML algorithm is consuming the training data. The decision tree is then used on "testing data" in order to make predictions. Obviously, there is an emphasis on the training data being diverse and telling as the algorithm will only be able to discover patterns present in the training data.

Many unique supervised learning algorithms and structures have been developed; all serve as candidates in solving a classification or regression problem. Below is a list of better known techniques:

- Decision trees
- Ensembles (Bagging, Boosting, Random forest)
- k-NN
- Linear Regression
- Naïve Bayers
- Neural Networks

- Logistic Regression
- Perceptron
- Support Vector Machine (SVM)
- Relevance Vector Machine (RVM)

This report examines decision trees in general while specifically focusing on the CSADT algorithm.

Unsupervised learning algorithms are applied to problems that do not have training data. There is no way to evaluate the effectiveness of a clustering technique on as a solution to a problem. Without the ability to train on training data, unsupervised (or clustering) techniques must make the assumption that instances with the same classification (or label) have similar attributes. Several clustering algorithms have been developed, such as:

- BIRCH
- Hierarchical
- k-means
- Expectation-Maximization (EM)
- DBSCAN
- OPTICS
- Mean-Shift

Finally, reinforcement learning is applied to a problem in a defined environment with a limited number of choices as actions. The game of chess is an example. The environment is known, well defined (meaning it can be modeled), and includes a finite number of moves that can be taken on any given turn. Reinforcement learning attempts to maximize at every level in order to achieve the desired result. Unsurprisingly, reinforcement learning is studied in depth in fields such as game theory, control theory, and information theory. It has also been applied to simulation-based optimization, statistics, and genetic algorithms.

### 2.2 Record Linkage

Record Linkage (RL) is the process of extracting records from a dataset and then matching them to other records. This challenge has presented itself in several disciplines, including but not limited to:

- Information Retrieval
- Machine Learning
- Statistics
- Natural Language Processing
- Database Management

Unsurprisingly, RL has taken on several names as different disciplines prefer certain descriptions. Common names are listed below:

- Record Linkage
- Entity Resolution
- Deduplication
- Reference Reconciliation
- Co-Reference Resolution
- Object Consolidation

Record Linkage has been applied to many problems spanning several disciplines. For example, when assimilating information on a subject, identical information is often presented differently (i.e.: 313 West Wallaby Street vs. 313 W. Wallaby St.). When this information is consumed by a computer, it is rarely perceived as identical information and thus results in the generation of erroneous duplicates. It would be ideal to recognize these duplicate records and combine their attributes into one all-encompassing record. This problem demands the direct application of an RL algorithm.

There are more complex problems that RL can be applied to. Suppose a potential new customer is investigating a music website and has demonstrated interest in a few songs by rating them. Record Linkage could be used to match the potential new customer to an existing user whose preferences are already known through their ratings. Once a match was made, the current customer's content would be delivered to the potential new customer.

Similarly, online consumers are often offered “suggested” products upon showing interest in one. Record linkage could be used to match a customer to a different customer with a similar purchase history and recommend additional items to purchase.

Still, there are other applications such as trying to model and respond to customers’ online behavior. Suppose a customer browses a new site, clicking on several links before finally making a purchase. Suppose a different customer browses the same site, also clicking on several links, but does not make a purchase. If several attributes regarding the specific behaviors of the many customers were organized into a dataset, an RL algorithm could consume that dataset. An RL algorithm would ultimately produce an understandable description of how purchasing visitors behave versus non-purchasing visitors. The trends and/or patterns which an RL algorithm would illuminate would provide a right feedback loop to the site designers. Obviously, the results derived by the RL algorithm should be incorporated into the site in an attempt to encourage more visitors to make a purchase.

In the domain of competitive sports, it would be useful to be able to predict the outcome of a sports team's game or season. Aligning "duplicates" on how the team performed in the past compared to today would yield insight as to how the team will perform in the future. Matching a player whose personal sports statistics are similar to a retired player would allow a team to predict the future productivity, or the lack thereof, of that player as they age.

The applications outlined above were specific examples of how RL can and has been used to solve difficult problems. It is clear that there are a multitude of real world situations that can benefit from an RL solution. Record Linkage has been used to help solve problems in the following domains.

- Advertising
- Marketing
- Online Product Shopping
- Online Services Shopping
- Knowledge Management
- Biometrics
- Nature
- Medical Conditions
- Network Science

- Database Management
- Sports Predictions

Although RL has already been applied to many problems, problems remain still whose solutions might lie in the application of an RL technique. Increased awareness for the need of RL results as buzz words such as “big data” and “analytics” are commonly used in both academia and industry. We are inundated with increasing data that needs to be integrated, aligned, and matched before further utility can be extracted (Getoor 2012).

# 3. LITERATURE SURVEY

## 3.1 Decision Trees

Decision trees are a structure that some supervised learning techniques use as a means to model data. Test data is then run through the decision tree resulting in a prediction/decision being made.

In the broadest sense, a decision tree is simply a graph, although generally a decision tree is a tree-like graph comprised of nodes, including but not limited to:

- The root node
- Attribute nodes
- Splitter nodes
- Prediction nodes
- Leaf Nodes

A decision tree is a tree-structured plan of a set of testable conditions used in order to make a prediction. Decision trees are constructed one node at a time starting with the root node. The root node is the first and topmost node. Leaf nodes are the bottom most nodes and may contain a prediction/classification. The remainder of the nodes are known as the tree's internal nodes. Attribute and decision nodes are comparators. Each is responsible for asking a question about a specific attribute. The tree is traversed in an order depending on the answers to the conditions contained within the nodes. Prediction nodes connect to splitter or decision nodes and often contain a score. In the CSADT algorithm, each boosting iteration adds a splitter node (containing a condition) and two predictor nodes (each containing a score). The condition contained in the splitter node is selected above other conditions based on its purity score (Pfahringer 2001).

Better known decision tree algorithms include:

- Concept Learning Systems (CLS)
- Interactive Dichotomizer 3 (ID3)
- C4.5 (Salzberg 1994)
- Classification and Regression Trees (CART)
- Supervised Learning In Quest (SLIQ)
- Scalable Parallelizable Induction of Decision Tree Algorithm (SPRINT)
- Alternating Decision Tree (ADT)
- Cost-Sensitive Alternating Decision Tree (CSADT)

Regardless of the type of decision tree algorithm, common denominators exist:

- Input: Training data
    - each entity has several attributes
    - each entity has a classification
- Input: Conditions (attribute comparators) are derived by one of the following.
    - the attributes themselves are conditions
    - the conditions are provided by the user
    - the conditions are generated by the algorithm
- Output: Decision tree
- Root node is determined by one of the following
    - selected via induction
    - created manually

- Induction
    - Typically, a decision repeats an induction step, selecting a condition as a node upon every iteration until the decision tree is complete. Some trees dictate their own completion based on whether or not all of the records in the training data have been accurately classified or after having exhausted all available conditions. Still other trees run until they have contain an arbitrary number of nodes typically dictated by the user.

The induction step is paramount for a decision tree algorithm because during this step, conditions are selected and added to the tree as nodes. “The essence of induction is to move beyond the training set, i.e., to construct a decision tree that correctly classifies not only objects from the training set, but other (unseen) objects as well.” (Quinlan 1986). In order to accomplish this, the decision tree should illustrate a relationship between an entity’s attributes and its classification.

### 3.2 Interactive Dichotomizer 3

The Interactive Dichotomizer (ID3) is one of the most popular inductive decision tree algorithms. In order to understand how the ID3 constructs its decision tree, an examples is offered with its respective training dataset and a step-by-step illustration of how that training data is used to build a decision tree. Below is a table containing the training dataset. There are four records and three attributes:

**Table 1: ID3 Records**

| | Attribute 1 | Attribute 2 | Attribute 3 | Classification |
|---|---|---|---|---|
| | Regularly Votes | Attends Church | College Graduate | Political Typology |
| Record 1 | No | No | No | Conservative |
| Record 2 | No | No | Yes | Conservative |
| Record 3 | No | Yes | No | Liberal |
| Record 4 | No | Yes | Yes | Liberal |

The three attributes from the above table are listed below (all three attributes expect a “Yes” or “No” response).

- Regularly Votes
- Attends Church
- College Graduate

The classification is a political typology which is expected to be either “Conservative” or “Liberal”. The ID3 algorithm is described below in six steps:

1. Determine the entropy of all available attributes (all attributes are available to start with)
    - Equivalent statement: Determine the information gain of all available attributes.
2. Select the attribute whose entropy is lowest
    - Equivalent statement: Select the attribute whose information gain is highest.
3. Split the entities into subsets based on the attribute selected in step 2.
4. Create a node containing the attribute from step 2 and add it to the decision tree.
5. Remove the attribute selected from step 2 from the available attributes.

6. Repeat steps 1-5 on the “yes” and “no” sides of the node selected in step 2.

### 3.2.1 Entropy

ID3 uses entropy in order to induce which attribute is *best*. “The induction task is to develop a classification rule that can determine the class of any object from its values of the attributes.” (Quinlan 1986) Information Theory defines entropy as the measure of the uncertainty associated with a variable. It is the measure of chaos associated with an attribute. High entropy means chaotic while low entropy means order. In step two above, ID3 selects the attribute whose entropy is *lowest*, or least chaotic. It selects the attribute that is the most telling with respect to its classification. For example, if a single attribute perfectly classifies all entities, that attribute would have the lowest entropy and therefore be selected. Mathematically, entropy is defined as:

$$Given\ collection\ S\ of\ \ c\ outcomes$$
$$I = a\ given\ class$$
$$p(I) = probability\ of\ S\ belonging\ to\ class\ I$$
$$Entropy(S) = -\sum_{i=1}^{c}[p(I)\cdot\log_2(I)]$$

### 3.2.2 Information Gain

Information gain is the change in entropy. When entropy is low, information gain is high. Essentially, the following two statements’ results are identical:

- Select the attribute whose entropy is lowest
- Select the attribute whose information gain is highest

Mathematically, information gain is defined as:

$$Gain(S, attribute) = Entropy(S) - (attribute\ proportion)\cdot \mathrm{Entropy(attribute)}$$

### 3.2.3 ID3 Example

Information Gain measures how well an attribute correctly classifies entities. The data clearly shows that if someone attends church, they are conservative; Conversely, if someone does not attend church, they are liberal. A very simple decision tree would contain only one node addressing the church attendance attribute. Although this single node decision tree would suffice for this small dataset, a detailed description of how the ID3 algorithm would digest this dataset and create a decision tree is provided.

**Table 2: ID3 Entropy**

| Dataset Entropy | |
|---|---|
| **Conservatives** | **Liberals** |
| 1 | 0 |
| 1 | 0 |
| 0 | 1 |
| 0 | 1 |

$$Entropy(S) = Entropy(Conservatives) + Entropy(Liberals)$$

$$Entropy(Conservatives) = -\frac{2}{4}\log_2\frac{2}{4} = 0.5$$

$$Entropy(Liberals) = -\frac{2}{4}\log_2\frac{2}{4} = 0.5$$

$$Entropy(S) = 0.5 + 0.5 = 1$$

#### 3.2.3.1 Attribute: Regularly Votes

The next step is to discover which of the available attributes carries the lowest entropy (or highest information gain). The first attribute considered is “regularly votes”. In order to determine the entropy of the voting attribute, the table and equations are provided below.

**Table 3: ID3 Attribute Vote**

| | | | Votes | | | !Votes | |
|---|---|---|---|---|---|---|---|
| Votes | !Votes | | Conservative | Liberal | | Conservative | Liberal |
| 0 | 1 | | 0 | 1 | | 1 | 0 |
| 0 | 1 | | 0 | 1 | | 1 | 0 |
| 0 | 1 | | 0 | 1 | | 0 | 1 |
| 0 | 1 | | 0 | 1 | | 0 | 1 |

$$Gain(S, regularly\ votes) = Entropy(S) - L(votes)$$

$$Gain(S, regularly\ votes) = 1 - L(votes)$$

$$L(votes) = (votes\ ratio) \cdot Entropy(S, votes) + (!\, votes\ ratio) \cdot Entropy(S, !\, votes)$$

$$votes\ ratio = \frac{0}{4} = 0$$

$$!\, votes\ ratio = \frac{4}{4} = 1$$

$$number\ of\ voters = 0$$

$$Since\ the\ number\ of\ voters\ is\ 0, Entropy(S, voting) = 0$$

$$number\ of\ non\ voters = 4$$

$$number\ of\ non\ voting\ conservatives = 0$$

$$ratio\ of\ non\ voting\ conservatives\ to\ all\ non\ voters\ = \frac{2}{4} = 0.5$$

$$number\ of\ non\ voting\ liberals = 2$$

$$ratio\ of\ non\ voting\ liberals\ to\ all\ non\ voters = \frac{2}{4} = 0.5$$

$$Entropy(S, !\, voting) = -0.5 \cdot log_2(0.5) + -0.5 \cdot log_2(0.5) = 1$$

$$Gain(S, regularly\ votes) = 1 - (0 \cdot 0 + 1 \cdot 1) = 0$$

#### 3.2.3.2 Attribute: Attends Church

The second attribute considered is “attends church”. In order to determine the entropy of the church attendance attribute, table and equations are provided below.

**Table 4: ID3 Attribute Church**

| | | | Attends Church | | | !Attends Church | |
|---|---|---|---|---|---|---|---|
| Attends Church | !Attends Church | | Conservative | Liberal | | Conservative | Liberal |
| 0 | 1 | | 0 | 0 | | 1 | 0 |
| 0 | 1 | | 0 | 0 | | 1 | 0 |
| 1 | 0 | | 0 | 1 | | 0 | 0 |
| 1 | 0 | | 0 | 1 | | 0 | 0 |

$$Gain(S, attends\ church) = Entropy(S) - L(church)$$

$$Gain(S, attends\ church) = 1 - L(church)$$

$$L(church) = (church\ ratio) \cdot Entropy(S, church) + (!\,church\ ratio) \cdot Entropy(S, !\,church)$$

$$church\ ratio = \frac{2}{4} = 0.5$$

$$!\,church\ ratio = \frac{2}{4} = 0.5$$

$$number\ of\ church\ attenders = 2$$

$$number\ of\ church\ attending\ conservatives = 0$$

$$ratio\ of\ church\ attending\ conservatives\ to\ all\ church\ attenders = \frac{0}{2} = 0$$

$$number\ of\ church\ attending\ liberals = 2$$

$$ratio\ of\ church\ attending\ liberals\ to\ all\ church\ attenders = \frac{2}{2} = 1$$

$$Entropy(S, church) = -0 \cdot log_2(0) + -1 \cdot log_2(1) = 0 + 0 = 0$$

$$Although\ log_2(0)\ is\ an\ impossible\ computation, since\ it\ is\ multiplied\ by\ 0, the\ result\ is\ 0$$

$$number\ of\ non\ church\ attenders = 2$$

$$number\ of\ non\ church\ attending\ conservatives = 2$$

$$ratio\ of\ non\ church\ conservatives\ to\ all\ non\ church\ attenders = \frac{2}{2} = 1$$

$$number\ of\ non\ church\ attending\ liberals = 0$$

$$ratio\ of\ non\ church\ attending\ liberals\ to\ all\ non\ church\ attenders = \frac{0}{2} = 0$$

$$Entropy(S, !\,church) = -1 \cdot log_2(1) + -0 \cdot log_2(0) = 0 + 0\ \ = 0$$

$$Although\ log_2(0)\ is\ an\ impossible\ computation, since\ it\ is\ multiplied\ by\ 0, the\ result\ is\ 0$$

$$Gain(S, attends\ church) = 1 - (0.5 \cdot 0 + 0.5 \cdot 0) = 1$$

### 3.2.3.3 Attribute: College Graduate

The third and final attribute considered is “college graduate”. In order to determine the entropy of the college graduate attribute, the table and equations are provided below.

**Table 5: ID3 Attribute College Graduate**

| | | College Graduate | | !College Graduate | |
|---|---|---|---|---|---|
| College Graduate | !College Graduate | Conservative | Liberal | Conservative | Liberal |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 |

$$Gain(S, college\ graduate) = Entropy(S) - L(college)$$

$$Gain(S, college\ graduate) = 1 - L(college)$$

$$L(college) = (college\ ratio) \cdot Entropy(S, college) + (!\,college\ ratio) \\ \cdot Entropy(S, !\,college)$$

$$college\ ratio = \frac{1}{2} = 0.5$$

$$!\,college\ ratio\ = \frac{1}{2} = 0.5$$

$$number\ of\ college\ graduates = 2$$

$$number\ of\ college\ graduating\ conservatives = 1$$

$$ratio\ of\ college\ graduating\ conservatives\ to\ all\ college\ graduates = \frac{1}{2} = 0.5$$

$$number\ of\ college\ graduating\ liberals = 1$$

$$ratio\ of\ college\ graduating\ liberals\ to\ all\ church\ attenders = \frac{1}{2} = 0.5$$

$$Entropy(S, college) = -0.5 \cdot log_2(0.5) + -0.5 \cdot log_2(0.5) = 0.5 + 0.5 = 1$$

$$number\ of\ non\ college\ graduates = 2$$

$number\ of\ non\ college\ graduating\ conservatives = 1$

$ratio\!: non\ college\ graduate\ conservatives\ to\ non\ college\ graduates = \frac{1}{2} = 0.5$

$number\ of\ non\ college\ graduating\ liberals = 2$

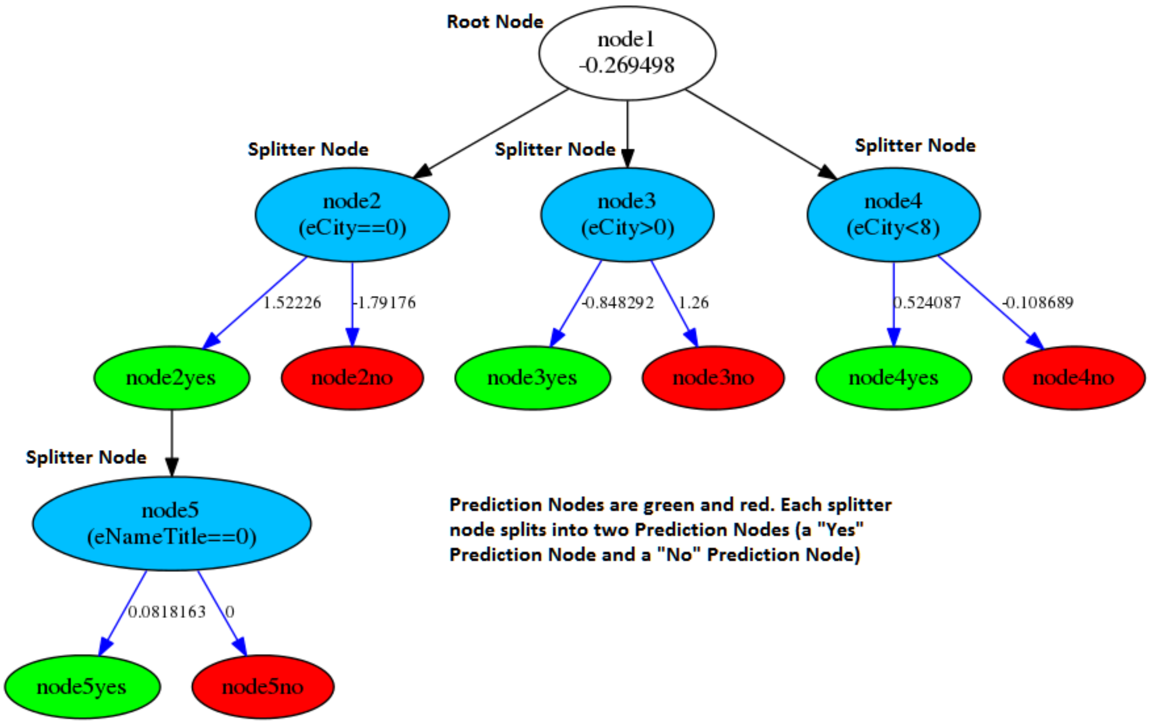
$$\begin{aligned} ratio\ of\ non\ college\ graduating\ liberals\ to\ all\ non\ college\ graduates &= \frac{1}{2} \\ &= 0.5 \cdot Entropy(S, !college) = -0.5 \cdot log_2(0.5) + -0.5 \cdot log_2(0.5) \\ &= 0.5 + 0.5 = 1 \end{aligned}$$

$Gain(S, college\ graduates) = 1 - (0.5 \cdot 1 + 0.5 \cdot 1) = 0$

The ID3 algorithm finds the “attends church” attribute to have the highest information gain and is therefore selected as the first node in its decision tree. This single node happens to identify the conservatives from the liberals perfectly. Because all of the records are classified correctly after just one node, the tree is deemed complete and recurring on either side of the “attends church” node is not necessary nor pursued. If even one record is not correctly identified, the algorithm recurs on either side of the newly added node using the remaining attributes as candidates.

