A Time Complexity Analysis for our Comps Group Implementation of Aequitas Fully Directed.

Michael Worrell, Yemi Shin, Yunping Wang, and Juanito Zhang Yang
Department of Computer Science, Carleton College, 300 North College Street, Northfield, MN, 55057
(Dated: March 16, 2022)

For our Comps this trimester, we did research into and worked to implement a modified version of the machine learning fairness algorithm “Aequitas.” For this analysis write up, we will be discussing the time complexity of our own implementation of the algorithm presented in the original Aequitas paper.

Time complexity analysis paper preface: What to expect, and Why.

The original Aequitas paper did not include a time complexity analysis of their own algorithm. We therefore would like to present our own time complexity analysis of our own implementation.

It is also important that we record the time complexity analysis of our own implementation, because we made fundamental changes to the original algorithm for the sake of runtime optimization that drastically changed the time complexity that would have resulted from perfectly implementing the original Aequitas paper.

When measuring the time complexity, we are measuring it in terms of Big O notation. However, we will be measuring the time complexity of our algorithm in the *average* case. The reason we are doing this is because several of the data structures we are using, such as sets (which in Python are implemented as dictionaries) have a time complexity of $O(1)$ in the average case, but a time complexity of $O(n)$ in the worst case.

To better represent what elements of Aequitas are likely to take up the largest number of computations, we will treat the average case as the standard for data structures, while keeping everything else in terms of the worst case.

This time complexity writeup will only discuss the time complexity of Aequitas Fully Directed. One reason for this is because out of all of the versions of the code we implemented, Aequitas Fully directed is the one we spent the most time optimizing, testing, and fixing throughout our comps project, and as a result is more in tune with the spirit of our Comps project.

Variable definitions used throughout this time complexity analysis

Here are variables that will be used throughout the remainder of this time complexity analysis. For any additional questions, read the

- g = *global_iteration_limit,* as defined in our Aequitas configuration file.
- l = *local_iteration_limit,* as defined in our Aequitas configuration file.
- S = The set of sensitive features
- Si = The i_th sensitive feature
- Ri = The set of values that are possible for the sensitive feature S_i
- $O(model.predict)$ = The time it takes to return classifications for an input in the machine learning model.
- P = The set of parameters in any given input.

Time Complexity Analysis: Helper function & common operations time complexities.

While the *Aequitas_fully_directed_sklearn* algorithm works using custom functions, it also relies upon previously designed functions and operations that have their own previously established time complexities. In this part, we will record their time complexities for reference when calculating the *Aequitas_fully_directed_sklearn* time complexity.

Time Complexity Analysis, Part 1: NumPy Python Library

The proof of concept implementation that we built our version of Aequitas from used NumPy (Numerical Python) functions. The NumPy library is what our implementation uses in the processing of multidimensional array objects, which Aequitas uses to store and manipulate inputs. The NumPy functions used in *Aequitas_fully_directed_sklearn* and their time complexities are as follows:

- np.asarray(input) has a time complexity of $O(|P|)$. This is true because this function is copying the data into a new array object, which runs in $O(n)$ time.
- np.delete(input, index) has a time complexity of $O(|P|)$. This works in an essentially identical way to removing items from an array, which runs in $O(n)$ time.
- np.reshape(input) has a time complexity of $O(1)$. This is because reshape appears to change how the stored data is interpreted, not how the data itself is stored.

Time Complexity Analysis, Part 2: Python Set Operations

Our Aequitas implementation uses sets to help keep track of which inputs were discovered as being discriminatory. The Python set operations used and their time complexities are as follows:

- Adding an item to a set in python has a time complexity of $O(1)$.
- Checking whether an item is present in a set in python has a time complexity of $O(1)$.

Time Complexity Analysis, Part 3: Python List Operations

Our Aequitas implementation uses sets to help keep track of which inputs were discovered as being discriminatory. The Python list operations used in Aequitas_fully_directed_sklearn and their time complexities are as follows:

- Appending an item to a python list has a time complexity of $O(1)$.
- Retrieving an item from a python list has a time complexity of $O(1)$.