### 3.3 Alternating Decision Tree

The Alternating Decision Tree (ADT) algorithm creates a top-down induction (TDI) tree used to solve binary classification problems. It is a supervised learning technique in the fact that it learns from training data (observations, measurements, etc.) which are accompanied by labels, or classifications, indicating the class of the observations. New data, or testing data, is classified based on the decision tree generated by the ADT algorithm after having consumed the training set. The ADT algorithm can be used for the task of record linkage (RL). ADT can group records in a database as record-pairs and classify these pairs as being the same (positive/duplicate) or different (negative/unique). The ADT algorithm creates a tree comprised of splitter nodes and prediction nodes. A splitter node specifies the condition that should be tested for the instance and splits into two prediction nodes. One of the two prediction nodes is consumed if the splitter node's condition is satisfied while the other is consumed if the splitter node's condition fails. A prediction node contains a real valued score which will be applied to the score of any instances which visit it. An ADT tree can be split multiple times at any point (it can attach more than one splitter node at any single prediction node). The ADT tree is not a binary tree. If the ADT algorithm calculates that the "best" condition should be added as a splitter node to a prediction node which already has two or more children, it is added nonetheless. An example is provided below where the root node has three children.



Prediction Nodes are green and red. Each splitter node splits into two Prediction Nodes (a "Yes" Prediction Node and a "No" Prediction Node)

**Figure 1: Labeled ADT Tree**

The ADT algorithm is a boosting algorithm. Boosting is a type of ensemble method for classification. There are multiple classifiers that work together in some fashion for the classification evaluation. Below is a diagram illustrating this process. The classifiers are represented as $M_x$ and are combined to make a prediction:

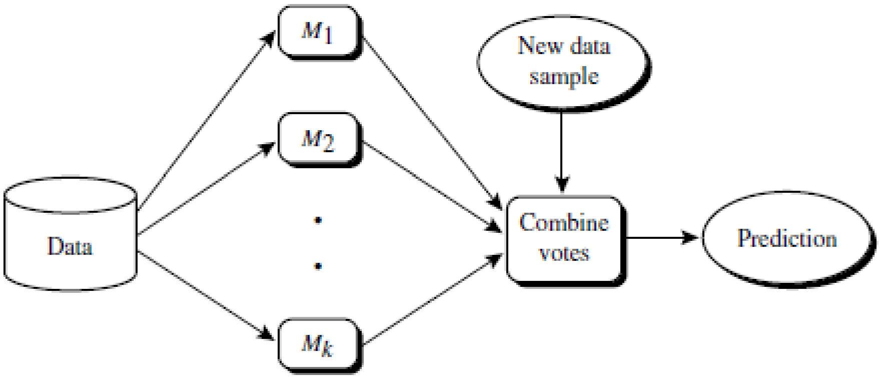


**Figure 2: ADT Classifiers**

The ADT algorithm uses boosting. In boosting, each instance in the training data is assigned a weight. A classifier builds and evaluates the training data. The instances that the algorithm classifies incorrectly receive an increased weight where conversely, the instances that the algorithm classifies correctly decrease in weight. Initially, the weights of all the instances sum to 1. As the algorithm progresses, the sum of the weights approaches 0. The following images demonstrate a visual overview of this concept.

Instance space of positives and negatives:

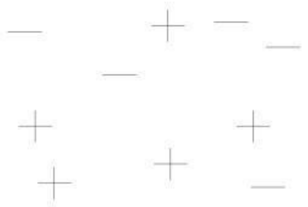


**Figure 3: Instance Space**

A line is drawn representing the initial classifier dividing the top half as a negative classification and the bottom half as a positive classification.

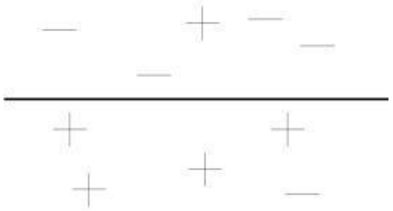


**Figure 4: Instance Space Separated**

The misclassified instances are assigned a larger weight and the positively classified examples are assigned a smaller weight.

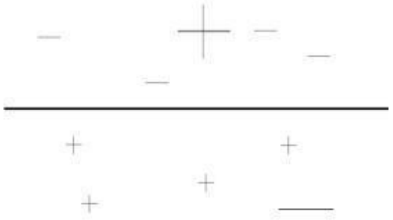


**Figure 5: Instance Space Weights**

The classification division becomes more complex and accurate to the training data as several classifiers are unified.

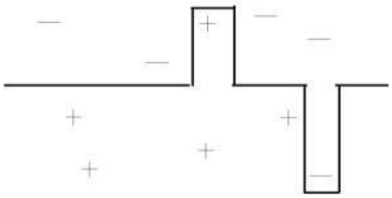


**Figure 6: Instance Space Division Complete**

The pseudo-code for the ADT algorithm is provided below.

```
$W_+(c) =$ sum of the weights of positively labeled examples that satisfy predicate $C$
$W_-(c) =$ sum of the weights of negatively labeled examples that satisfy predicate $C$
$W(c) = W_+(c) + W_-(c)$ the sum of weights of examples that satisfy predicate $C$
$W_i = \frac{1}{m} for\ all\ i$
$a = \frac{1}{2} ln \frac{W_+(true)}{W_-(true)}$
$R_0 = a\ rule\ with\ scores\ a\ and\ 0, precondition$ true" and condition true."
$R_0$ is the root node that  may be set using a heuristic
$R_0$ may be arbitrarily set to bias the tree toward positive or negative predictions
$P = \{true\}$
$C = \ the\ set\ of\ all\ possible\ conditions$
$z = 2\left(\sqrt{W_+(p \wedge c)W_-(p \wedge c)} + \sqrt{W_+(p \wedge \neg c)W_-(p \wedge \neg c)}\right) + \mathrm{W}(\neg p)$

$for\ j = 1 \dots T$
    $p \epsilon P, c \epsilon C\ get\ values\ that\ minimize\ z$
    $P += p \wedge c + p \wedge \neg c$
    $\boldsymbol{alpha1 = \frac{1}{2} ln \frac{W_+(p \wedge c) + 1}{W_-(p \wedge c) + 1}}$
    $\boldsymbol{alpha2 = \frac{1}{2} ln \frac{W_+(p \wedge \neg c) + 1}{W_-(p \wedge \neg c) + 1}}$
    $R_j = new\ rule\ with\ precondition\ p, condition\ c, \&\ weights\ alpha1\ and\ alpha2$
    $\boldsymbol{w_i = w_i e^{-y_i R_j x_i}}$
$return\ set\ of\ R_j$
```

The first task that the ADT algorithm pursues is determining the initial prediction node (the root node). Although any heuristic may be employed at this step. Typically, the first prediction adheres to the following:

$let\ p = sum\ of\ the\ weights\ of\ the\ positively\ labeled\ examples$

$let\ n = sum\ of\ the\ weights\ of\ the\ negatively\ labeled\ example$

$root\ node\ is\ positive\ if\ p > n;\ root\ node\ is\ negative\ if\ n > p$

This is the initial bias it has towards the data. For example, if a strong negative score is given to this root node, the ADT starts out assuming that the test data is likely to have a negative classification. There exists an inverse relationship between false positives and false negatives. Often times this initial node is manipulated to bias the ADT predictions toward either false positives (a positive root node) or false negatives (a negative root node).

After the root node is determined, the process of selecting the “best” conditions and adding them to the tree as splitter nodes begins. In order to select the “best” condition, a z-score for every precondition/condition pair is computed.

$Suppose\ preconditions\ [a, b, c, d]\ (available\ conditions)$

$Suppose\ conditions\ [m, n, o]\ (conditions\ that\ have\ been\ previously\ used\ in\ the\ tree)$

$z - scores\ of\ node\ pairs[am, an, ao, bm, bn, bo, cm, cn, co, dm, dn, do]$

The z-score is minimized when the pair provides the lowest amount of entropy in the system. It is sensitive to the attributes and weights of the instances in the training data set. The next splitter node added to the tree will contain condition in the precondition/condition pair whose z-score is lowest.

Once the “best” condition is determined, the splitter node is created containing the “best” condition. Before the splitter node can be added to the tree, two scores need to be computed which will ultimately be added as prediction nodes on either side of splitter node. The “yes” prediction node’s score is the weighted sum of the positively labeled examples divided by the weighted sum of the positively labeled examples that satisfy the precondition/condition pair. Likewise, for the “no” prediction node, it is the same calculation but for examples that do not satisfy the precondition/condition pair.

At this point, the splitter node and its two prediction nodes are attached to the ADT tree. Next, the newly formed tree is used to evaluate the training data. If an instance is misclassified, its weight increases. If an instance is classified correctly, its weight decreases. Finally, the process of selecting the next “best” condition (to be added in a splitter node) is repeated.

The algorithm does not automatically decide how many nodes the ADT tree will ultimately have. It is an option that the user specifies. Alternately, a heuristic may be employed in order to accomplish this task.

Upon the completion of the ADT tree, the training data is used to evaluate the accuracy of the classification. An input instance is fed to the tree (traversing every branch). At each prediction node, the prediction node’s score is aggregated to an overall score. Once the entire ADT tree traversal is complete, the aggregate score is analyzed. If the score is positive, the classification is positive. Conversely, if score is negative, the classification is negative. The degree of confidence on a classification is directly proportional to the distance its score is from zero. In other words, the more positive a score, the more likely an instance is classified correctly as a positive. The more negative the score, the more likely an instance is classified correctly as a negative. Knowledge contained in the tree is distributed as multiple paths must be traversed to form predictions. Instances that satisfy multiple splitter nodes have the values of these prediction nodes that they reach summed to form an overall prediction value. A positive sum represents one class and a negative sum represents the other in the two-class setting (Holmes 2002).

### 3.4 Cost Sensitive Alternating Decision Tree

The CSADT algorithm is an upgrade from the classic ADT algorithm. It augments ADT by including two cost-sensitive variables: $C_+$ and $C_-$. Cost-sensitivity adds a cost value is assigned to the false positives and false negatives. The higher the $C_+$, the more weight the algorithm places on false positives. The higher the $C_-$, the more weight the algorithm places on false negatives. The fundamental assumption is that the instances whose weights are higher are increased faster than those with lower weights. The learning focus of CSADT tree will be biased towards instances with higher weights.

CSADT implements cost-sensitivity by modifying how the alpha1 and alpha2 scores are computed as well as how the weights are updated on every iteration. The following table shows the difference in calculation for alpha1 and alpha2 between the ADT and CSADT algorithms.

**Table 6: ADT alphas vs. CSADT alphas**

| ADT | CSADT |
|---|---|
| $alpha1 = \frac{1}{2} ln \frac{W_+(p \wedge c) + 1}{W_-(p \wedge c) + 1}$ | $alpha1 = \frac{1}{2} ln \frac{C_+ \cdot W_+(p \wedge c) + 1}{C_- \cdot W_-(p \wedge c) + 1}$ |
| $alpha2 = \frac{1}{2} ln \frac{W_+(p \wedge \neg c) + 1}{W_-(p \wedge \neg c) + 1}$ | $alpha2 = \frac{1}{2} ln \frac{C_+ \cdot W_+(p \wedge \neg c) + 1}{C_- \cdot W_-(p \wedge \neg c) + 1}$ |

The following table shows the difference in how the ADT and CSADT algorithms update weights.

**Table 7: ADT weights vs. CSADT weights**

| | ADT | CSADT |
|---|---|---|
| if labeled example is positive | $w_i = w_i e^{-y_i R_j x_i}$ | $w_i = C_+ \cdot w_i e^{-y_i R_j x_i}$ |
| if labeled example is negative | | $w_i = C_- \cdot w_i e^{-y_i R_j x_i}$ |

### 3.4.1 CSADT Advantages

Without cost-sensitive training, the false positives and false negative have equal weights. Weighing false positives differently than false negatives has positive implications in industry. Identifying false positives in business-related activities can be detrimental. For example, a bank positively identifies a person as being a good candidate for a loan, although this person has poor money management. Clearly the bank would be making a poor investment in that person. Another example is positively identifying a person as a criminal when in fact they are not. With the ability to weigh false positives differently than false negatives, delicate real-world situations such as these could be better managed.

Due to its boosting approach (maintaining a list of weights for every training instance), the CSADT algorithm is advantageous, since it counteracts the negative effect of training data that contains an unequal distribution of classes.

CSADT offers other advantages when compared to alternatives as a solution for an RL problem. It is easy to interpret how false positives are weighted and how the tree decides on a classification. Not only is a classification determined, but also a prediction score stating how certain that classification is determined.

Clustering-based algorithms would perform poorly for record linkage. If non-consequential attributes cause the record-pairs to be equal-distant, the distance between same and different records would be the same. CSADT can use authority data (training data) to discover which attributes are similar among duplicates. It can ignore attributes that are non-consequential in determining the same identity.

The table below (Chen 2011) outlines differences between the following algorithms: Decision Trees, Support Vector Machines, Boosted Decision Tree, Alternating Decision Tree, and Cost-Sensitive Alternating Decision Tree.

**Table 8: ML Algorithm Comparison**

| Algorithm | Human Interpretability | Cost Sensitivity | Captures Feature Interactions | Handles Diverse Feature Types |
|---|---|---|---|---|
| Decision Tree | Yes | No | Yes | Yes |
| Support Vector Machine | No | With Custom Kernel | With Custom Kernel | Not Easily |
| Boosted Decision Tree | No | No | Yes | Yes |
| Alternating Decision Tree | Yes | No | Yes | Yes |
| Cost-Sensitive Alternating Decision Tree | Yes | Yes | Yes | Yes |

# 4. PERSON DATA-DRIVEN CSADT REQUIREMENTS

## 4.1 Person Data

The Department of Geography at Texas State University provided a dataset containing several hundred thousand person records from public sources (Zabasearch.com, Whitepages.com, Addresses.com) accumulated over several years. The list below contains the attributes parsed from data gathered.

1. Full name
2. First name
3. Last name
4. Middle name
5. Age
6. Full address
7. Apartment
8. City
9. State
10. Zip code
11. Phone
12. Date
13. Date of birth
14. Relatives

The person dataset was organized into 14 attributes above. In addition to the organization, a series of tasks were employed to filter unwanted records (such as businesses) and well as unwanted text in a person record’s name such as name suffixes, name titles, and arbitrary spaces or punctuation.

Most of the attributes contained in the dataset are strings (simple text). These text attributes require a method to determine the difference between them. Such a method needs to determine that Jon and John are similar while Jon and Martha are not. The method found to deliver a distance corresponding to strings’ differences would compute the Levenshtein distance.

### 4.2 Levenshtein Distance

The Levenshtein distance is used to determine the difference between two strings. A string is any finite sequence of characters (i.e., letters, numerals, symbols and punctuation marks). The Levenshtein distance is defined as the minimum number of modifications to transform one string into another (Levenshtein 1999). A modification can be one of the following:

- An Insertion
- A Deletion
- A Substitution

Below is a table illustrating how the Levenshtein Distance is determined. Every cell that contains a number represents the Levenshtein Distance between the [sub] strings corresponding to that cell. The bottom-right most cell represents the Levenshtein Distance between the complete strings.

The cell at the intersection of the K (in CLARKE) and the C (in CLERK) is highlighted in red and contains the number four. The four is the Levenshtein distance between CLARK and C (CLARK - ‘L’, ‘A’, ‘R’, and ‘K’ = C).

The cell at the intersection of the A (in CLARKE) and the E (in CLERK) is a highlighted in green and contains the number one. The one is the Levenshtein distance between CLA and CLE (just one replacement; the A for the E).

The cell at the intersection of the E (in CLARKE) and the K (in CLERK) is highlighted in orange and contains the number two. The two is the Levenshtein distance between CLARKE and CLERK (one replacement; the A for the E and one deletion; the E).

**Table 9: Levenshtein Distance Example**

|  |  | C | L | A | R | K | E |
|---|---|---|---|---|---|---|---|
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| C | 1 | 0 | 1 | 2 | 3 | 4 | 5 |
| L | 2 | 1 | 0 | 1 | 2 | 3 | 4 |
| E | 3 | 2 | 1 | 1 | 2 | 3 | 4 |
| R | 4 | 3 | 2 | 2 | 1 | 2 | 3 |
| K | 5 | 4 | 3 | 3 | 2 | 1 | 2 |

The Levenshtein Distance was used to determine the difference between two records' attributes. For example, the following are the full names (full name is an attribute) from two different records:

1. Charles Tu
2. Charles C Tu

The Levenshtein Distance between the full names above is 2 since the minimum number of modifications is two. In order to transform the first full name into the other, two insertions are required:

1. Insert a space
2. Insert the character “C”

### 4.3 Record Comparisons

Each individual record was compared to all other records. Therefore, the number of unique record combinations must be computed. The equations and examples below explain and demonstrate how the number of unique record-pairs are computed:

$Unique\ records\ =\ n$

$Number\ chosen\ =\ r$

$$Number\ of\ Record\ Pairs\ =\ \frac{n!}{(n-r)!\cdot r!}$$

An example is offered to demonstrate the equation above. Suppose there are four unique records (a, b, c, and d) and we are only interested in pair-wise comparisons.

$n = 4\ (a, b, c, d)$

$r\ =\ 2\ (two\ to\ a\ pair)$

$$Number\ of\ Record\ Pairs\ = \frac{4!}{(4-2)!\cdot 2!} = \frac{4!}{2!\cdot 2!} = \frac{4\cdot 3\cdot 2\cdot 1}{2\cdot 1\cdot 2\cdot 1} = \frac{24}{4} = 6$$

The unique record-pairs are listed below (which confirms that the computation is correct).

1. (a, b)
2. (a, c)
3. (a, d)

4. (b, c)

5. (b, d)

6. (c, d)

The graph below illustrates the record-pairs exponential growth as the number of records increases.

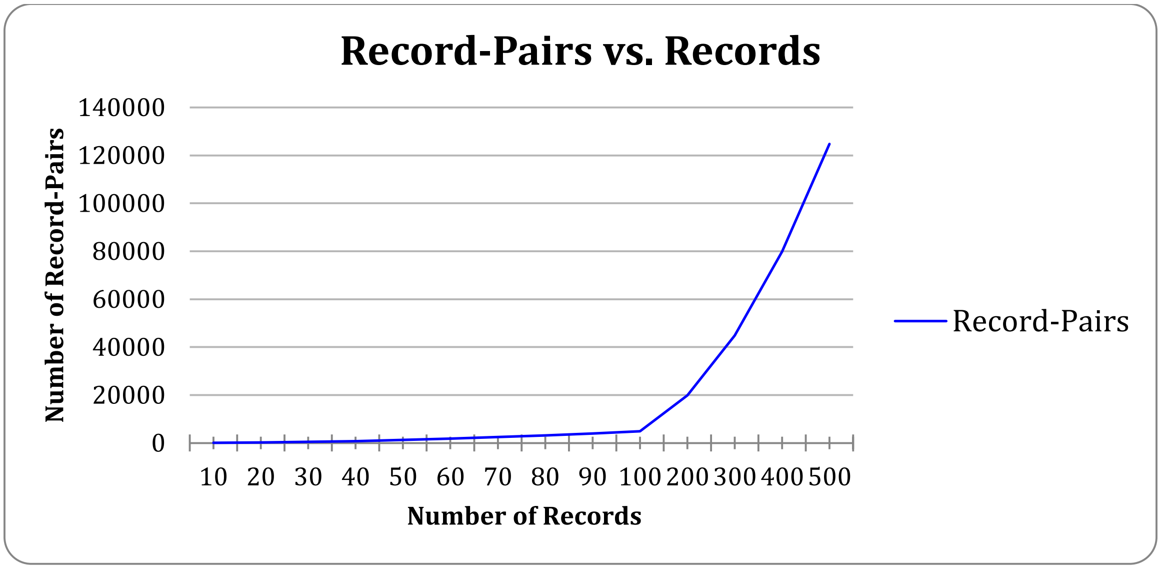


**Figure 7: Record-Pairs' Growth**

When a record is compared to another record, every attribute of each record is compared. Assuming these attribute comparisons require a Levenshtein Distance, record comparisons are computationally expensive.

$Suppose\ a\ =\ number\ of\ attributes$

$Suppose\ c\ =\ number\ of\ record\ pairs$

$Suppose\ n\ =\ number\ of\ conditions$

$Suppose\ I\ =\ number\ of\ iterations$

The tree's time complexity is given by $O(a\ \cdot\ c\ \cdot\ n\ \cdot\ i)$. Since the CSADT algorithm updates condition weights, the space complexity given by $O(c)$.

For example, suppose a dataset contained 50 records (1,225 record-pairs) 10 attributes, 50 conditions, and the CSADT algorithm was set to create a tree with 12 nodes.

$$1{,}225 \cdot 10 \cdot 50 \cdot 12 = 7{,}350{,}000\ comparisons$$

### 4.4 Intelligent Condition Generation

One significant advantage of using the CSADT algorithm over another supervised learning technique is that the CSADT algorithm will consume as many conditions as it is offered, sift through them and select the “best” conditions. Therefore, the number of useful conditions offered to the CSADT has a direct impact on its accuracy. The user, who is running the algorithm, can always provide as many conditions as they like in an effort to maximize the algorithm’s accuracy. However, it would be ideal if well-chosen conditions were offered as options regardless of what the user supplies.

The automatic intelligent condition generation involved determining the average Levenshtein distance for each match attribute and each non-match attribute for all record-pairs. Conditions were added corresponding to these averages. The standard deviation (the amount of variation or dispersion of a set of data) was then computed amongst match and non-match record-pairs. Conditions were added for each attribute average (for matches and non-matches) after adding/subtracting one standard deviation.

For example, suppose that the average for matches of the first name attribute was 0.5 (matching records had an average Levenshtein distance of 0.5) and the standard deviation was 1. Suppose that the average for non-matches of the first name attribute was 12 (non-matching records had an average Levenshtein distance of 12) and the standard deviation was 4. The following “intelligent” conditions would have been added automatically:

1. First name <, =, > 0.5 (First name <, =, > match average)
2. First name <, =, > 1.5 (First name <, = , > match average + match standard deviation)
3. First name <, =, > *-0.5* (First name <, =, > match average - match standard deviation)
    a. Since the standard deviation subtracted from the match average is negative, this condition would not be added as a Levenshtein distance can never be negative. Two strings that are identical have a Levenshtein distance of zero, meaning that zero insertions, deletions, and replacements are required in order to transform one string into the other. Nevertheless, this condition is listed only as an example to illustrate the intelligent condition generation procedure.
4. First name <, =, > 12 (First name <, =, >  non-match average)
5. First name <, =, > 16 (First name <, =, > non-match average + non-match standard deviation)
6. First name <, =, > 8 (First name <, =, > non-match average - non-match standard deviation)

Although the intelligent condition generation was only implemented to use the Levenshtein distance average, it is not limited by this. If a quantitative distance is computed between two attributes, the intelligent condition generation will compute the average and the standard deviation for that attribute and create conditions based on those results.

# 5. CSADT IMPLEMENTATION

## 5.1 Program Outline

The CSADT algorithm described in section 3 was first implemented in Python. The Python implementation required more than 10 minutes to process only 100 person records (4,950 person record-pairs). This performance was determined insufficient. The computation-heavy sections of the algorithm were re-implemented in C++ in order to increase their performance. The table below outlines which sections were implemented by which language and a brief description is given.

**Table 10: CSADT Language Implementation**

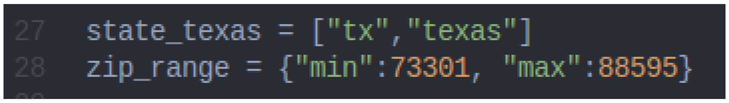
| Python | C++ | Description |
|---|---|---|
| Input Organizing/Cleaning | | Organize/Clean raw input |
| Input Conversion<br>*Output: Training.txt* | | Convert cleaned input into \| delimited file |
| Compare Record-Pairs<br>*Input: Training.txt* | Generate Unique Pair IDs<br>*Input: Training.txt* | ● Create unique record-pairs<br>● Compute the Levenshtein distances of all record-pairs' attributes |
| | Compare Record-Pairs | |
| Create Tree<br>*Output: CSADT.txt* | Create Tree<br>*Output: CSADT.txt* | Run the CSADT algorithm (Delivers a decision tree) |
| Illustrate Tree<br>*Input: CSADT.txt* | | Create an illustrated graph of the tree (nodes and edges) |
| Classification Accuracy<br>*Input: CSADT.txt* | | Compute the accuracy of the decision tree |
| Classify Testing<br>*Input: CSADT.txt* | | Run testing data through the decision tree |

### 5.1.1 CSADT Python

#### 5.1.1.1 Input Organizing/Cleaning

The input data contained Texan person records scraped from the Internet in 2009, 2010, and 2012. This input cleaning task organized this information into records (containing specific attributes).

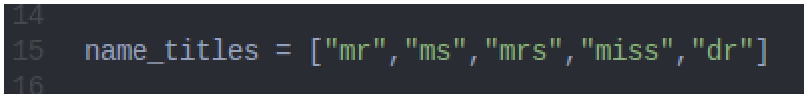
In order to ensure that the person records were representative or individuals living in Texas, the addresses of the records were parsed and analyzed for their state and zip code.



```
state_texas = ["tx","texas"]
zip_range = {"min":73301, "max":88595}
```

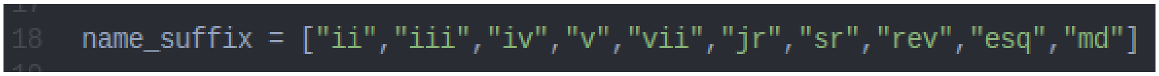
**Figure 8: Texas Filter**

This task was also responsible for filtering out text that was not unique to that individual. For example, name titles, name suffixes, and arbitrary spaces were all filtered out of the original person records. The snippets below are examples of text that was filtered out of person records.



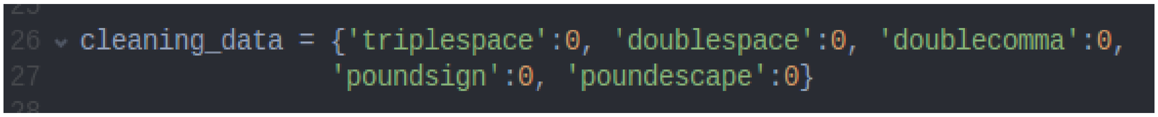
```
name_titles = ["mr","ms","mrs","miss","dr"]
```

**Figure 9: Filter Name Titles**



```
name_suffix = ["ii","iii","iv","v","vii","jr","sr","rev","esq","md"]
```

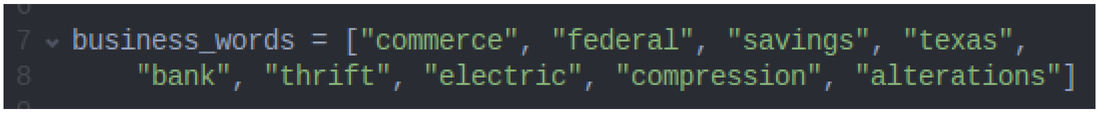
**Figure 10: Filter Name Suffixes**



```
cleaning_data = {'triplespace':0, 'doublespace':0, 'doublecomma':0,
                 'poundsign':0, 'poundescape':0}
```

**Figure 11: Input Cleaning Symbols**

The original contained many “bad” records such as businesses instead of people. If a record contained any of the following business words, it was dismissed.



```
business_words = ["commerce", "federal", "savings", "texas",
    "bank", "thrift", "electric", "compression", "alterations"]
```

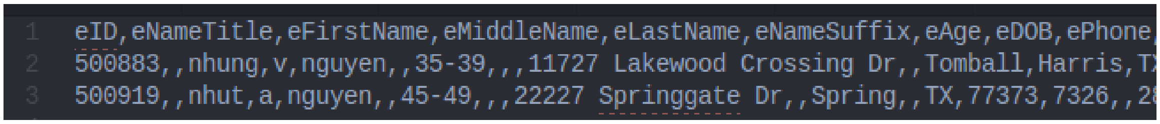
**Figure 12: Input Cleaning Words**

Many records contained unnecessary characters. These unnecessary characters were parsed out in an effort to deliver clean person records.

### 5.1.1.2 Input Conversion

Input data is received in either a .csv file or a .xlsx file. The input conversion step simply converts either input file type into a pipe, or |, delimited text file named Training.txt. It also creates an HTML file containing the input data in a table for easy viewing.

Below is a snippet from an example of a converted input file. The first line contains the attribute names while all other lines are legitimate training data corresponding to their attributes.



```
eID,eNameTitle,eFirstName,eMiddleName,eLastName,eNameSuffix,eAge,eDOB,ePhone,
500883,,nhung,v,nguyen,,35-39,,,11727 Lakewood Crossing Dr,,Tomball,Harris,T
500919,,nhut,a,nguyen,,45-49,,,22227 Springgate Dr,,Spring,,TX,77373,7326,,2
```

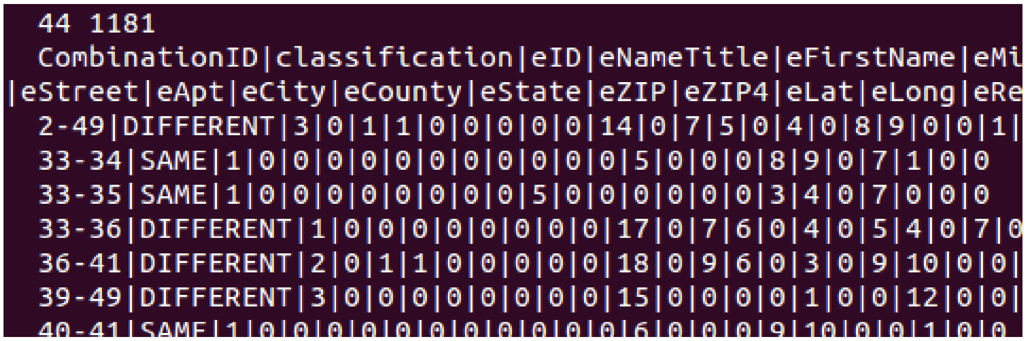
**Figure 13: Person Record**

### 5.1.1.3 Compare Record-Pairs

The decision of which records will be compared to one another is made in this step. The result of this decision is a list of record-pairs. Once the record-pairs are determined, the Levenshtein distances of selected attributes of every record-pair is calculated and saved into a file: ComparisonRecords.txt.

As demonstrated from the snippet from an example of ComparisonRecords.txt below, the differences between attributes are simple integers. The first line contains the number of matches in the dataset (23) and the number of non-matches (412). The next line contains the names of the attributes. The remaining lines contain:

1. record-pair combination ID
    - 22-29 represents record 22 being compared to record 29
2. classification (SAME or DIFFERENT)
3. Levenshtein distances of all attributes



```
  44 1181
  CombinationID|classification|eID|eNameTitle|eFirstName|eMi
|eStreet|eApt|eCity|eCounty|eState|eZIP|eZIP4|eLat|eLong|eRe
  2-49|DIFFERENT|3|0|1|1|0|0|0|0|0|14|0|7|5|0|4|0|8|9|0|0|1|
  33-34|SAME|1|0|0|0|0|0|0|0|0|0|0|0|5|0|0|0|8|9|0|7|1|0|0
  33-35|SAME|1|0|0|0|0|0|0|0|0|5|0|0|0|0|0|0|3|4|0|7|0|0|0
  33-36|DIFFERENT|1|0|0|0|0|0|0|0|0|17|0|7|6|0|4|0|5|4|0|7|0
  36-41|DIFFERENT|2|0|1|1|0|0|0|0|0|18|0|9|6|0|3|0|9|10|0|0|
  39-49|DIFFERENT|3|0|0|0|0|0|0|0|0|15|0|0|0|0|1|0|0|12|0|0|
  40-41|SAME|1|0|0|0|0|0|0|0|0|0|0|0|6|0|0|0|9|10|0|0|1|0|0
```

**Figure 14: Combinations Example**

#### 5.1.1.4 Create Tree

The CSADT algorithm has already been discussed in depth in the “Cost Sensitive Alternating Decision Tree.” Below is a diagram indicating how it was implemented in Python:

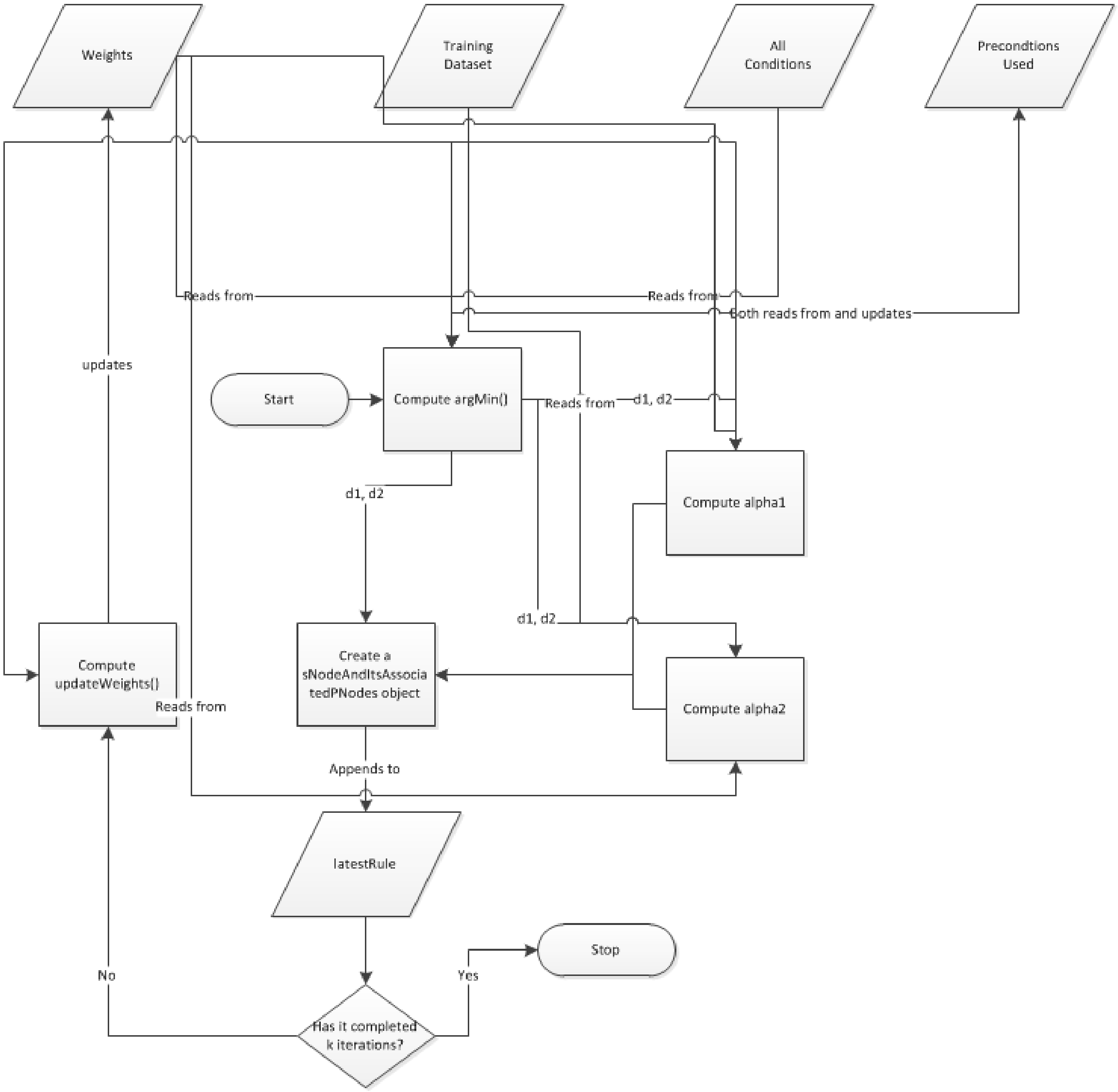
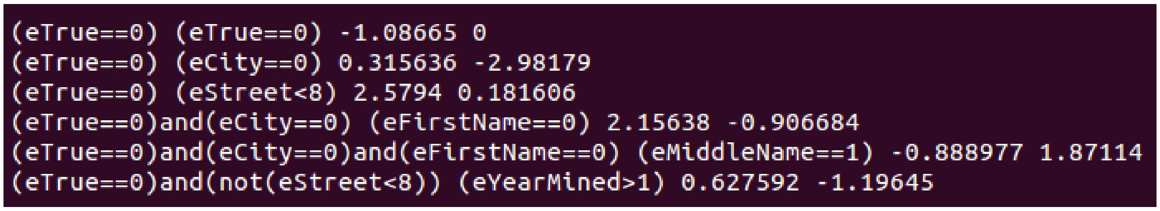


**Figure 15: ADT Flowchart**

Below is an example a four node Tree.txt generated by the Python implementation of the CSADT algorithm.



```
(eTrue==0) (eTrue==0) -1.08665 0
(eTrue==0) (eCity==0) 0.315636 -2.98179
(eTrue==0) (eStreet<8) 2.5794 0.181606
(eTrue==0)and(eCity==0) (eFirstName==0) 2.15638 -0.906684
(eTrue==0)and(eCity==0)and(eFirstName==0) (eMiddleName==1) -0.888977 1.87114
(eTrue==0)and(not(eStreet<8)) (eYearMined>1) 0.627592 -1.19645
```

**Figure 16: ADT Tree (text)**

### 5.1.1.5 Illustrate Tree

In order to better understand the Tree.txt (displayed in the previous section), a tree illustrator was developed. Below is the decision tree generated by the cited Tree.txt.

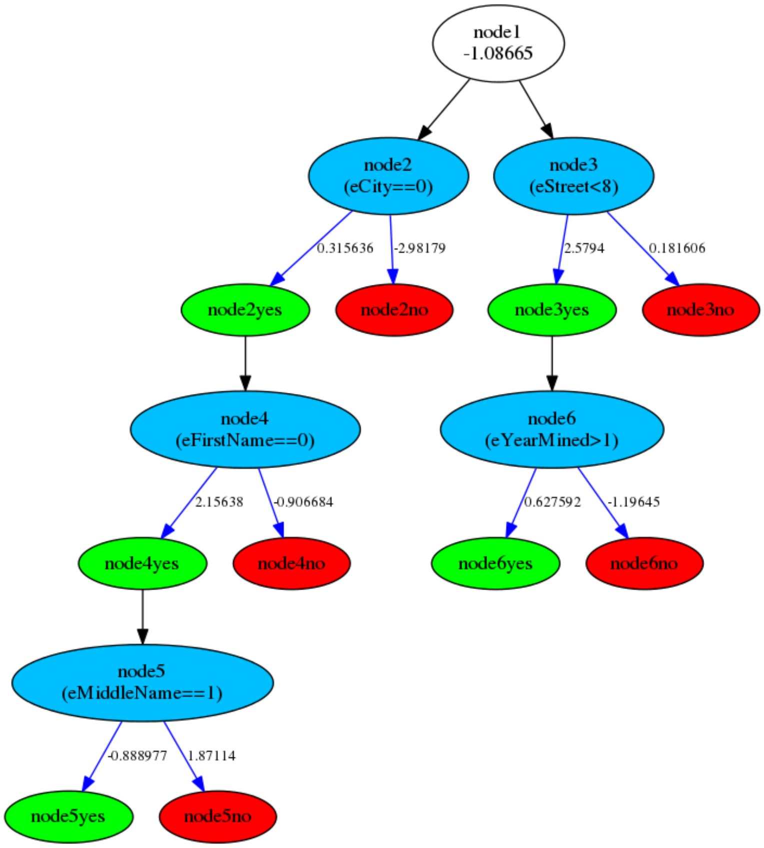
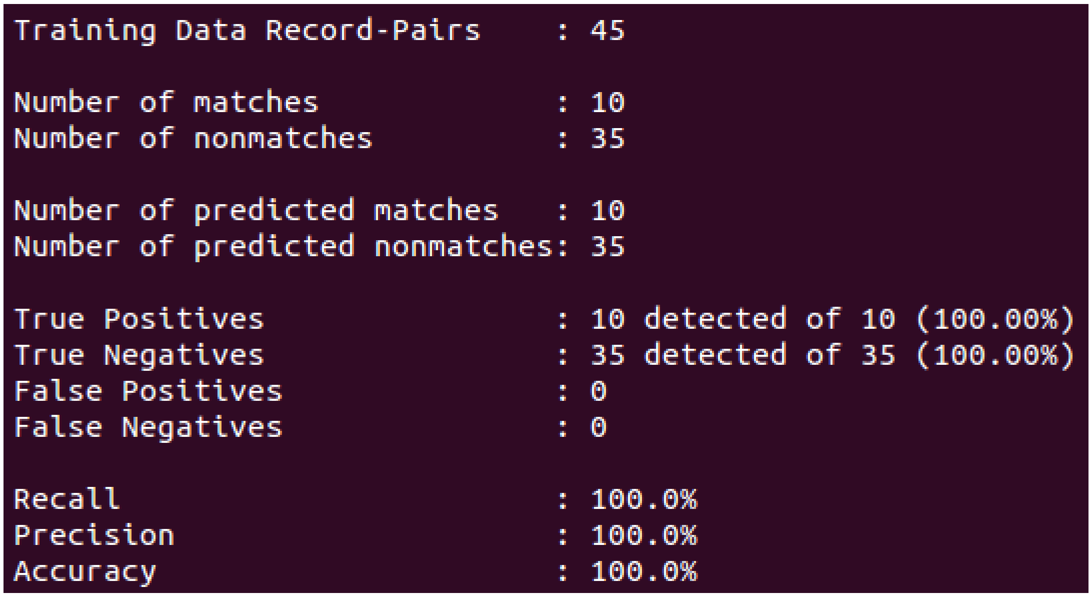


**Figure 17: ADT Tree**

#### 5.1.1.6 Classification Accuracy

After a CSADT tree is created, training data must be run through the CSADT tree in order to compute the tree's effectiveness at prediction classifications correctly. In this small dataset of only 45 record-pairs, the decision tree was perfect in its prediction classifications:



```
Training Data Record-Pairs    : 45

Number of matches             : 10
Number of nonmatches          : 35

Number of predicted matches   : 10
Number of predicted nonmatches: 35

True Positives                : 10 detected of 10 (100.00%)
True Negatives                : 35 detected of 35 (100.00%)
False Positives               : 0
False Negatives               : 0

Recall                        : 100.0%
Precision                     : 100.0%
Accuracy                      : 100.0%
```

**Figure 18: Classification Accuracy**

#### 5.1.1.7 Classify Testing Data

This last step enables the CSADT algorithm to make prediction classifications on real-world testing data.