Time Complexity Analysis, Part 4: Miscellaneous functions

There are also miscellaneous functions used throughout several important functions. The time complexities of these functions that are used in Aequitas_fully_directed_sklearn are as follows:

- random.randint(0, n) has a time complexity of $O(1)$.
- random.choice(array *or* list) has a time complexity of $O(1)$.
- Fully_Direct.normalize_probability has a time complexity of $O(|P|)$.

Time Complexity Analysis, Part 5: Basinhopping

Basinhopping, as we use it in our implementation, has the following parameters:

- func (function to put under test)
- x0 (the starting inputs for the basinhopping process)
- stepsize
- take_step (algorithm determining how perturbation takes a step, or a constant value)
- minimizer_kwargs (minimizer used)
- niter (number of iterations)

The basinhopping function works by taking some input, deciding how large of a step to take and where to take the step, perturbing the input by that step amount, using the function on the perturbed inputs using the minimizer to receive a score it is attempting to minimize, and then repeating until conditions inherent to the minimizer are met, up to a maximum of *niter* times.

For each call to basinhopping in our algorithm, stepsize is constant. So, the only parameters that we need to consider for runtime purposes are func, x0, take_step, minimizer_kwargs, and niter. From this, we see that the time complexity of the basin hopping algorithm is

$O(niter * (O(func) + O(take\_step) + O(minimizer\_kwargs)))$.

Time Complexity Analysis: Aequitas_fully_directed_sklearn.

Aequitas_fully_directed_sklearn can be split into two parts. The first of these parts performs global discovery to find discriminatory inputs, and the second of these parts performs local discovery to find disriminatory inputs in the vicinity of discrimnatory inputs found during global discovery.

Aequitas_fully_directed_sklearn, Part 1: Global discovery time complexity

For the global discovery portion of the Aequitas algorithm we implemented, Aequitas runs the basinhopping function one time. This means that the global discovery algorithm has a time complexity of $O(basinhopping)$. The next thing to do is calculate the time complexity of basinhopping during this global discovery process.

Aequitas_fully_directed_sklearn, Part 1, Subsection 1: basinhopping

Here, the basin hopping parameters that are relevant to the time complexity are:

- func ← Fully_Direct.evaluate_global
- take_step ← Fully_Direct.global_discovery
- minimizer_kwargs ← minimizer (more specifically, a L-BFGS-B method minimizer).
- niter ← g

Using the definition from *Time Complexity Analysis, Part 5,* we find that the time complexity for each runthrough of the basin hopping algorithm during local search becomes

$O(niter * (O(func) + O(take\_step) + O(minimizer\_kwargs)))$

Expanding upon the time complexity equation we found in "*Aequitas_fully_directed_sklearn, Part 2",* the time complexity of global discovery becomes

$$O(g*(O(fully\_direct.evaluate\_global)+O(fully\_direct.global\_discovery)+O(minimizer)))$$.

We will now describe the time complexity of each relevant function that is used within basinhopping for *aequitas_fully_directed_sklearn* global discovery, and will then bring them together after each function has been evaluated in detail.

Aequitas_fully_directed_sklearn, Part 1, Subsection 2: Fully_Direct.global_discovery

First, global discovery generates a random seed, which has a time complexity of $O(1)$. Then, the function random.randint(low, high) repeats for each parameter in the provided input, and uses low and high corresponding to the range of possible values for the chosen parameter. Since random.randint(low, high) has a time complexity of $O(1)$, this means that generating a new global input adds $O(|P|)$ to the time complexity of *Fully_Direct.global_discovery*.

Lastly, Setting the sensitive feature value for the new input to 0 has an average case time complexity of $O(1)$. This means that this function has a time complexity of $O(|P|)$.

Aequitas_fully_directed_sklearn, Part 1, Subsection 3: Fully_Direct.evaluate_global

For the first part of this function, the function makes calls to np.asarray, np.reshape, and model.predict are made, as well as adds to a set and checks whether an item is already present in a set. This has an average time complexity of $O(O(model.predict) * |P|)$. If the original input is present in the global_discovery discriminatory input set, the function ends here.