### 5.1.2 CSADT C++

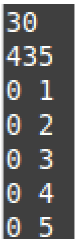
The C++ implementation of the CSADT algorithm begins by consuming the Training.txt file generated in Python "input conversion" step. The training data is parsed into unique record-pairs after which their attributes are compared. Finally a CSADT tree is generated. At this point, the C++ implementation is completed. Therefore, all preprocessing, the illustration of the tree, the accuracy computed, and the classification of testing data are all still completed by their Python implementations.

The Python “Compare Record-Pairs” step was divided into two steps and they are explained below:

- Generate Unique Pair IDs
- Compare Record-Pairs

#### 5.1.2.1 Generate Unique Pair IDs

This step’s sole task is to determine unique record-pair combinations. Below is a snippet of the output from this step.



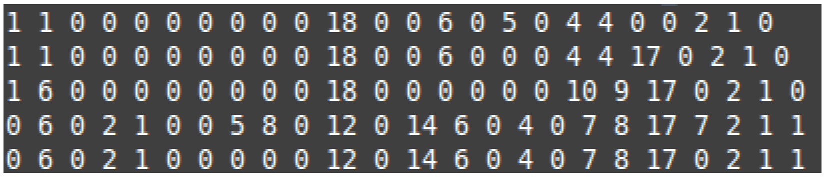
```
30
435
0 1
0 2
0 3
0 4
0 5
```

**Figure 19: Unique Pairs**

The first line contains the number of records: 30. The second line contains the number of combinations: 435. The following lines represent record-pair combinations the first of which is “0 1”. The “0 1” represents record 0 and record 1 as a record-pair.

#### 5.1.2.2 Compare Record-Pairs

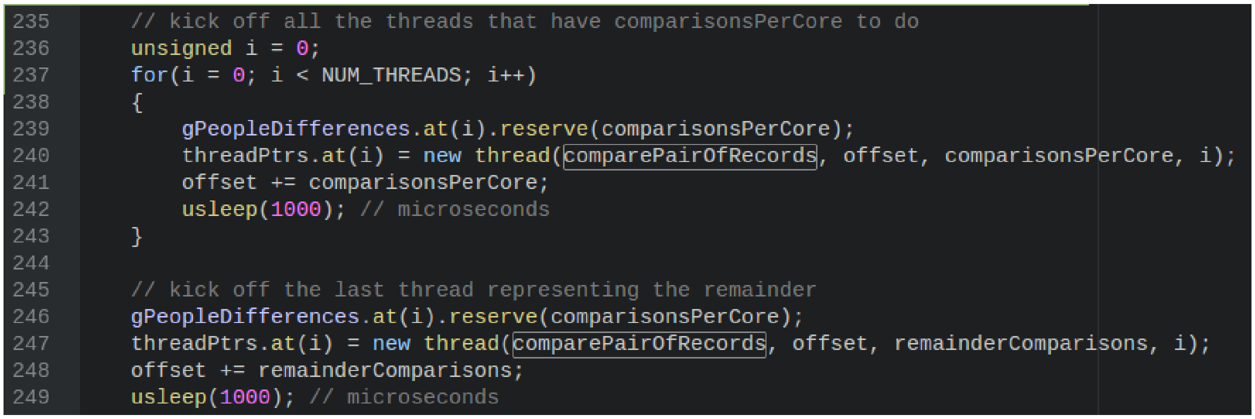
This step receives the record-pairs just determined as input and computes the Levenshtein distances of the attributes of those records. Below is a snippet containing the Levenshtein distances of the attributes of the record-pairs: 0-1, 0-2, 0-3, 0-4, and 0-5.



```
1 1 0 0 0 0 0 0 0 0 18 0 0 6 0 5 0 4 4 0 0 2 1 0
1 1 0 0 0 0 0 0 0 0 18 0 0 6 0 0 0 4 4 17 0 2 1 0
1 6 0 0 0 0 0 0 0 0 18 0 0 0 0 0 0 10 9 17 0 2 1 0
0 6 0 2 1 0 0 5 8 0 12 0 14 6 0 4 0 7 8 17 7 2 1 1
0 6 0 2 1 0 0 0 0 0 12 0 14 6 0 4 0 7 8 17 0 2 1 1
```

**Figure 20: Input Levenshtein Distances**

The C++ code uses threads to improve performance. Below is the code responsible for invoking the threads which compute the Levenshtein distances of the record-pairs.

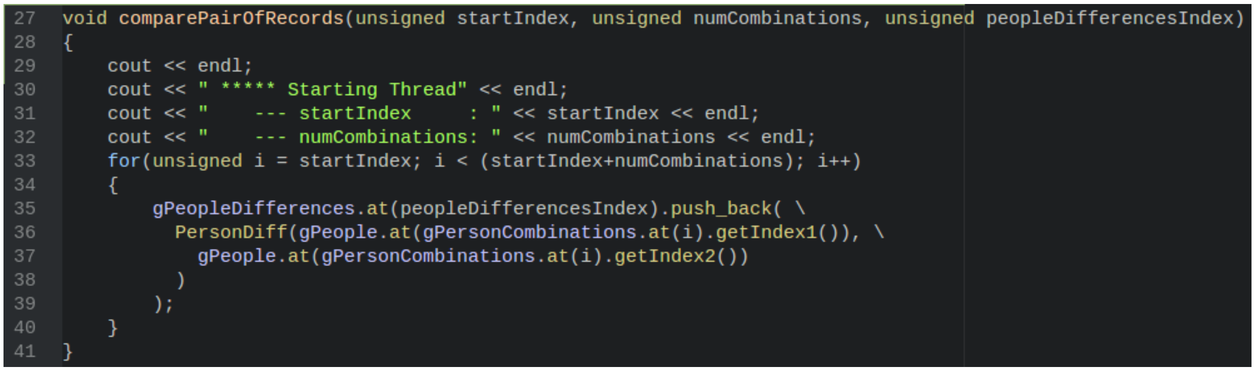


```cpp
    // kick off all the threads that have comparisonsPerCore to do
    unsigned i = 0;
    for(i = 0; i < NUM_THREADS; i++)
    {
        gPeopleDifferences.at(i).reserve(comparisonsPerCore);
        threadPtrs.at(i) = new thread(comparePairOfRecords, offset, comparisonsPerCore, i);
        offset += comparisonsPerCore;
        usleep(1000); // microseconds
    }

    // kick off the last thread representing the remainder
    gPeopleDifferences.at(i).reserve(comparisonsPerCore);
    threadPtrs.at(i) = new thread(comparePairOfRecords, offset, remainderComparisons, i);
    offset += remainderComparisons;
    usleep(1000); // microseconds
```

**Figure 21: Levenshtein Distance Threads**

Below is the function that each thread performs:

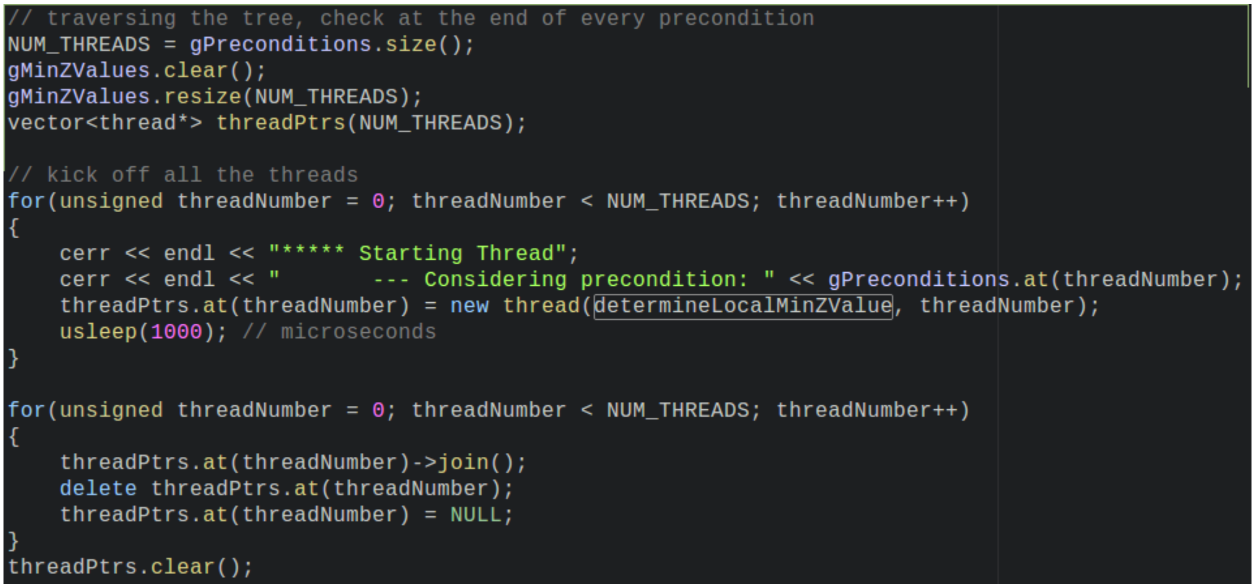


```
void comparePairOfRecords(unsigned startIndex, unsigned numCombinations, unsigned peopleDifferencesIndex)
{
    cout << endl;
    cout << " ***** Starting Thread" << endl;
    cout << "    --- startIndex     : " << startIndex << endl;
    cout << "    --- numCombinations: " << numCombinations << endl;
    for(unsigned i = startIndex; i < (startIndex+numCombinations); i++)
    {
        gPeopleDifferences.at(peopleDifferencesIndex).push_back( \
          PersonDiff(gPeople.at(gPersonCombinations.at(i).getIndex1()), \
            gPeople.at(gPersonCombinations.at(i).getIndex2())
          )
        );
    }
}
```

**Figure 22: Levenshtein Distance Thread Function**

### 5.1.2.3 Create Tree

The CSADT C++ version invokes a thread on every precondition when searching for the best condition among the remaining conditions. This dramatically decreases the computation time.



```
// traversing the tree, check at the end of every precondition
NUM_THREADS = gPreconditions.size();
gMinZValues.clear();
gMinZValues.resize(NUM_THREADS);
vector<thread*> threadPtrs(NUM_THREADS);

// kick off all the threads
for(unsigned threadNumber = 0; threadNumber < NUM_THREADS; threadNumber++)
{
    cerr << endl << "***** Starting Thread";
    cerr << endl << "       --- Considering precondition: " << gPreconditions.at(threadNumber);
    threadPtrs.at(threadNumber) = new thread(determineLocalMinZValue, threadNumber);
    usleep(1000); // microseconds
}

for(unsigned threadNumber = 0; threadNumber < NUM_THREADS; threadNumber++)
{
    threadPtrs.at(threadNumber)->join();
    delete threadPtrs.at(threadNumber);
    threadPtrs.at(threadNumber) = NULL;
}
threadPtrs.clear();
```

**Figure 23: C++ Pre-Condition Threads**

# 6. EVALUATION

## 6.1 Person Data Computation Time Analysis

The evaluation of the CSADT algorithm on people records includes an analysis of the computation time for creating the CSADT tree model and the accuracy of the predictions made based on that model. All other tasks required an insignificant amount of computation time when compared to the creation of the CSADT tree.

The following variables were studied in order to understand their effect on the computation time required in the creation of a CSADT tree.

- Number of conditions considered by the CSADT algorithm
- Number of nodes in the CSADT tree
- Number of records consumed by the CSADT algorithm

All experiments were completed on a desktop computer with the following specifications:

- 8-Core 3.6GHz AMD FX-8150 Zambezi Socket AM3+ 125W Processor
- 32GB of RAM
- 256GB SSD

Below are three tables which show that the creation of the CSADT tree consumed the most computation time compared to the other tasks. This is independent of which implementation was used (C++ Threaded, C++ Non-Threaded, or Python).

**Table 11: C++ Threaded Tasks**

| Number of Nodes | 10 | **C++ Threaded** | | | |
|---|---|---|---|---|---|
| Number of Records | 100 | | | | |
| **Number of Conditions** | **25** | **50** | **100** | **125** | **150** |
| **Generate Unique Pair IDs** | 0.04 s | 0.04 s | 0.03s | 0.04 s | 0.05 s |
| **Compare Record-Pairs** | 0.38 s | 0.41 s | 0.4 s | 0.37 s | 0.41 s |
| **Create Tree** | **9.69 s** | **15.01 s** | **30.07 s** | **36.92 s** | **45.24 s** |
| **Translate Tree** | 0.02 s | 0.02 s | 0.02 s | 0.02 s | 0.02 s |
| **Illustrate Tree** | 0.24 s | 0.2 s | 0.18 s | 0.19 s | 0.2 s |
| **Classification Accuracy** | 1.4 s | 1.5 s | 1.45 s | 1.52 s | 1.42 s |

**Table 12: C++ Non-Threaded Tasks**

| Number of Conditions | 169 | **C++ Non-Threaded** | | | |
|---|---|---|---|---|---|
| Number of Records | 100 | | | | |
| **Number of Nodes** | **5** | **7** | **10** | **12** | **15** |
| **Generate Unique Pair IDs** | 0.04 s | 0.03 s | 0.04 s | 0.03 s | 0.04 s |
| **Compare Record-Pairs** | 0.38 s | 0.41 s | 0.4 s | 0.44 s | 0.41 s |
| **Create Tree** | **62.47 s** | **120.62 s** | **232.29 s** | **320.08 s** | **496.66 s** |
| **Translate Tree** | 0.02 s | 0.03 s | 0.02 s | 0.02 s | 0.02 s |
| **Illustrate Tree** | 0.14 s | 0.16 s | 0.2 s | 0.23 s | 0.23 s |
| **Classification Accuracy** | 0.83 s | 1.1 s | 1.64 s | 1.8 s | 2.2 s |

**Table 13: Python Tasks**

| Number of Conditions | 20 | **Python** | | | |
|---|---|---|---|---|---|
| Number of Nodes | 5 | | | | |
| **Number of Records** | **10** | **20** | **30** | **40** | **50** |
| **Input Conversion** | 0.02 s | 0.03 s | 0.02 s | 0.02 s | 0.02 s |
| **Compare Record-Pairs** | 0.05 s | 0.15 s | 0.32 s | 0.55 s | 0.84 s |
| **Create Tree** | **5.2 s** | **22.84 s** | **52.18 s** | **85.59 s** | **144.98 s** |
| **Illustrate Tree** | 0.12 s | 0.11 s | 0.11 s | 0.11 s | 0.13 s |
| **Classification Accuracy** | 0.03 s | 0.05 s | 0.09 s | 0.13 s | 0.22 s |

### 6.1.1 CSADT Creation Time vs. Conditions

The number of conditions considered affects the computation time required for the creation of the CSADT tree. As the number of conditions increases, its computation time increases. The graph below confirms this and shows that the computation time required by the Python implementation was significantly higher than both the C++ Non-Threaded and C++ Threaded implementations. A tree created from 30 records, 100 conditions, creating only 5 nodes finished in ~272 seconds using the Python implementation while the same tree required less than 1 second for the C++ Threaded implementation (~272 times faster).

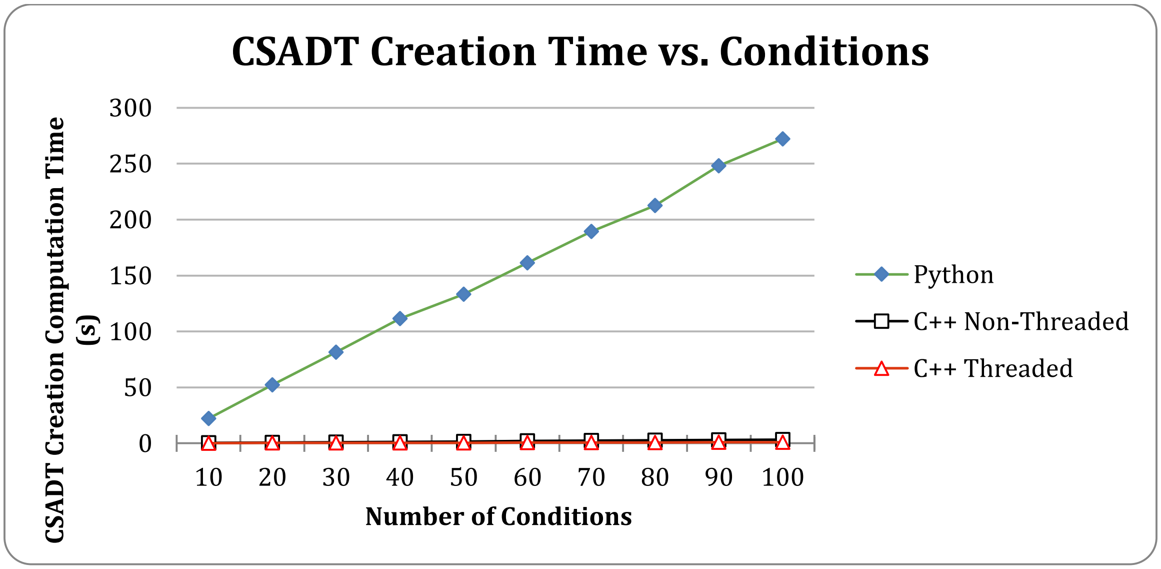


**Figure 24: CSADT Creation Time vs. Conditions**

In the graph above, the C++ Non-Threaded and C++ Threaded lines are too close to each other to differentiate them. Therefore, additional experiments were performed to demonstrate the difference, in computation time, between the C++ Non-Threaded and C++ Threaded implementations.

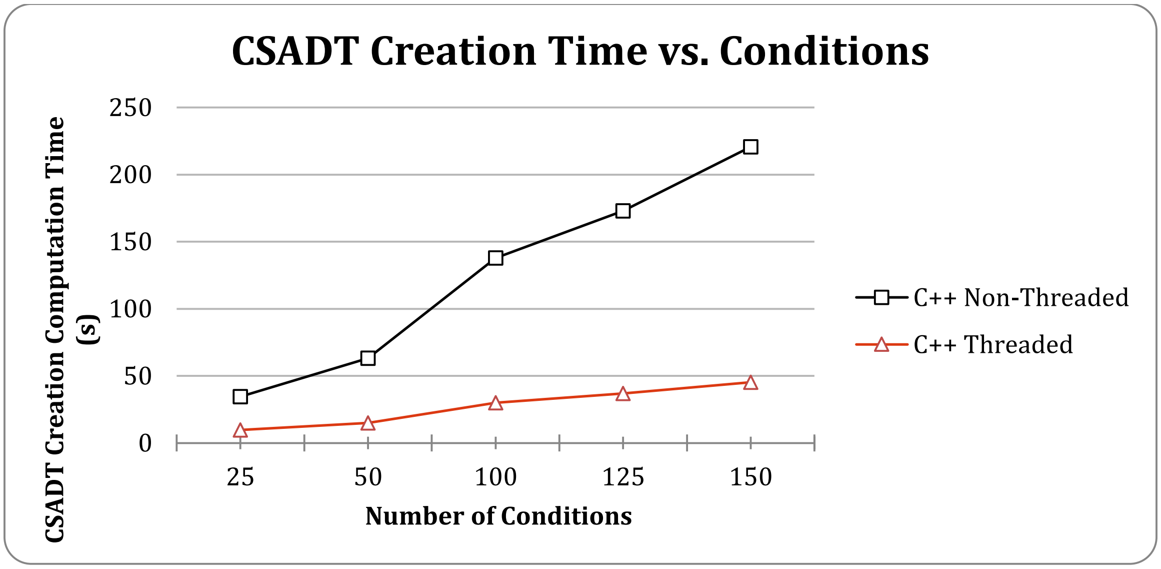


**Figure 25: C++ CSADT Creation Time vs. Conditions**

The computation time required in the creation of the CSADT tree is significantly decreased by using C++ as the language of implementation when compared to Python. Additionally, the computation time required in the creation of the CSADT tree is again significantly decreased by employing a threaded C++ implementation vs. a non-threaded solution.

### 6.1.2 CSADT Creation Time vs. Nodes

The number of nodes in a CSADT tree affects the computation time required for the creation of the CSADT tree. As the number of nodes increases, its computation time increases. The graph below confirms this and shows that the computation time required by the Python implementation was significantly higher than both the C++ Non-Threaded and C++ Threaded implementations. A tree created from 30 records, 20 conditions, creating only 10 nodes finished in ~206 seconds using the Python implementation while the same tree required less than 2 seconds for the C++ Threaded implementation (~103 times faster).