If the original input is not present in the global_discovery discriminatory input set, the function loops through the $|R_{input}|$ inputs that differ only in the given sensitive feature. For each new input, calls to np.asarray, np.reshape, and model.predict are made. This has an average time complexity of $O(O(model.predict) * |P|)$ for each new input tested. This means that the entirety of this internal loop has a time complexity of$O(|R_{input}| * O(model.predict) * |P|)$.

This continues until a discriminatory input is discovered, or until the function proves that the original input is not discriminatory for the sensitive feature selected. In the event that a discriminatory input is discovered, adding the original input to the global_discovery discriminatry input set and appending it to the global_discovery input list has an average time complexity of $O(1)$.

All of this tells us that our **Fully_Direct.evaluate_global function** has three cases to consider.

- In the first case, the original input is already present in the global_discovery discriminatory input set. This case has a time complexity of

  $O(O(model.predict) * |P|)$.

- In the second case, the original input is not present in the global_discovery discriminatory input set, and is discovered to be discriminatory in regards to the selected sensitive feature. This case has a time complexity of $O(O(model.predict) * |P|)) + O(|R_{input}| * O(model.predict) * |P|) + O(1))$, which can be simplified to $O(|R_{input}| * O(model.predict) * |P|)$.

- In the third case, the original input is not present in the global_discovery discriminatory input set, and is discovered to *not* be discriminatory in regards to the selected sensitive feature. This case has a time complexity of

  $O(O(model.predict) * |P|)) + O(|R_{input}| * O(model.predict) * |P|)$,

  which can be simplified to $O(|R_{input}| * O(model.predict) * |P|)$.

Putting all of this together, this means that the function *fully_direct.evaluate_global* has a time complexity of $O(|R_{input}| * O(model.predict) * |P|)$.

Aequitas_fully_directed_sklearn, Part 1, Subsection 4: minimizer

The minimizer Aequitas uses is L-BFGS-B. According to Dewi Retno Sari Saputro and Purnami Widyaningsih, that particular minimizer algorithm has a time complexity of $O(mn)$, where *m* is the number of inputs (in our case, the number of parameters for the input to test), and *n* is the maximum number of variable metric corrections used to define the limited memory matrix. However, the value of *n* is locked to a default value for the scipy.optimize.minimize("L-BFGS-B") function. This means that the time complexity of the minimizer function is $O(|P|)$.

Aequitas_fully_directed_sklearn, Part 1, Subsection 5: Bringing everything together

Now that we know the time complexity for all of the functions and methods utilized in the global discovery stages of our implementation, we can put them together. The new time complexity can be expanded and then simplified as follows:

$\rightarrow O(niter * (O(func) + O(take\_step) + O(minimizer\_kwargs)))$

$\rightarrow O(g*(O(Fully\_Direct.evaluate\_global)+O(Fully\_Direct.global\_discovery)+O(minimizer)))$

$\rightarrow O(g * (O(|R_{input}| * O(model.predict) * |P|) + O(P) + O(|P|)))$

$\rightarrow O(g * O(|R_{input}| * O(model.predict) * |P|))$

$\rightarrow O(g * |R_{input}| * |P| * O(model.predict))$

Aequitas_fully_directed_sklearn, Part 2: Global discovery time complexity

For the local discovery portion of the Aequitas algorithm we implemented, Aequitas runs the basinhopping function once *per discriminatory input found during global discovery*. This means that the local discovery algorithm has a time complexity of $O(g * O(basinhopping))$. The next thing to do is calculate the time complexity of basinhopping during this global discovery process.

Aequitas_fully_directed_sklearn, Part 2, Subsection 1: basinhopping

Here, the basin hopping parameters that are relevant to the time complexity are:

- func ← Fully_Direct.evaluate_local
- take_step ← Fully_Direct.local_perturbation
- minimizer_kwargs ← minimizer (more specifically, a L-BFGS-B method minimizer).
- niter ← l

Using the definition from *Time Complexity Analysis, Part 5,* we find that the time complexity for each runthrough of the basin hopping algorithm during local search becomes

$O(niter * (O(func) + O(take\_step) + O(minimizer\_kwargs)))$

Expanding upon the time complexity equation we found in *Aequitas_fully_directed_sklearn, Part 2,* the time complexity of local discovery becomes

$$O(local\_iteration\_limit * (O(fully\_direct.evaluate\_local) + O(fully\_direct.local\_perturbation) + O(minimizer)))$$
.