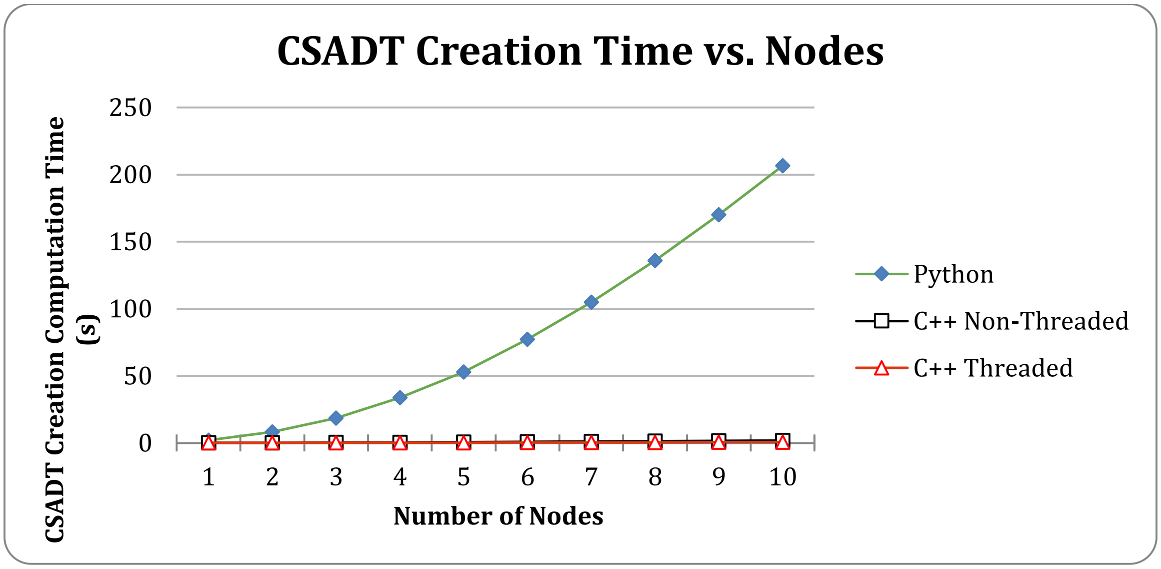


**Figure 26: CSADT Creation Time vs. Nodes**

Again, the C++ Non-Threaded and C++ Threaded lines are too close to each other. Therefore, additional experiments were performed to demonstrate the difference, in computation time, between the C++ Non-Threaded and C++ Threaded implementations.

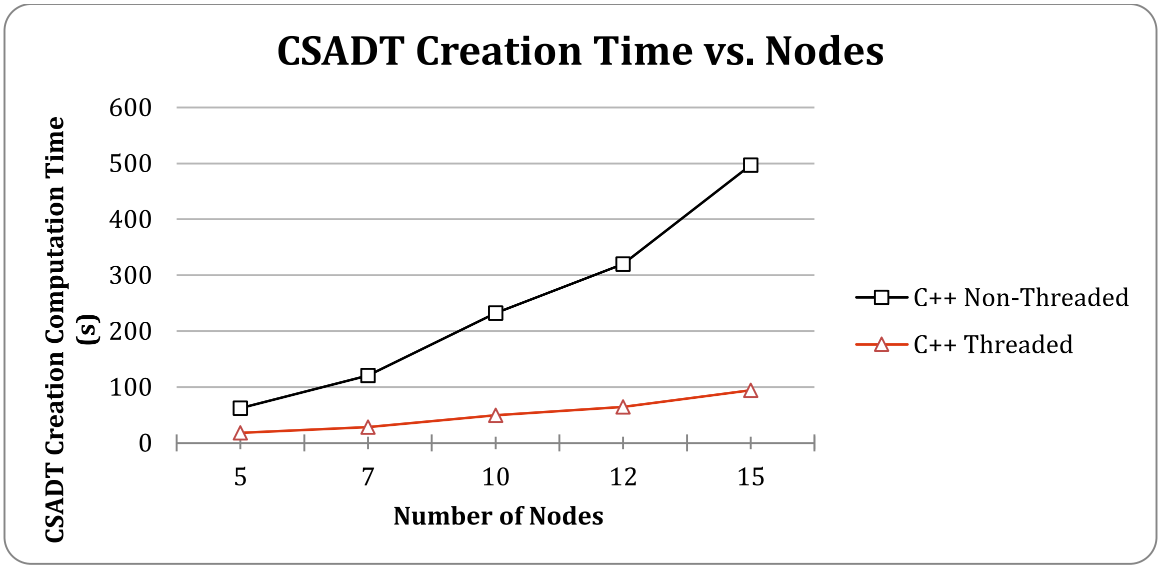


**Figure 27: C++ CSADT Creation Time vs. Nodes**

### 6.1.3 CSADT Creation Time vs. Records

The number of records consumed by the CSADT tree algorithm affects the computation time required for the creation of the CSADT tree. As the number of records increases, its computation time increases. The graph below confirms this and shows that the computation time required by the Python implementation was significantly higher than both the C++ Non-Threaded and C++ Threaded implementations. A tree created from 100 records, 20 conditions, creating only 5 nodes finished in ~618 seconds using the Python implementation while the same tree required less than 3 seconds for the C++ Threaded implementation (~206 times faster).

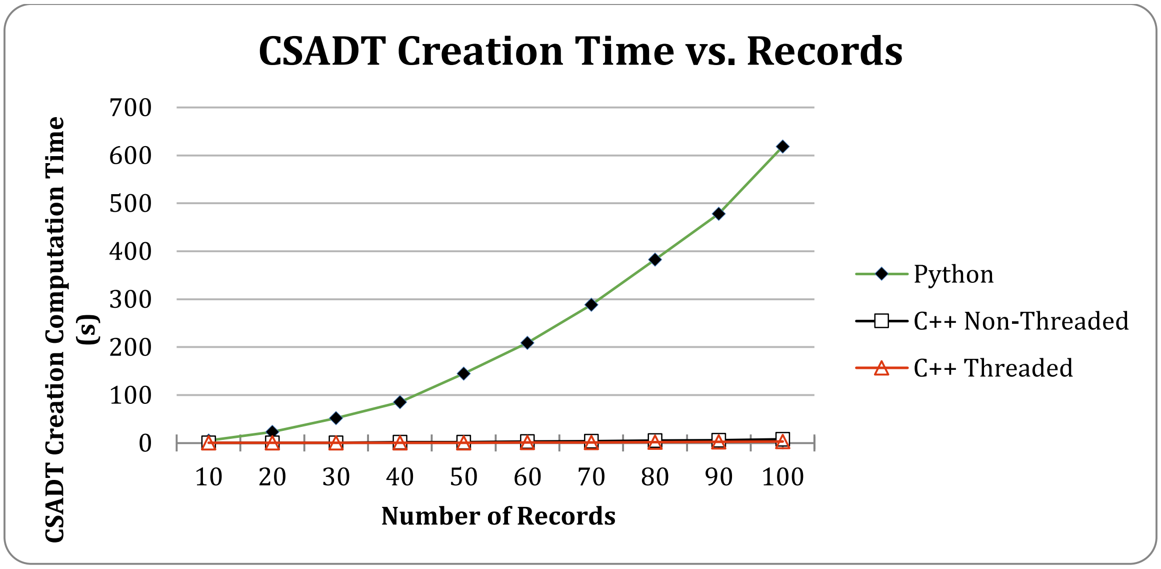


**Figure 28: CSADT Creation Time vs. Records**

Again, the C++ Non-Threaded and C++ Threaded lines are too close to each other to illustrate any differences. Therefore, additional experiments were performed to demonstrate the difference, in computation time, between the C++ Non-Threaded and C++ Threaded implementations.

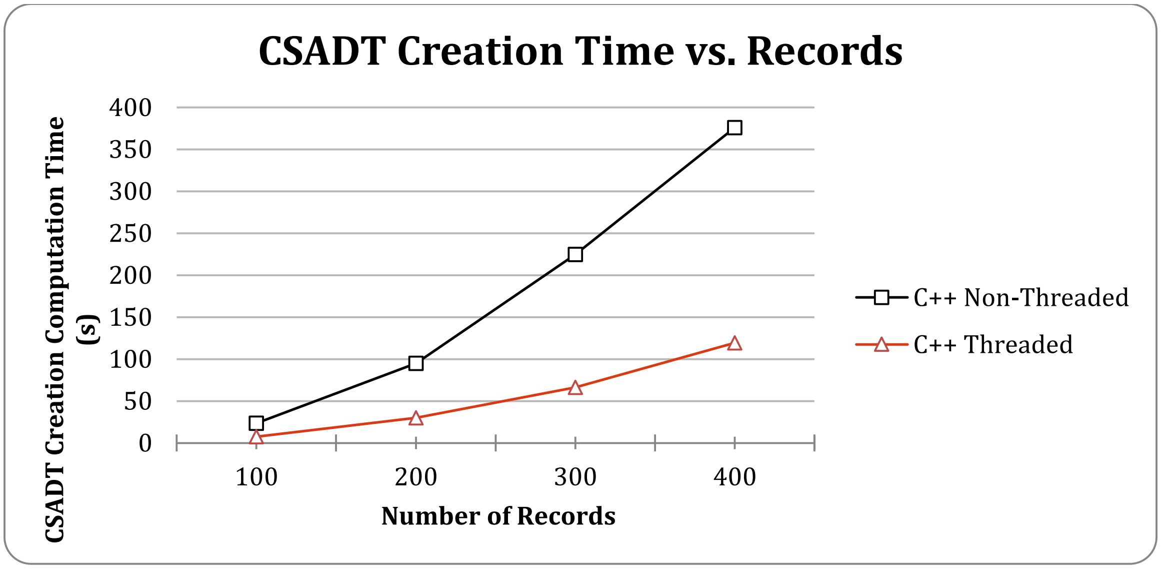


**Figure 29: C++ CSADT Creation Time vs. Records**

### 6.1.4 CSADT Threads

The C++ Threaded implementation invoked threads in batches to minimize runtime. The image below shows that the time required in the creation of the CSADT tree is directly related to the number of dedicated CPUs available.

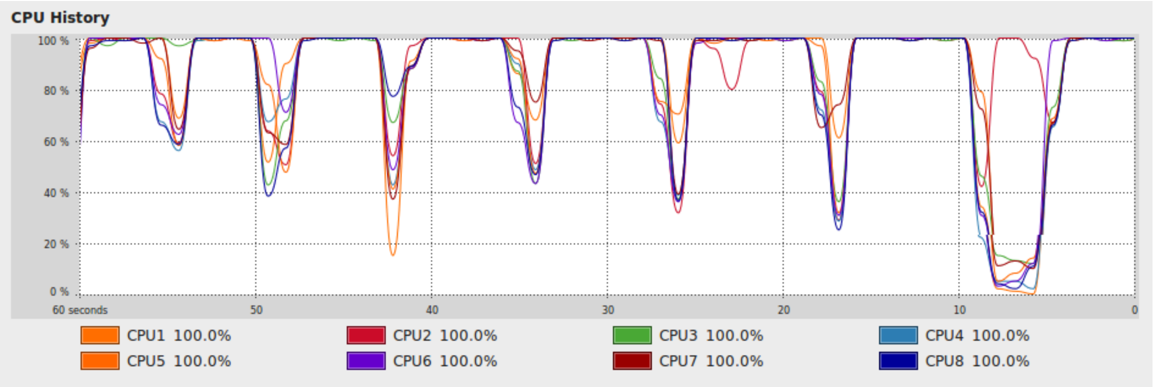


**Figure 30: C++ CPU Threads**

As shown the image above, all eight processors listed are at 100%. It is also apparent that all eight processors at 100% usage for ~5 seconds, at which time they are released only to be put back to work in less than 1 second. The implementation invokes a set number of threads at a time. In essence, the threads are invoked in batches. In an effort to simulate a set number of cores available the computer running the CSADT algorithm, the CSADT algorithm was run on the same dataset invoking 1, 2, 3, … and so on threads per batch. The dataset included:

- 178 conditions
- 15 nodes
- 100 records

Again, all experiments were run on an 8-core desktop computer. From the graph below, it is clear that the threads per batch decreases the CSADT Tree Creation computation time noticeably until ~8 threads per batch.

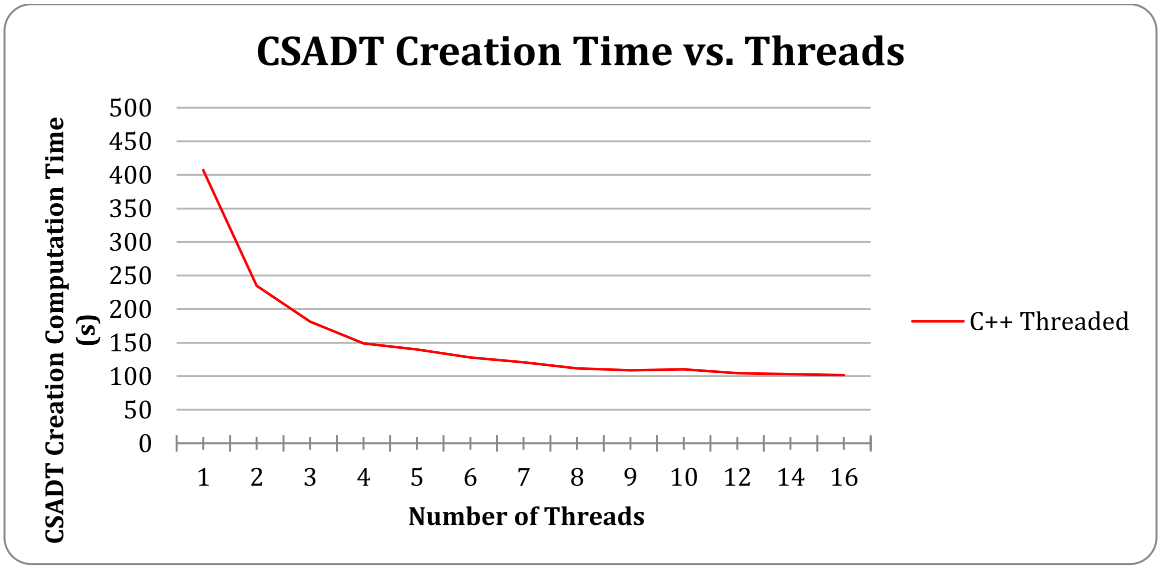


**Figure 31: C++ CSADT Threading Performance**

### 6.2 Person Data Accuracy Analysis

In order to quantitatively judge the accuracy of the CSADT tree, its training data was replayed on itself. Record-pairs were predicted as matches or non-matches.

Comparing a record-pair's prediction to its actual classification dictates whether the prediction/classification is a true positive, true negative, false positive, or false negative.

- True positives
    - when two records are predicted as a match and they actually are a match
- True negatives
    - when two records are predicted as not matching and they actually do not match
- False positives
    - when two records are predicted as a non-match when they are actually a match
- False negatives
    - when two records are predicted as not matching while they are actually a match

The description above is illustrated in the confusion matrix below.

**Table 14: Confusion Matrix**

| | | Actual | |
|---|---|---|---|
| | | Negative | Positive |
| Prediction | Negative | True Negatives | False Positives |
| | Positive | False Negatives | True Positives |

Once the record-pairs have been classified and the confusion matrix is created, the precision, recall, and accuracy were computed:

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives}$$

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives}$$

$$All\ Positives = True\ Positives + False\ Positives$$

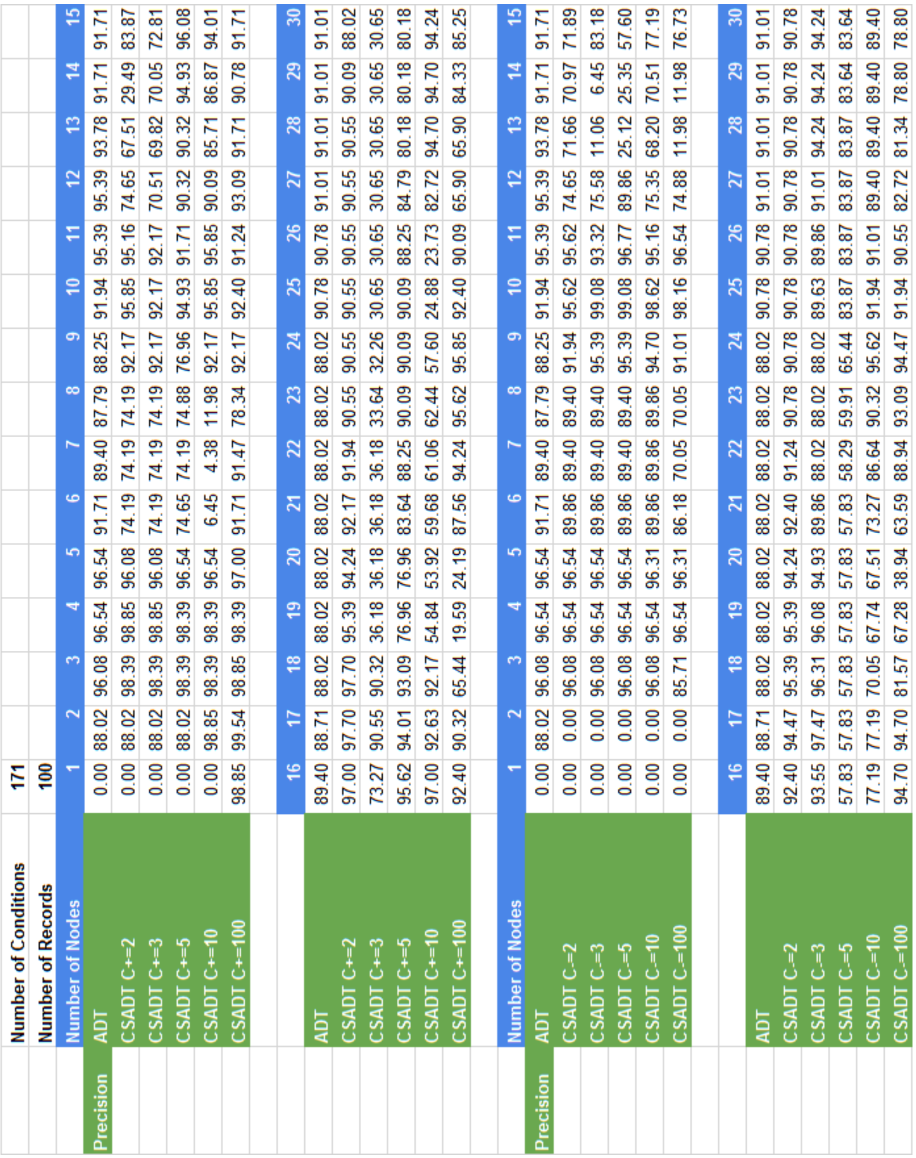
$$All\ Negatives = True\ Negatives + False\ Negatives$$

$$Accuracy = \frac{True\ Positives + True\ Negatives}{All\ Positives + All\ Negatives}$$

Nearly all of the input record-pairs are non-matches. The training data included 644 records (207,046 record-pairs). Of the 207,046 record-pairs, 206,612 were non-matches (99.79%) while only 434 were matches (0.21%). Therefore, if a decision tree predicted *every* record pair as a non-match, it would have 99.79% accuracy. Clearly, the accuracy is not the most important metric to pursue in this record linkage challenge. Instead, precision and recall are explored depth as it is far more important to have a high precision and high recall concerning this person-record dataset.

### 6.2.1 CSADT Precision

The figure below contains the precision percentages derived from CSADT trees generated from 171 conditions, 100 records (4,950 record-pairs), containing 1 to 30 nodes.



| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Number of Conditions | 171 | | | | | | | | | | | | | | |
| | Number of Records | 100 | | | | | | | | | | | | | | |
| | Number of Nodes | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| Precision | ADT | 0.00 | 88.02 | 96.08 | 96.54 | 96.54 | 91.71 | 89.40 | 87.79 | 88.25 | 91.94 | 95.39 | 95.39 | 93.78 | 91.71 | 91.71 |
| | CSADT C+=2 | 0.00 | 88.02 | 98.39 | 98.85 | 96.08 | 74.19 | 74.19 | 74.19 | 92.17 | 95.85 | 95.16 | 74.65 | 67.51 | 29.49 | 83.87 |
| | CSADT C+=3 | 0.00 | 88.02 | 98.39 | 98.85 | 96.08 | 74.19 | 74.19 | 74.19 | 92.17 | 92.17 | 92.17 | 70.51 | 69.82 | 70.05 | 72.81 |
| | CSADT C+=5 | 0.00 | 88.02 | 98.39 | 98.39 | 96.54 | 74.65 | 74.19 | 74.88 | 76.96 | 94.93 | 91.71 | 90.32 | 90.32 | 94.93 | 96.08 |
| | CSADT C+=10 | 0.00 | 98.85 | 98.39 | 98.39 | 96.54 | 6.45 | 4.38 | 11.98 | 92.17 | 95.85 | 95.85 | 90.09 | 85.71 | 86.87 | 94.01 |
| | CSADT C+=100 | 98.85 | 99.54 | 98.85 | 98.39 | 97.00 | 91.71 | 91.47 | 78.34 | 92.17 | 92.40 | 91.24 | 93.09 | 91.71 | 90.78 | 91.71 |
| | | | | | | | | | | | | | | | | |
| | | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| | ADT | 89.40 | 88.71 | 88.02 | 88.02 | 88.02 | 88.02 | 88.02 | 88.02 | 88.02 | 90.78 | 90.78 | 91.01 | 91.01 | 91.01 | 91.01 |
| | CSADT C+=2 | 97.00 | 97.70 | 97.70 | 95.39 | 94.24 | 92.17 | 91.94 | 90.55 | 90.55 | 90.55 | 90.55 | 90.55 | 90.55 | 90.09 | 88.02 |
| | CSADT C+=3 | 73.27 | 90.55 | 90.32 | 36.18 | 36.18 | 36.18 | 36.18 | 33.64 | 32.26 | 30.65 | 30.65 | 30.65 | 30.65 | 30.65 | 30.65 |
| | CSADT C+=5 | 95.62 | 94.01 | 93.09 | 76.96 | 76.96 | 83.64 | 88.25 | 90.09 | 90.09 | 90.09 | 88.25 | 84.79 | 80.18 | 80.18 | 80.18 |
| | CSADT C+=10 | 97.00 | 92.63 | 92.17 | 54.84 | 53.92 | 59.68 | 61.06 | 62.44 | 57.60 | 24.88 | 23.73 | 82.72 | 94.70 | 94.70 | 94.24 |
| | CSADT C+=100 | 92.40 | 90.32 | 65.44 | 19.59 | 24.19 | 87.56 | 94.24 | 95.62 | 95.85 | 92.40 | 90.09 | 65.90 | 65.90 | 84.33 | 85.25 |
| | | | | | | | | | | | | | | | | |
| | Number of Nodes | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| Precision | ADT | 0.00 | 88.02 | 96.08 | 96.54 | 96.54 | 91.71 | 89.40 | 87.79 | 88.25 | 91.94 | 95.39 | 95.39 | 93.78 | 91.71 | 91.71 |
| | CSADT C-=2 | 0.00 | 0.00 | 96.08 | 96.54 | 96.54 | 89.86 | 89.40 | 89.40 | 91.94 | 95.62 | 95.62 | 74.65 | 71.66 | 70.97 | 71.89 |
| | CSADT C-=3 | 0.00 | 0.00 | 96.08 | 96.54 | 96.54 | 89.86 | 89.40 | 89.40 | 95.39 | 99.08 | 93.32 | 75.58 | 11.06 | 6.45 | 83.18 |
| | CSADT C-=5 | 0.00 | 0.00 | 96.08 | 96.54 | 96.54 | 89.86 | 89.40 | 89.40 | 95.39 | 99.08 | 96.77 | 89.86 | 25.12 | 25.35 | 57.60 |
| | CSADT C-=10 | 0.00 | 0.00 | 96.08 | 96.54 | 96.31 | 89.86 | 89.86 | 89.86 | 94.70 | 98.62 | 95.16 | 75.35 | 68.20 | 70.51 | 77.19 |
| | CSADT C-=100 | 0.00 | 0.00 | 85.71 | 96.54 | 96.31 | 86.18 | 70.05 | 70.05 | 91.01 | 98.16 | 96.54 | 74.88 | 11.98 | 11.98 | 76.73 |
| | | | | | | | | | | | | | | | | |
| | | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| | ADT | 89.40 | 88.71 | 88.02 | 88.02 | 88.02 | 88.02 | 88.02 | 88.02 | 88.02 | 90.78 | 90.78 | 91.01 | 91.01 | 91.01 | 91.01 |
| | CSADT C-=2 | 92.40 | 94.47 | 95.39 | 95.39 | 94.24 | 92.40 | 91.24 | 90.78 | 90.78 | 90.78 | 90.78 | 90.78 | 90.78 | 90.78 | 90.78 |
| | CSADT C-=3 | 93.55 | 97.47 | 96.31 | 96.08 | 94.93 | 89.86 | 88.02 | 88.02 | 88.02 | 89.63 | 89.86 | 91.01 | 94.24 | 94.24 | 94.24 |
| | CSADT C-=5 | 57.83 | 57.83 | 57.83 | 57.83 | 57.83 | 57.83 | 58.29 | 59.91 | 65.44 | 83.87 | 83.87 | 83.87 | 83.87 | 83.64 | 83.64 |
| | CSADT C-=10 | 77.19 | 77.19 | 70.05 | 67.74 | 67.51 | 73.27 | 86.64 | 90.32 | 95.62 | 91.94 | 91.01 | 89.40 | 89.40 | 89.40 | 89.40 |
| | CSADT C-=100 | 94.70 | 94.70 | 81.57 | 67.28 | 38.94 | 63.59 | 88.94 | 93.09 | 94.47 | 91.94 | 90.55 | 82.72 | 81.34 | 78.80 | 78.80 |

**Figure 32: Precision Results**

The following two graphs visually illustrate the effect of varying cost sensitivities on precision.

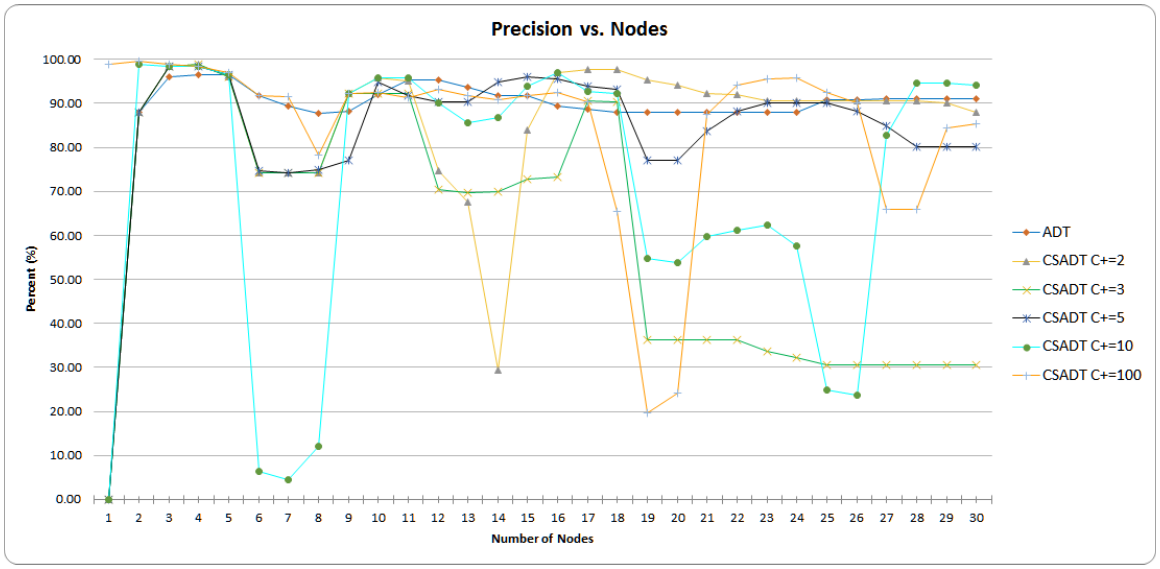


**Figure 33: Precision Comparisons C+**

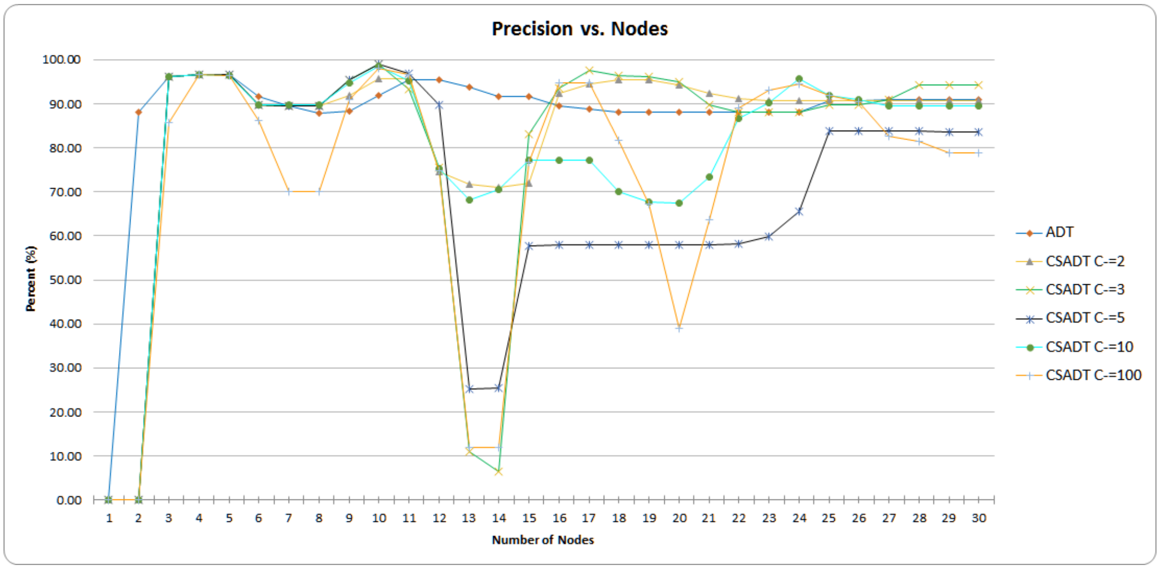
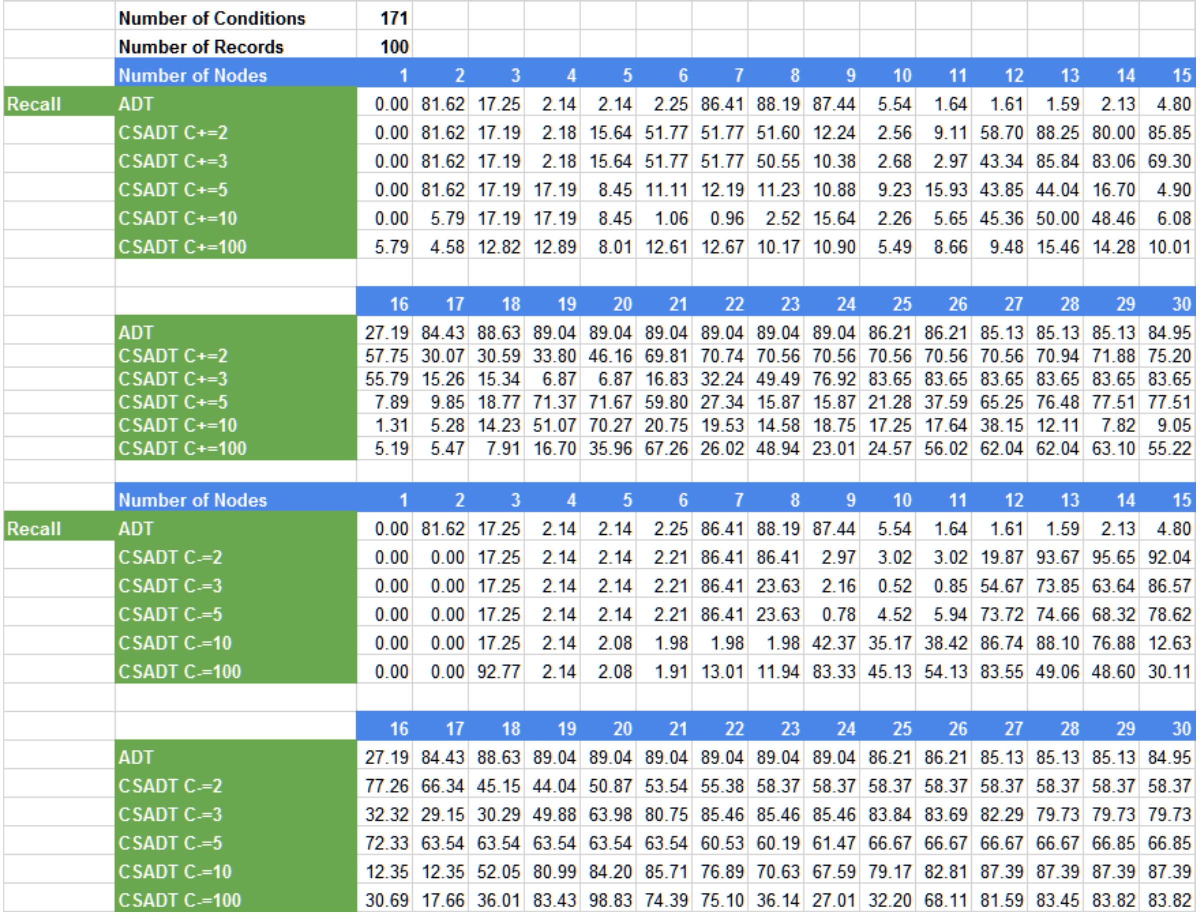


**Figure 34: Precision Comparisons C-**

After adding 30 nodes, two cost sensitivity pairs resulted in the highest precision at 94.24%: C+=10, C-=1 and C+=1, C-=3.

### 6.2.2 CSADT Recall

The figure below contains the recall percentages derived from CSADT trees generated from 171 conditions, 100 records (4,950 record-pairs), containing 1 to 30 nodes.



| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Number of Conditions | 171 | | | | | | | | | | | | | | |
| | Number of Records | 100 | | | | | | | | | | | | | | |
| | Number of Nodes | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| Recall | ADT | 0.00 | 81.62 | 17.25 | 2.14 | 2.14 | 2.25 | 86.41 | 88.19 | 87.44 | 5.54 | 1.64 | 1.61 | 1.59 | 2.13 | 4.80 |
| | CSADT C+=2 | 0.00 | 81.62 | 17.19 | 2.18 | 15.64 | 51.77 | 51.77 | 51.60 | 12.24 | 2.56 | 9.11 | 58.70 | 88.25 | 80.00 | 85.85 |
| | CSADT C+=3 | 0.00 | 81.62 | 17.19 | 2.18 | 15.64 | 51.77 | 51.77 | 50.55 | 10.38 | 2.68 | 2.97 | 43.34 | 85.84 | 83.06 | 69.30 |
| | CSADT C+=5 | 0.00 | 81.62 | 17.19 | 17.19 | 8.45 | 11.11 | 12.19 | 11.23 | 10.88 | 9.23 | 15.93 | 43.85 | 44.04 | 16.70 | 4.90 |
| | CSADT C+=10 | 0.00 | 5.79 | 17.19 | 17.19 | 8.45 | 1.06 | 0.96 | 2.52 | 15.64 | 2.26 | 5.65 | 45.36 | 50.00 | 48.46 | 6.08 |
| | CSADT C+=100 | 5.79 | 4.58 | 12.82 | 12.89 | 8.01 | 12.61 | 12.67 | 10.17 | 10.90 | 5.49 | 8.66 | 9.48 | 15.46 | 14.28 | 10.01 |
| | | | | | | | | | | | | | | | | |
| | | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| | ADT | 27.19 | 84.43 | 88.63 | 89.04 | 89.04 | 89.04 | 89.04 | 89.04 | 89.04 | 86.21 | 86.21 | 85.13 | 85.13 | 85.13 | 84.95 |
| | CSADT C+=2 | 57.75 | 30.07 | 30.59 | 33.80 | 46.16 | 69.81 | 70.74 | 70.56 | 70.56 | 70.56 | 70.56 | 70.56 | 70.94 | 71.88 | 75.20 |
| | CSADT C+=3 | 55.79 | 15.26 | 15.34 | 6.87 | 6.87 | 16.83 | 32.24 | 49.49 | 76.92 | 83.65 | 83.65 | 83.65 | 83.65 | 83.65 | 83.65 |
| | CSADT C+=5 | 7.89 | 9.85 | 18.77 | 71.37 | 71.67 | 59.80 | 27.34 | 15.87 | 15.87 | 21.28 | 37.59 | 65.25 | 76.48 | 77.51 | 77.51 |
| | CSADT C+=10 | 1.31 | 5.28 | 14.23 | 51.07 | 70.27 | 20.75 | 19.53 | 14.58 | 18.75 | 17.25 | 17.64 | 38.15 | 12.11 | 7.82 | 9.05 |
| | CSADT C+=100 | 5.19 | 5.47 | 7.91 | 16.70 | 35.96 | 67.26 | 26.02 | 48.94 | 23.01 | 24.57 | 56.02 | 62.04 | 62.04 | 63.10 | 55.22 |
| | | | | | | | | | | | | | | | | |
| | Number of Nodes | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| Recall | ADT | 0.00 | 81.62 | 17.25 | 2.14 | 2.14 | 2.25 | 86.41 | 88.19 | 87.44 | 5.54 | 1.64 | 1.61 | 1.59 | 2.13 | 4.80 |
| | CSADT C-=2 | 0.00 | 0.00 | 17.25 | 2.14 | 2.14 | 2.21 | 86.41 | 86.41 | 2.97 | 3.02 | 3.02 | 19.87 | 93.67 | 95.65 | 92.04 |
| | CSADT C-=3 | 0.00 | 0.00 | 17.25 | 2.14 | 2.14 | 2.21 | 86.41 | 23.63 | 2.16 | 0.52 | 0.85 | 54.67 | 73.85 | 63.64 | 86.57 |
| | CSADT C-=5 | 0.00 | 0.00 | 17.25 | 2.14 | 2.14 | 2.21 | 86.41 | 23.63 | 0.78 | 4.52 | 5.94 | 73.72 | 74.66 | 68.32 | 78.62 |
| | CSADT C-=10 | 0.00 | 0.00 | 17.25 | 2.14 | 2.08 | 1.98 | 1.98 | 1.98 | 42.37 | 35.17 | 38.42 | 86.74 | 88.10 | 76.88 | 12.63 |
| | CSADT C-=100 | 0.00 | 0.00 | 92.77 | 2.14 | 2.08 | 1.91 | 13.01 | 11.94 | 83.33 | 45.13 | 54.13 | 83.55 | 49.06 | 48.60 | 30.11 |
| | | | | | | | | | | | | | | | | |
| | | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| | ADT | 27.19 | 84.43 | 88.63 | 89.04 | 89.04 | 89.04 | 89.04 | 89.04 | 89.04 | 86.21 | 86.21 | 85.13 | 85.13 | 85.13 | 84.95 |
| | CSADT C-=2 | 77.26 | 66.34 | 45.15 | 44.04 | 50.87 | 53.54 | 55.38 | 58.37 | 58.37 | 58.37 | 58.37 | 58.37 | 58.37 | 58.37 | 58.37 |
| | CSADT C-=3 | 32.32 | 29.15 | 30.29 | 49.88 | 63.98 | 80.75 | 85.46 | 85.46 | 85.46 | 83.84 | 83.69 | 82.29 | 79.73 | 79.73 | 79.73 |
| | CSADT C-=5 | 72.33 | 63.54 | 63.54 | 63.54 | 63.54 | 63.54 | 60.53 | 60.19 | 61.47 | 66.67 | 66.67 | 66.67 | 66.67 | 66.85 | 66.85 |
| | CSADT C-=10 | 12.35 | 12.35 | 52.05 | 80.99 | 84.20 | 85.71 | 76.89 | 70.63 | 67.59 | 79.17 | 82.81 | 87.39 | 87.39 | 87.39 | 87.39 |
| | CSADT C-=100 | 30.69 | 17.66 | 36.01 | 83.43 | 98.83 | 74.39 | 75.10 | 36.14 | 27.01 | 32.20 | 68.11 | 81.59 | 83.45 | 83.82 | 83.82 |

Figure 35: Recall Results

The following two graphs visually illustrate the effect of varying cost sensitivities on recall.

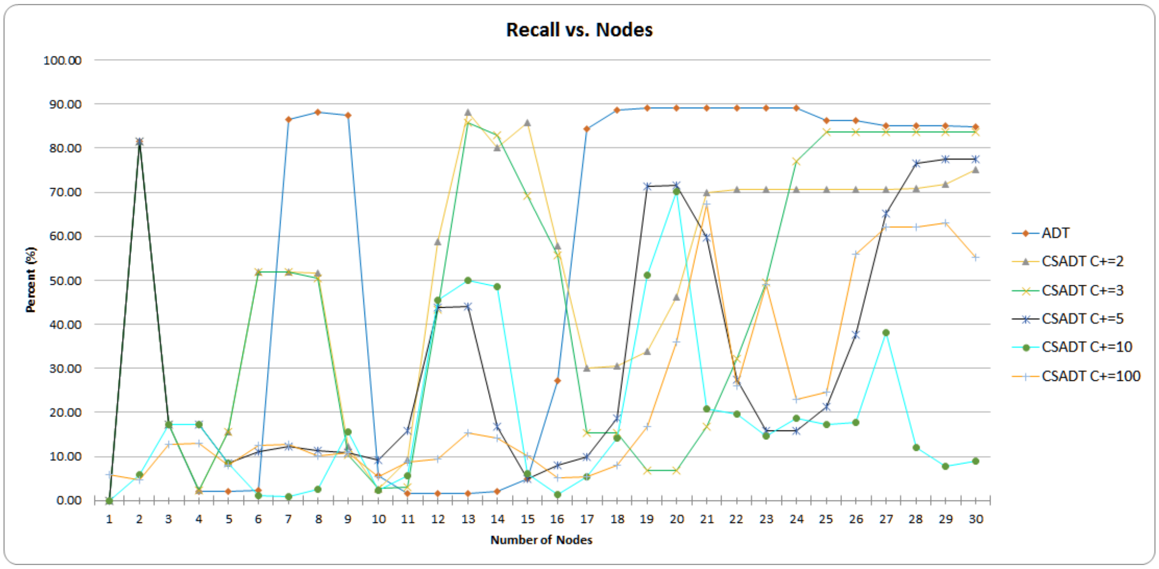


Figure 36: Recall Comparisons C+

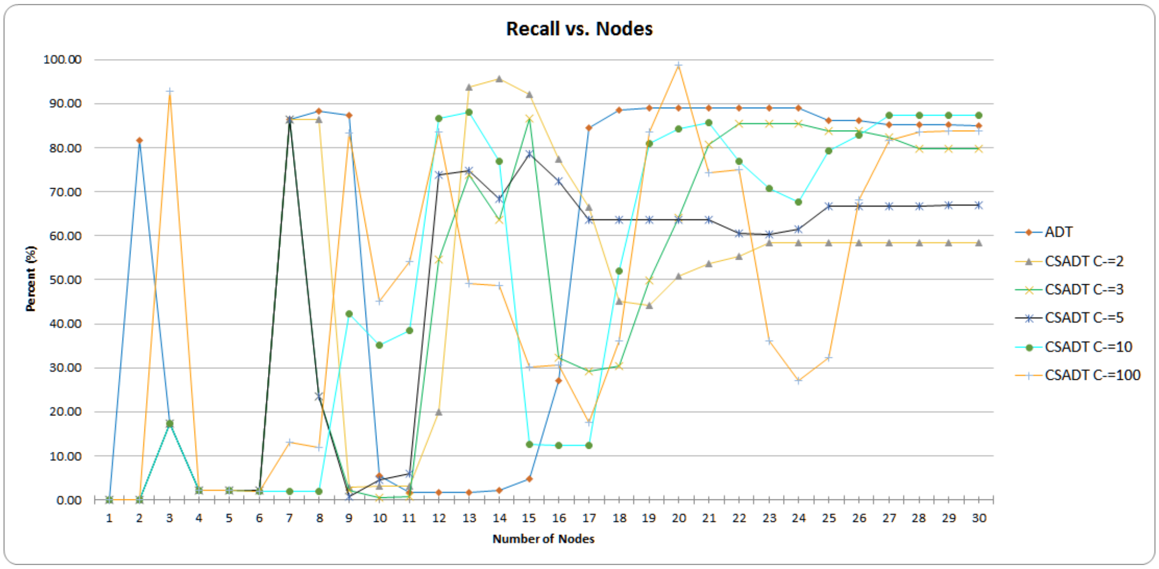


Figure 37: Recall Comparisons C-

After adding 30 nodes, one cost sensitivity pair resulted in the highest recall at 87.39%: C+=1, C-=10.