We will now describe the time complexity of each relevant function that is used within basinhopping for *aequitas_fully_directed_sklearn* local discovery, and will then bring them together after each function has been evaluated in detail.

<u>Aequitas_fully_directed_sklearn, Part 2, Subsection 2: Fully_Direct.evaluate_local</u>

For the first part of this function, the function makes calls to np.asarray, np.reshape, and model.predict are made, as well as adds to a set and checks whether an item is already present in a one of two separate sets, one from the global discovery step, and one from the local discovery step. This has an average time complexity of $O(O(model.predict) * |P|)$. If the original input is present in either the global_discovery discriminatory input set or the local_discovery discriminatory input set, the function ends here.

If the original input is not present in either discriminatory input set, the function loops through the $|R_{input}|$ inputs that differ only in the given sensitive feature. For each new input, calls to np.asarray, np.reshape, and model.predict are made. This has an average time complexity of $O(O(model.predict) * |P|)$ for each new input tested. This means that the entirety of this internal loop has a worst case time complexity of $O(|R_{input}| * O(model.predict) * |P|)$.

This continues until a discriminatory input is discovered, or until the function proves that the original input is not discriminatory for the sensitive feature selected. In the event that a discriminatory input is discovered, adding the original input to the local_discovery discriminatry input set and appending it to the local_discovery input list has an average time complexity of $O(1)$.

All of this tells us that our ***Fully_Direct.evaluate_local* function** has three cases to consider.

- In the first case, the original input is already present in either of the global_discovery or local_discovery discriminatory input sets. This case has a time complexity of

  $O(O(model.predict) * |P|)$.

- In the second case, the original input is not present in either discriminatory input set, and is discovered to be discriminatory in regards to the selected sensitive feature. This case has a time complexity of $O(O(model.predict) * |P|)) + O(|R_{input}| * O(model.predict) * |P|) + O(1))$, which can be simplified to $O(|R_{input}| * O(model.predict) * |P|)$.

- In the third case, the original input is not present in the either discriminatory input set, and is discovered to *not* be discriminatory in regards to the selected sensitive feature. This case has a time complexity of

  $O(O(model.predict) * |P|)) + O(|R_{input}| * O(model.predict) * |P|)$,

  which can be simplified to $O(|R_{input}| * O(model.predict) * |P|)$.

Putting all of this together, this means that the function *Fully_Direct.evaluate_local* has a time complexity of $O(|R_{input}| * O(model.predict) * |P|)$.

<u>Aequitas_fully_directed_sklearn, Part 2, Subsection 3: Fully_Direct.evaluate_input</u>

This particular function behaves identically to the function *Fully_Direct.evaluate_local*, with a few notable exceptions. When it runs, it does not check or add to any discriminatory input sets. Rather, it returns a different numerical result once the input has been identified as discriminatory or non-discriminatory. This changes the cases to consider as follows:

- The first case no longer exists, so there is no time complexity to consider.
- In the second case, the original input is discovered to be discriminatory in regards to the selected sensitive feature. This case has a time complexity of $O(|R_{input}| * O(model.predict) * |P|)$.
- In the third case, the original input is discovered to *not* be discriminatory in regards to the selected sensitive feature. This case has a time complexity of

  $O(|R_{input}| * O(model.predict) * |P|)$.

Putting all of this together, this means that the function *Fully_Direct.evaluate_input* has a time complexity of $O(|R_{input}| * O(model.predict) * |P|)$.

<u>Aequitas_fully_directed_sklearn, Part 2, Subsection 4: Fully_Direct.local_perturbation</u>

First, this function copies input bounds, one per parameter index, to a new array. This part has a time complexity of $O(|P|)$. Then, it performs between 2 and 3 random.choice() operations that each have a time complexity of $O(1)$, which since means the time complexity for *fully_direct.local_perturbation* is still at $O(|P|)$.

After this, *fully_direct.local_perturbation* runs *fully_direct.evaluate_input*, which has a time complexity of $O(|R_{input}| * O(model.predict) * |P|)$.