### 6.2.3 CSADT Accuracy

The training dataset contained 644 records. From 644 records, 207,046 unique record-pairs can be derived. Using 171 conditions generated automatically, multiple experiments were run. The accuracy metrics concerning the CSADT tree whose combined recall and precision was highest is described shown below.

**Table 15: Best CSADT tree results**

| Description | Value |
|---|---|
| Cost Sensitivity + | 1 |
| Cost Sensitivity - | 10 |
| Number of Records | 644 |
| Training Data Record-Pairs | 207,046 |
| Number of Matches | 434 |
| Number of Non-Matches | 206,612 |
| Number of Predicted Matches | 444 |
| Number of Predicted Non-Matches | 206,602 |
| True Positives | 388 (89.40%) |
| True Negatives | 206,612 (99.97%) |
| False Positives | 56 |
| False Negatives | 46 |
| Recall | 89.40% |
| Precision | 87.39% |
| Accuracy | 99.95% |

The following is the actual decision tree that can predict with 99.95% accuracy.

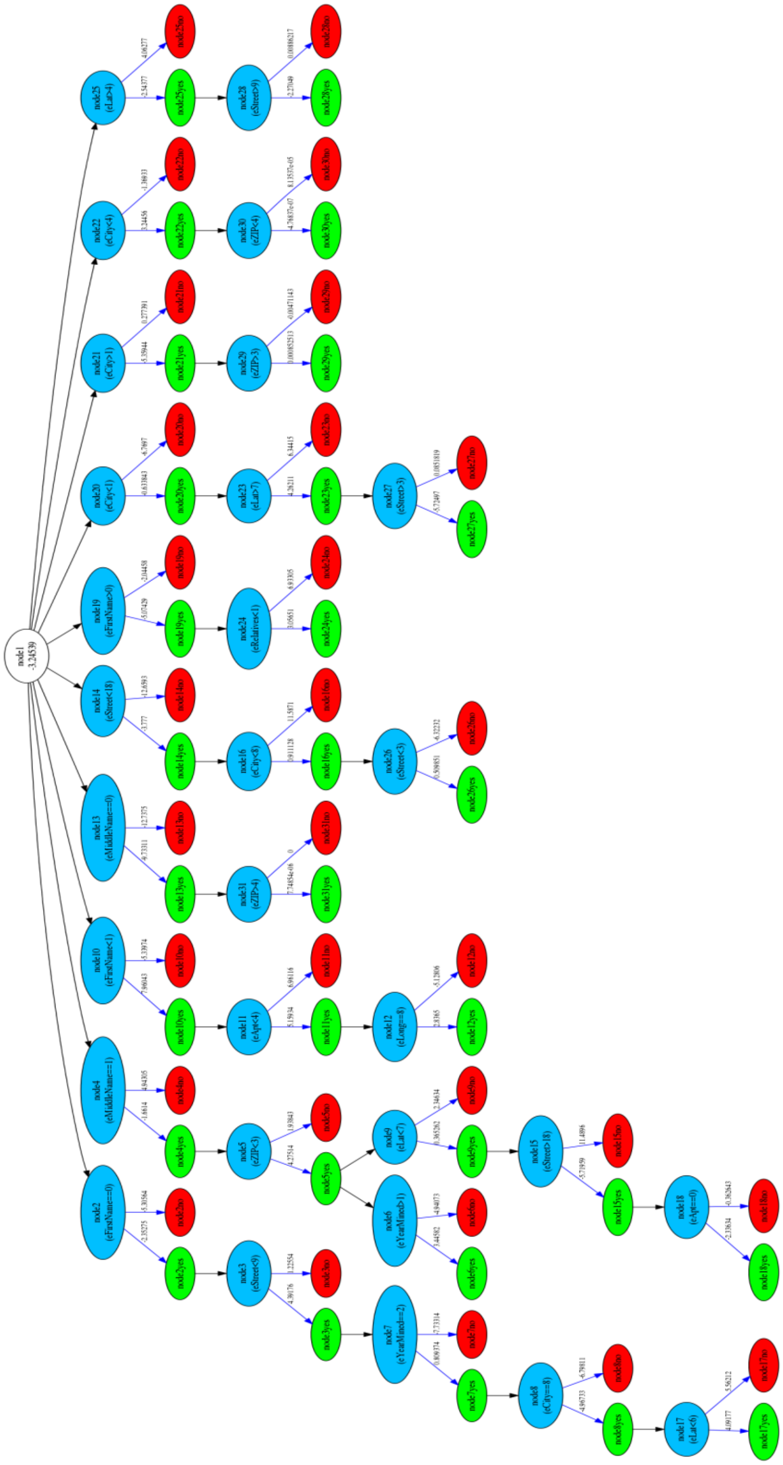


**Figure 38: People CSADT Tree 99.95% Accuracy**

## 6.3 NBA Dataset

The CSADT algorithm can be applied to non RL classification problems. To demonstrate this, NBA statistics were gathered and organized into training data.

### 6.3.1 NBA Statistics

The following statistics are available on http://www.databasebasketball.com.

1. Player regular season statistics
2. Player regular season career totals
3. Player playoff statistics
4. Player playoff career totals
5. Player All-Star game statistcs
6. Team regular season statistics
7. Complete draft history

The sixth element above (team regular season statistics) includes the following basketball attributes:

1. Points
2. Offensive rebounds
3. Defensive rebounds
4. Rebounds
5. Assists
6. Steals
7. Blocks
8. Turn-overs
9. Personal fouls

10. Field goal attempted

11. Field goal made

12. Free throws attempted

13. Free throws made

14. Three pointers attempted

15. Three points

### 6.3.2 NBA Accuracy Analysis

There are 30 teams in the NBA; 15 are Western Conference teams while the other 15 are Eastern Conference teams. At the end of the regular season, the top eight of either conference advance to the playoffs. Given all 30 teams with the 15 attributes listed above, the following question was proposed to test the CSADT algorithm on, “Will team X merit a spot in the playoffs?”

Given enough nodes, the CSADT algorithm was able to reach over 80% accuracy with greater than 75% precision and recall. Given the ever unpredictable sports landscape, these results are surprisingly encouraging.

10 Folds Cross Validation with 10 Nodes:

- Precision: 77.64%
- Recall: 67.57%
- Accuracy: 78.58%

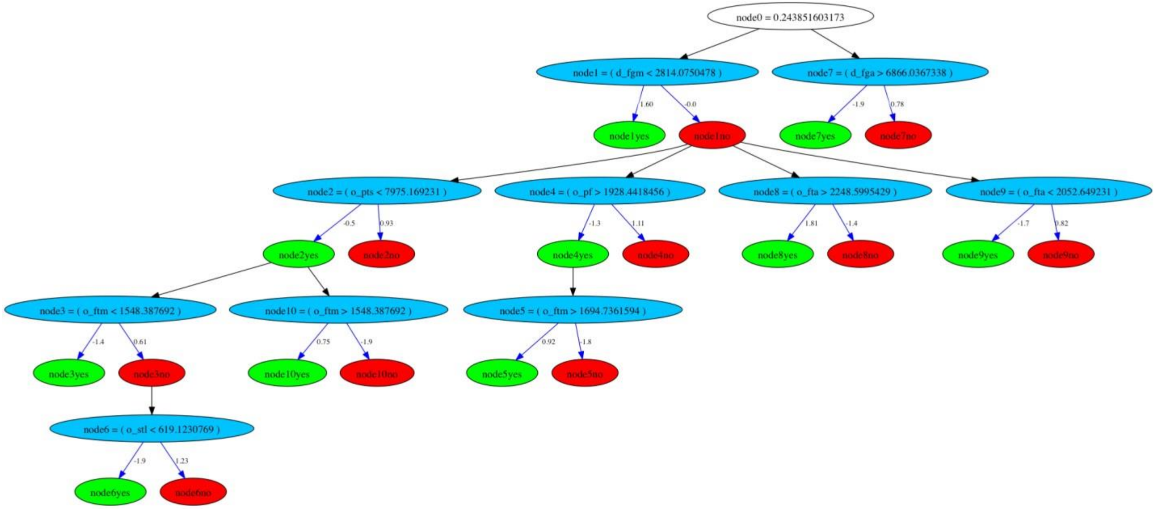

node0 = 0.243851603173
node1 = ( d_fgm < 2814.0750478 )
node7 = ( d_fga > 6866.0367338 )
1.60
-0.0
-1.9
0.78
node1yes
node1no
node7yes
node7no
node2 = ( o_pts < 7975.169231 )
node4 = ( o_pf > 1928.4418456 )
node8 = ( o_fta > 2248.5995429 )
node9 = ( o_fta < 2052.649231 )
-0.5
0.93
-1.3
1.11
1.81
-1.4
-1.7
0.82
node2yes
node2no
node4yes
node4no
node8yes
node8no
node9yes
node9no
node3 = ( o_ftm < 1548.387692 )
node10 = ( o_ftm > 1548.387692 )
node5 = ( o_ftm > 1694.7361594 )
-1.4
0.61
0.75
-1.9
0.92
-1.8
node3yes
node3no
node10yes
node10no
node5yes
node5no
node6 = ( o_stl < 619.1230769 )
-1.9
1.23
node6yes
node6no

**Figure 39: NBA CSADT Tree 1**

2 Folds Cross Validation with 25 Nodes

- Precision: 86.42%
- Recall: 75.27%
- Accuracy: 80.79%

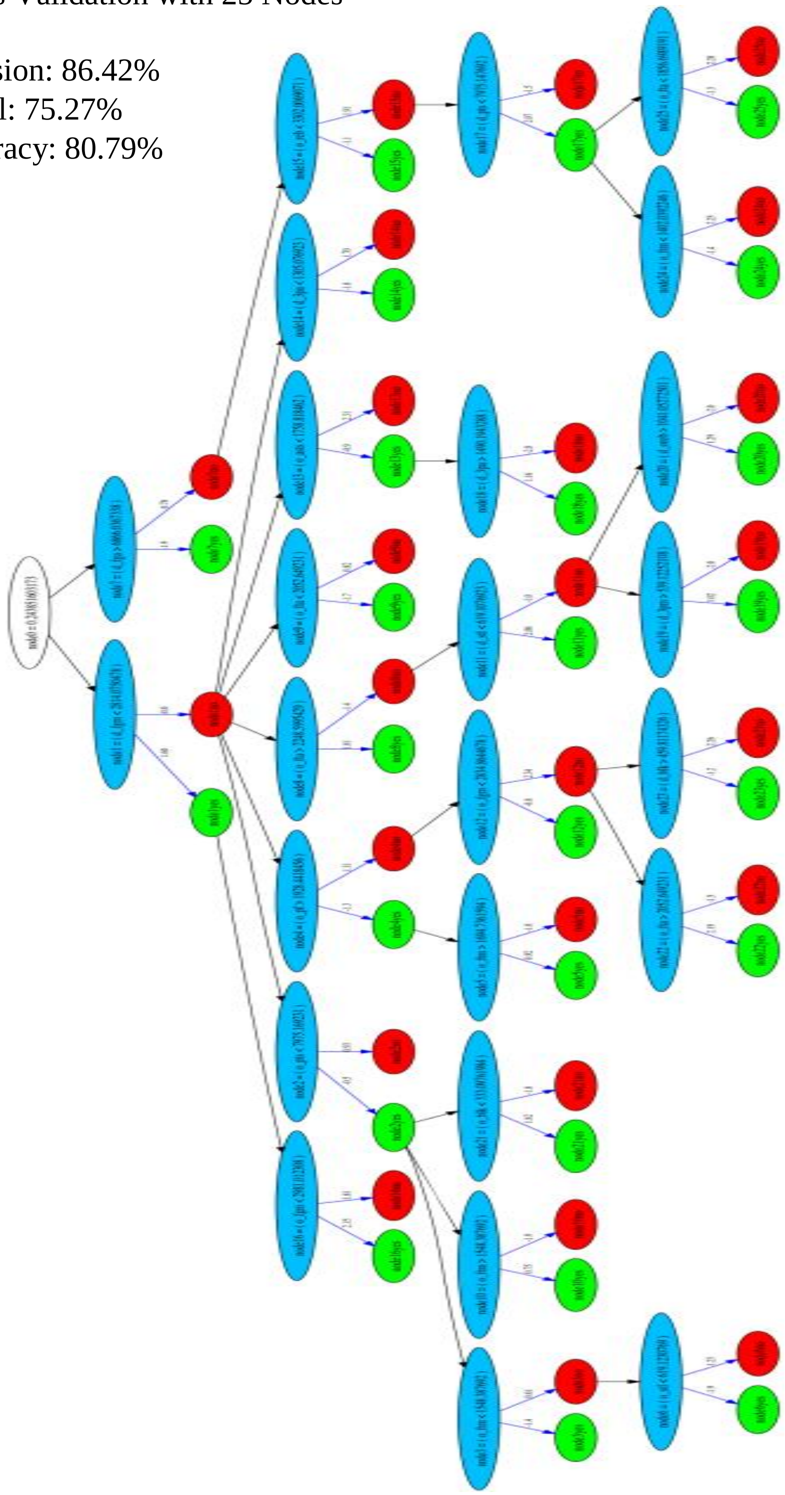


**Figure 40: NBA CSADT Tree 2**

# 7. FUTURE WORK AND CONCLUSION

## 7.1 Person Data Accuracy Improvements

There are several ideas that may improve the accuracy of the CSADT algorithm. Below are a few ideas that could be explored.

### 7.1.1 Additional Conditions

Typically, ML algorithms use attributes as conditions. Sometimes multiple conditions are generated based upon one attribute. Oftentimes attributes contain information that could be extracted into something new to use as a condition. For example, a phone number is one of the attributes present in the dataset used in this experiment. As expected, records' phone numbers were compared wholesale. Additional conditions stemming from the phone number attribute could be derived from the following procedure:

1. Extract the area code out of the phone number
2. Determine the geographical location of the extracted area code from step 1
3. Calculate the distance between the locations determined in step 2
4. Determine average distances between locations in step 3 (in matches and non-matches)
5. Create condition based on the averages found in step 4

## 7.2 Person Data Speed Improvements

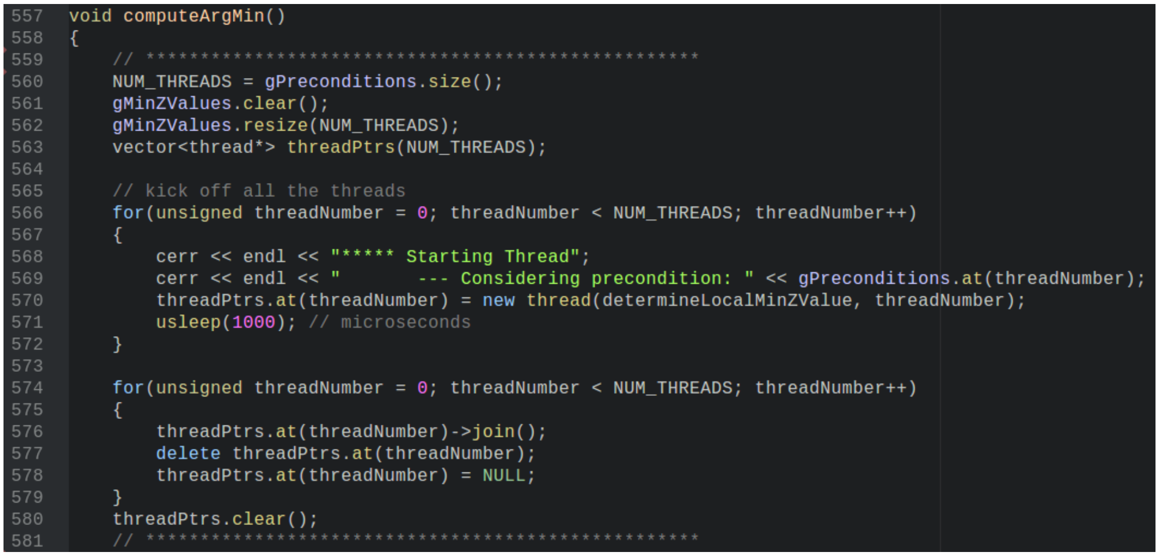
In the CSADT algorithm's current implementation, several updates could be made in order to increase the speed of computation.

### 7.2.1 Language Selection

This system was originally implemented in Python. While Python served its purpose in allowing the system to be developed relatively quickly, it failed to produce sufficient speed requirements. The CSADT algorithm contains several procedures that could be parallelized. One of the major limitations of the Python language is that it does not support threads, effectively making it impossible to run tasks in parallel. Therefore, in an effort to speed up the computation time required in generating a decision tree, the sections of the project requiring the most CPU cycles were developed C++. C++ was an obvious choice as it supports threads (using boost C++ libraries) and its speed is unrivaled largely due to it being strongly typed and closer to the hardware than most languages.

### 7.2.2 Parallelization

Decision tree algorithms iterate over a loop which is responsible for selecting a condition (through induction) to add as a node to the tree it is creating. The CSADT algorithm iterates over the function *computeArgMin* as this function selects the "best" condition to add to the decision tree. Improving this function could dramatically improve the speed of the CSADT algorithm because of its high computation cost. While this function has already been improved by replacing its original iterative solution with a threaded solution, the task of optimizing this function (and the functions it calls) is far from over. By reviewing the code provided below, it is easily seen that for each precondition, a thread is kicked off which will consider the available (unselected) conditions as candidates for the next splitter node.



```cpp
void computeArgMin()
{
    // ***************************************************
    NUM_THREADS = gPreconditions.size();
    gMinZValues.clear();
    gMinZValues.resize(NUM_THREADS);
    vector<thread*> threadPtrs(NUM_THREADS);

    // kick off all the threads
    for(unsigned threadNumber = 0; threadNumber < NUM_THREADS; threadNumber++)
    {
        cerr << endl << "***** Starting Thread";
        cerr << endl << "       --- Considering precondition: " << gPreconditions.at(threadNumber);
        threadPtrs.at(threadNumber) = new thread(determineLocalMinZValue, threadNumber);
        usleep(1000); // microseconds
    }

    for(unsigned threadNumber = 0; threadNumber < NUM_THREADS; threadNumber++)
    {
        threadPtrs.at(threadNumber)->join();
        delete threadPtrs.at(threadNumber);
        threadPtrs.at(threadNumber) = NULL;
    }
    threadPtrs.clear();
    // ***************************************************
```

**Figure 41: Thread per Pre-Condition**

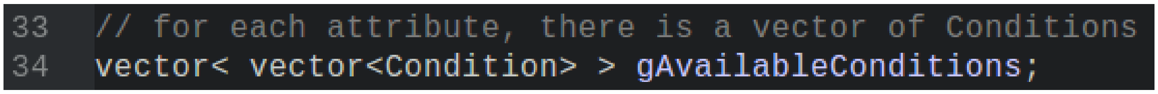
For example, suppose there were three preconditions and eight available, not yet selected, conditions remaining. Three threads would be created, each considering the eight available conditions. For each thread (for each precondition), the “best” available condition is found by first computing the z-values of the precondition/condition pairs for that thread. The thread’s precondition is paired with each of the eight considered conditions and the z-values for each pair is computed. The precondition/condition pair whose z-value was the lowest is determined to be that thread’s (or that precondition’s) “best” condition.

After the three threads have finished considering all eight available conditions, the three z-values (representing the lowest z-value precondition/condition pair from each thread) would be compared. The precondition/condition pair whose z-value was the lowest would be selected as the next splitter node in the decision tree. The splitter node (containing the condition in the precondition/condition pair whose z-value was lowest) would be attached to the node already in the tree containing the precondition from the precondition/condition pair. The number of z-values needed to be calculated on each iteration is given by the following formula:

$$number\_z\_values = number\_preconditions \cdot number\_available\_conditions$$

Creating n threads, independent of the number of preconditions (instead of creating a thread per condition), and distributing the z-value computations uniformly amongst the n threads would speed up this costly function.

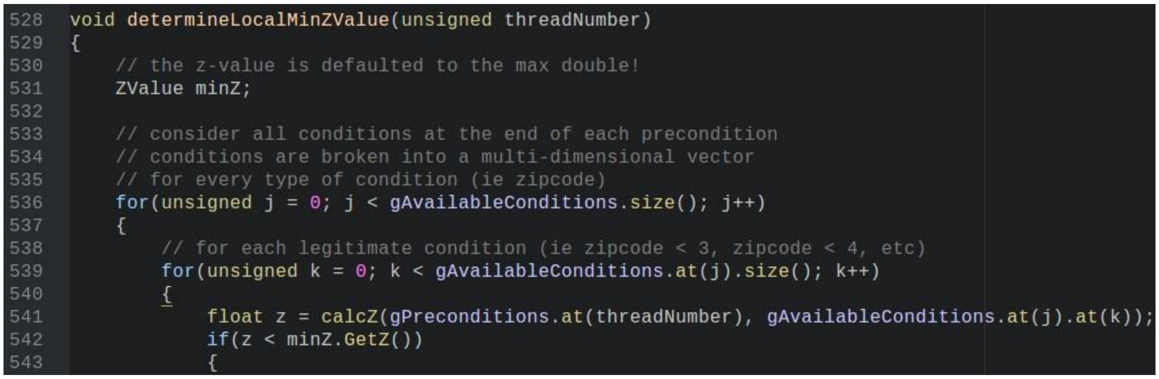
Another improvement that could be made is within the *determineLocalMinZValue* function (which each thread runs). The available conditions are stored in a two-dimensional vector (a C++ data-type provided by the standard template library, or STL, which most closely resembles a stack). The outer dimenson’s length is equal to the number of attributes in the dataset while the inner dimension contains the conditions available concerning its outer dimenson’s attribute:



```
// for each attribute, there is a vector of Conditions
vector< vector<Condition> > gAvailableConditions;
```

**Figure 42: Available Conditions**

The function (*determineLocalMinZValue*) iterates through the available conditions via a nested for-loop:



```
void determineLocalMinZValue(unsigned threadNumber)
{
    // the z-value is defaulted to the max double!
    ZValue minZ;

    // consider all conditions at the end of each precondition
    // conditions are broken into a multi-dimensional vector
    // for every type of condition (ie zipcode)
    for(unsigned j = 0; j < gAvailableConditions.size(); j++)
    {
        // for each legitimate condition (ie zipcode < 3, zipcode < 4, etc)
        for(unsigned k = 0; k < gAvailableConditions.at(j).size(); k++)
        {
            float z = calcZ(gPreconditions.at(threadNumber), gAvailableConditions.at(j).at(k));
            if(z < minZ.GetZ())
            {
```

**Figure 43: Local Minimum Z Value**

This iterative approach could be replaced with a threaded approach. The z-value for each condition could be calculated, in theory, at the same time meaning that the task is parallelizable. The current upper limit of *determineLocalMinZValue* is:

$$O(number\_of\_attributes \cdot available\_conditions)$$

If threads replaced this nested for-loop, the upper limit would be improved to:

$$O(number\_of\_threads \cdot number\_of\_conditions\_to\_consider\_per\_thread)$$

To illustrate this, suppose there were ten attributes and ten conditions per attribute with each z-value computation requiring ten seconds. The current solution would require more than 15 minutes to complete the task:

$$10\ attributes \cdot 10\ conditions\ per\ attribute = 100\ z - value\ computations$$

$$100\ z - value\ computations \cdot 10\ s = 1000\ s\ (16\ minutes\ 40\ seconds)$$

In the proposed threaded solution, suppose there were ten threads. 100 z-values would still need to be computed, but this task would be broken into ten equal parts, distributed amongst the ten threads to be pursued in parallel. Each of the ten threads would be responsible for ten z-value computations which results in only 100 seconds (1 minute 40 seconds):

$10\ attributes\ \cdot\ 10\ conditions\ per\ attribute\ =\ 100\ z-value\ computations$

$100\ z-value\ computation\ /\ 10\ threads\ =\ 10\ z-values\ per\ thread$

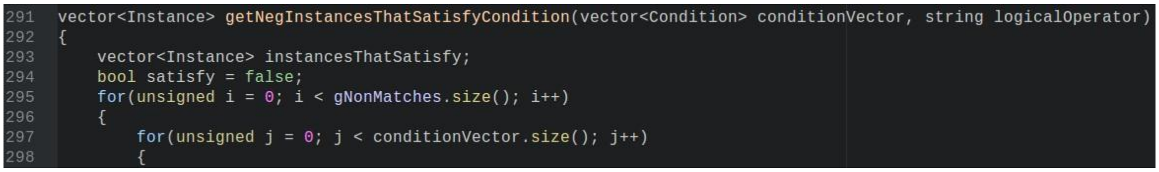
$10\ z-value\ computations\ \cdot\ 10\ s\ =\ 100\ s\ (1\ minute\ 40\ seconds)\ per\ thread$

Since threads run in parallel, the time required for computing all 100 z-values is only 1 minute 40 seconds. It is clear that parallelizing this task could greatly decrease the required time the CSADT algorithm takes.

Buried several layers deep in *determineLocalMinZValue* are two additional functions that could be improved:

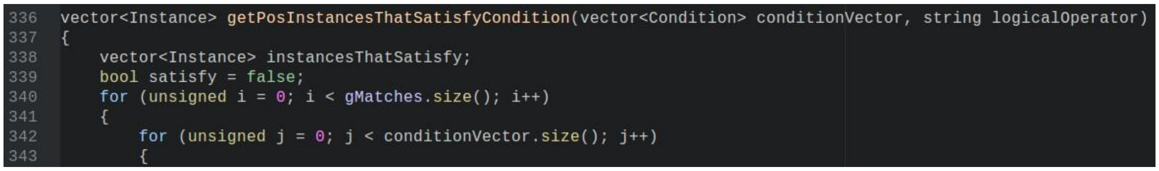
- getNegInstancesThatSatisfyCondition
- getPosInstancesThatSatisfyCondition

These functions are responsible for ascertaining which entities satisfy a given condition and which do not. Below are their implementations:



```
vector<Instance> getNegInstancesThatSatisfyCondition(vector<Condition> conditionVector, string logicalOperator)
{
    vector<Instance> instancesThatSatisfy;
    bool satisfy = false;
    for(unsigned i = 0; i < gNonMatches.size(); i++)
    {
        for(unsigned j = 0; j < conditionVector.size(); j++)
        {
```

**Figure 44: Positive Instances**



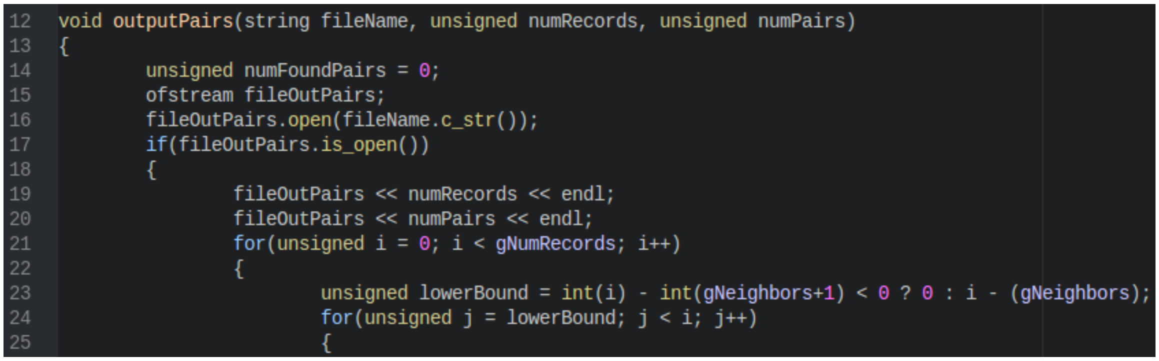
```
vector<Instance> getPosInstancesThatSatisfyCondition(vector<Condition> conditionVector, string logicalOperator)
{
    vector<Instance> instancesThatSatisfy;
    bool satisfy = false;
    for (unsigned i = 0; i < gMatches.size(); i++)
    {
        for (unsigned j = 0; j < conditionVector.size(); j++)
        {
```

**Figure 45: Negative Instances**

Both of the functions contain nested for-loops. As explained earlier, these nested for-loops could be replaced with a threaded solution to drastically increase the speed performance of these two functions.

The last three places that could be improved are similar situations in that an iterative nested for-loop solution could be replaced by a threaded solution. However, improving these areas would not significantly improve the speed of the CSADT algorithm as they are outside of the induction step. Nevertheless, they are explained below.

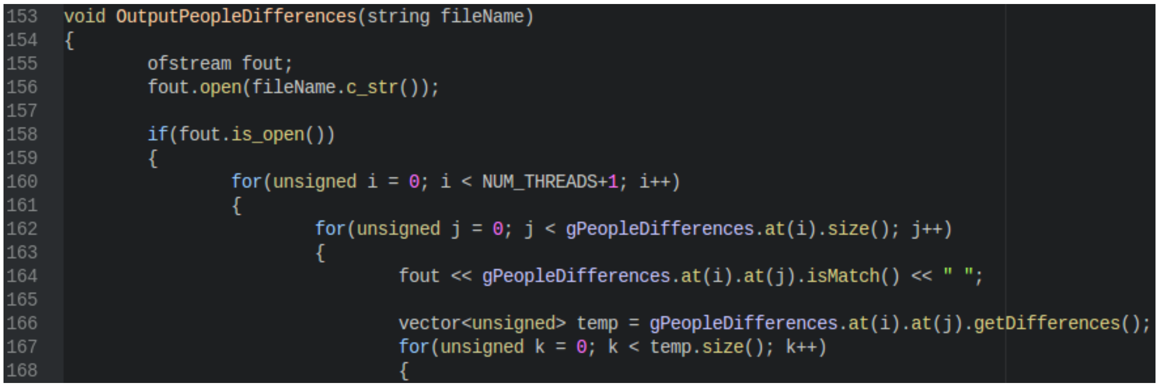
In order to train on sample data, records must be coupled together to make a record-pair. Although the task of generating which record should be compared to which other record was improved by employing a threaded solution, the output of this information has yet to undergo this same procedure as the code snippet demonstrates:



```
void outputPairs(string fileName, unsigned numRecords, unsigned numPairs)
{
        unsigned numFoundPairs = 0;
        ofstream fileOutPairs;
        fileOutPairs.open(fileName.c_str());
        if(fileOutPairs.is_open())
        {
                fileOutPairs << numRecords << endl;
                fileOutPairs << numPairs << endl;
                for(unsigned i = 0; i < gNumRecords; i++)
                {
                        unsigned lowerBound = int(i) - int(gNeighbors+1) < 0 ? 0 : i - (gNeighbors);
                        for(unsigned j = lowerBound; j < i; j++)
                        {
```

**Figure 46: Output Record-Pairs**

After the pairs are known, the “difference” of two person records needs to be constructed. The difference of any two record-pairs is simply a data-structure containing the Levenshtein Distances between each of the records’ attributes. In the snippet below, this nested for-loop approach could be replaced with threads to save computation time.



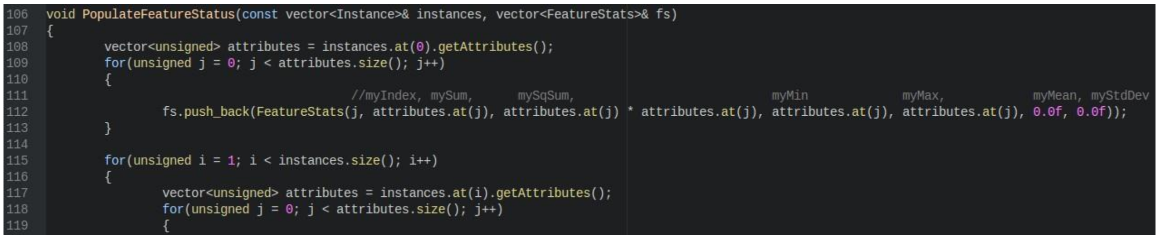
```
void OutputPeopleDifferences(string fileName)
{
        ofstream fout;
        fout.open(fileName.c_str());

        if(fout.is_open())
        {
                for(unsigned i = 0; i < NUM_THREADS+1; i++)
                {
                        for(unsigned j = 0; j < gPeopleDifferences.at(i).size(); j++)
                        {
                                fout << gPeopleDifferences.at(i).at(j).isMatch() << " ";

                                vector<unsigned> temp = gPeopleDifferences.at(i).at(j).getDifferences();
                                for(unsigned k = 0; k < temp.size(); k++)
                                {
```

**Figure 47: Output People Levenshtein Distances**

Lastly, the automatic condition generation code could be improved by replacing its nested for-loop implementation with threads as shown below.



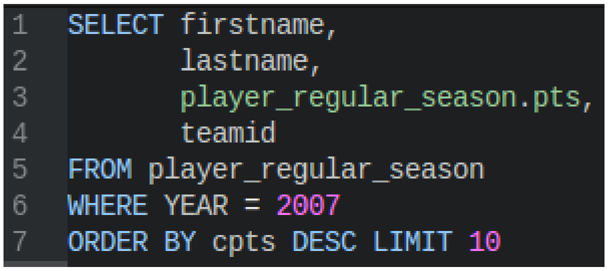
```
void PopulateFeatureStatus(const vector<Instance>& instances, vector<FeatureStats>& fs)
{
        vector<unsigned> attributes = instances.at(0).getAttributes();
        for(unsigned j = 0; j < attributes.size(); j++)
        {
                                    //myIndex, mySum,      mySqSum,                                 myMin             myMax,            myMean, myStdDev
                fs.push_back(FeatureStats(j, attributes.at(j), attributes.at(j) * attributes.at(j), attributes.at(j), attributes.at(j), 0.0f, 0.0f));
        }

        for(unsigned i = 1; i < instances.size(); i++)
        {
                vector<unsigned> attributes = instances.at(i).getAttributes();
                for(unsigned j = 0; j < attributes.size(); j++)
                {
```

**Figure 48: Intelligent Condition Generation**

## 7.3 NBA Improvements

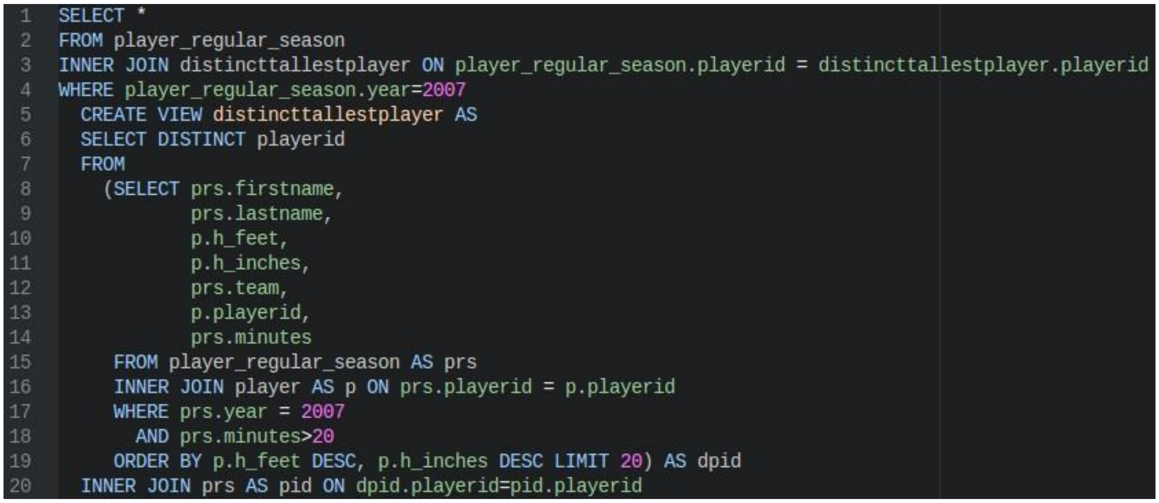
In order to improve the accuracy of the answer to the question posed (or to ask other interesting questions) in the application of the NBA statistical data to the CSADT algorithm, additional conditions need to be created by analyzing the data. For instance, knowing the best ten offensive players in the league could be incorporated as a condition. This data is available.



```
SELECT firstname,
       lastname,
       player_regular_season.pts,
       teamid
FROM player_regular_season
WHERE YEAR = 2007
ORDER BY cpts DESC LIMIT 10
```

**Figure 49: NBA SQL 1**

Perhaps there is some correlation between a team's tallest player and their win/loss record. Again, this information is available:



```
SELECT *
FROM player_regular_season
INNER JOIN distincttallestplayer ON player_regular_season.playerid = distincttallestplayer.playerid
WHERE player_regular_season.year=2007
  CREATE VIEW distincttallestplayer AS
  SELECT DISTINCT playerid
  FROM
    (SELECT prs.firstname,
            prs.lastname,
            p.h_feet,
            p.h_inches,
            prs.team,
            p.playerid,
            prs.minutes
     FROM player_regular_season AS prs
     INNER JOIN player AS p ON prs.playerid = p.playerid
     WHERE prs.year = 2007
       AND prs.minutes>20
     ORDER BY p.h_feet DESC, p.h_inches DESC LIMIT 20) AS dpid
  INNER JOIN prs AS pid ON dpid.playerid=pid.playerid
```

**Figure 50: NBA SQL 2**

With the ability to model such knowledge, it is likely that the accuracy could be improved for the question asked.

## 7.4 Interface

A graphical user-friendly interface would provide easier interaction between the CSADT algorithm and a user. This would allow anyone to upload their data to be processed by the CSADT algorithm.

# REFERENCES

Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. (2007). Duplicate record detection: A survey. *Knowledge and Data Engineering, IEEE Transactions on*, *19*(1).

Annaleen Vanassche, Darek Krzywania, Joris Vaneyghen, Jan Struyf, and Hendrik Blockeel. (2003). First order alternating decision trees. In *Inductive Logic Programming, 13th International Conference, ILP 2003, Szeged, Hungary, Short Presentations* (pp. 116-125).

Bernhard Pfahringer, Geoffrey Holmes, and Richard Kirkby. (2001). Optimizing the induction of alternating decision trees. In *Advances in Knowledge Discovery and Data Mining*. (pp. 477-487). Springer Berlin Heidelberg.

Committee on the Fundamentals of Computer Science: Challenges and Opportunities, National Research Council. (2004). *Computer Science: Reflections on the Field, Reflections from the Field*. National Academies Press.

Geoffrey Holmes, Bernhard Pfahringer, Richard Kirkby, Eibe Frank and Mark Hall. (2002). Multiclass alternating decision trees. In *Machine Learning: ECML 2002* (pp. 161-172). Springer Berlin Heidelberg.

J. R. Quinlan. (1986). Induction of decision trees. *Machine Learning*, *1*(1).

J. R. Quinlan. (1996). Bagging, boosting, and C4.5. In *AAAI/AAI, Vol. 1* (pp. 725-730).

Jan Bohacik, C. Kambhampati, Darryl N. Davis, and John G. F. Cleland. (2013). Alternating decision tree applied to risk assessment of heart failure patients. *Journal of Information Technologies*, *6*(2).

Kuang-Yu Liu, Jennifer Lin, Xiaobo Zhou, and Stephen T. C. Wong. (2005). Boosting alternating decision trees modeling of disease trait information. *BMC Genetics*, *6*(Suppl 1).

Liang Jin, Chen Li, and Sharad Mehrotra. (2003). Efficient record linkage in large data sets. In *Database Systems for Advanced Applications, 2003, (DASFAA). Proceedings, Eighth International Conference on (*pp. 137-146). (IEEE).

Lisa Getoor and Ashwin Machanavajjhala. (2012). Entity resolution: theory, practice & open challenges. *Proceedings of the VLDB Endowment*, 5(12).

Pierre Geurts, Alexandre Irrthum, and Louis Wehenkel. (2009). Supervised learning with decision tree-based methods in computational and systems biology. *Molecular BioSystems, 5*(12).

Steven Salzberg. (1994). Book Review: C4.5: Programs for machine learning by J. Ross Quinlan, Morgan Kaufmann Publishers, Inc., 1993. *Machine Learning*, *16*(3).

Samuel Schulter, Paul Wohlhart, Christian Leistner, Amir Saffari, Peter M. Roth, and Horst Bischof. (1985). Alternating decision forests. In *Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference* (pp. 508-515). (IEEE).

Sanjukta Bhowmick, Victor Eijkhout, Yoav Freund, Erika Fuentes, and David Keyes. (2010). Application of alternating decision trees in selecting sparse linear solvers. In *Software Automatic Tuning*, (pp. 153-173). Springer New York.

Sheng Chen. (2011). The case for cost-sensitive and easy to interpret models in industrial record linkage. *Ninth international workshop on Quality in Databases (QDB).*

V. I. Levenshtein (1999). Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics doklady 10*(8).

Yoav Freund and Llew Mason. (1999). The alternating decision tree learning algorithm. In *Machine Learning, Proceedings, 16th International Conference on, 99, (pp. 124-133). (ICML).*