Then, it performs constant time operations until it normalizes the probability for the perturbed input, which has a time complexity of $O(|P|)$.

Bringing this all together, we find that *Fully_Direct.local_perturbation* has a time complexity of $O(|R_{input}| * O(model.predict) * |P|)$.

Aequitas_fully_directed_sklearn, Part 2, Subsection 5: minimizer

The minimizer function used for local discovery is identical to the minimizer function used for global discovery (see *Aequitas_fully_directed_sklearn, Part 1, Subsection 4)*. This means that the time complexity of the minimizer is $O(|P|)$.

Aequitas_fully_directed_sklearn, Part 2, Subsection 6: Bringing everything together

Now that we know the time complexity for all of the functions and methods utilized in the local discovery stages of our implementation, we can put them together. The new time complexity can be expanded and then simplified as follows:

$\rightarrow O(g * O(basinhopping))$

$\rightarrow O(g * O(niter * (O(func) + O(take\_step) + O(minimizer\_kwargs))))$

$\rightarrow O(g*O(l*(O(Fully\_Direct.evaluate\_local)+O(Fully\_Direct.local\_perturbation)+ O(minimizer))))$

$\rightarrow O(g*O(l*(O(|R_{input}|*O(model.predict)*|P|)+O(|R_{input}|*O(model.predict)* |P|) + O(|P|))))$

$\rightarrow O(g * O(l * O(|R_{input}| * O(model.predict) * |P|)))$

$\rightarrow O(g * l * |R_{input}| * |P| * O(model.predict))$

Aequitas_fully_directed_sklearn, Part 3, Subsection 1: Combining both parts.

Now we can look at the time complexities of global discovery and local discovery alongside each other. Combining both to the same equation, the time complexity becomes

$O(g*|R_{input}|*|P|*O(model.predict))+O(g*l*|R_{input}|*|P|*O(model.predict))$,

which can be simplified to a time complexity of $O(g * l * |R_{input}| * |P| * O(model.predict))$.

Aequitas_fully_directed_sklearn, Part 3, Subsection 2: Time complexity with multiple sensitive features

In our implementation, we allow the user to test multiple sensitive features at once. However, we do it slightly differently than the original Aequitas paper suggests we should. We chose to do this in order to improve runtime to make the site easier to use and more accessible.

To do this, we run the algorithm all over again on different sensitive features. This translates into a time complexity of

$$O\left(\sum_{\forall i:S_i\in S} g * l * |P| * O(model.predict) * |R_i|\right).$$

This is great, but it could look cleaner. After pulling all of the constants out of the summation, the time complexity for Aequitas Fully Directed when testing for multiple sensitive features simultaneously in our implementation can be rewritten in a way we believe is more informative.

$$O\left(g * l * |P| * O(model.predict) * \left(\sum_{\forall i:S_i\in S} |R_i|\right)\right)$$

We present to you, the worst case time complexity for our implementation of the Aequitas Algorithm that allows for multiple non-binary sensitive features while making runtime optimizations.

Citations Page

Udeshi, S., Arora, P., & Chattopadhyay, S. (2018). Automated Directed Fairness Testing. ArXiv. https://doi.org/10.48550/ARXIV.1807.00468

Harris, C.R., Millman, K.J., van der Walt, S.J. et al. Array programming with NumPy. Nature 585, 357–362 (2020). DOI: 10.1038/s41586-020-2649-2..

Van Rossum, G. (2020). The Python Library Reference, release 3.8.2. Python Software Foundation.

Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E.A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. (2020) SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. Nature Methods, 17(3), 261-272.

Saputro, D. R. S., & Widyaningsih, P. (2017). Limited memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) method for the parameter estimation on geographically weighted ordinal logistic regression model (GWOLR). In AIP Conference Proceedings. THE 4TH INTERNATIONAL CONFERENCE ON RESEARCH, IMPLEMENTATION, AND EDUCATION OF MATHEMATICS AND SCIENCE (4TH ICRIEMS): Research and Education for Developing Scientific Attitude in Sciences And Mathematics. Author(s). https://doi.org/10.1063/1.4995124