# Aplicación de técnicas de machine learning para predecir el consumo de gas en transacciones Ethereum

## Application of machine learning techniques to predict gas consumption in Ethereum transactions


Universidad Complutense de Madrid
Facultad de Informática

Trabajo de fin de grado del Grado en Ingeniería Informática

*Carlos Barona Dorado*
*Jose Luis Cantarero Delgado*

Dirigido por:
Elvira Albert Albiol
Pablo Gordillo Alguacil

Curso 2023-2024

# Abstract

For the execution of any transaction in Ethereum, the amount of gas that can be consumed by the transaction during its execution must be specified, as if it were fuel. Numerous transactions fail every day because of setting an amount of gas lower than the one required for their correct execution. The Solidity compiler is able to infer an upper bound on the gas consumption of a contract is public methods. However, sometimes the compiler returns "infinity" as a value, due to the complexity of the analyzed methods, resulting in unhelpful information for the user. The lack of an informed upper bound, gives rise to these possible errors due to the ignorance of the the amount of gas that will be consumed by the execution. Thus, taking into account that the amount of gas consumed by a transaction is specified by a well-defined cost model, it is therefore predictable and does not depend on market fluctuations or network congestion.

The objective of this TFG is to apply machine learning techniques to infer the gas consumption required to execute a given transaction. For this purpose, different machine learning models have been developed and evaluated for each public function of the 100 most used contracts deployed on the blockchain since 2023, taking as a parameter the transaction input. At the same time, an analysis of the bytecode of a simple contract has been conducted for the following functions with distinguished costs in the evaluations: constant, based on the input parameter, based on the size of the input, and accessing storage. A cost function has been derived from the control flow graph.

Writing to storage involves an uncertainty that makes gas consumption unpredictable when considering only the values of the arguments used to call the transaction, since it depends on the state of the storage at the time the transaction is executed. On the other hand, for cases where the storage is not involved, it is possible to approximate a cost function from the input parameters since they do not depend on any external agent.

## Keywords:

Ethereum, Ethereum Virtual Machine, Smart Contracts, Blockchain, Gas, Machine Learning, Bytecode, Control-Flow Graph

# Resumen

Para la ejecución de cualquier transacción en Ethereum, se debe establecer la cantidad de gas que puede ser consumido por la transacción durante su ejecución, como si de combustible se tratase. Diariamente numerosas transacciones fallan por establecer una cantidad de gas inferior a la necesaria para la correcta ejecución. El compilador de Solidity es capaz de inferir una cota superior sobre el consumo de gas de los métodos públicos de un contrato. Sin embargo, en ocasiones el compilador devuelve "infinito" como valor, debido a la complejidad de los métodos analizados, resultando una información poco útil para el usuario. La falta de una cota superior informada, da lugar a estos posibles errores por el desconocimiento de la cantidad de gas que consumirá dicha ejecución. Teniendo en cuenta que la cantidad de gas que consume de una transacción está especificado por un modelo de coste bien definido, es por tanto predecible y no depende de la fluctuación del mercado, ni de la saturación de la red.

El objetivo de este TFG es aplicar técnicas de *machine learning* para inferir el consumo de gas necesario para ejecutar una determinada transacción. Para ello, se han desarrollado y evaluado distintos modelos de aprendizaje automático para cada función pública de los 100 contratos más ejecutados desde 2023, tomando como parámetro el input de la transacción. A su vez se ha realizado un análisis del bytecode de un contrato simple para las funciones con los diferentes costes distinguidos en las evaluaciones: constante, en función del parámetro de entrada, en función del tamaño de la entrada y accediendo al storage. Obteniendo una función de coste a partir del grafo de control de flujo.

La escritura en storage supone una incertidumbre que propicia que el consumo de gas no sea predecible únicamente considerando los valores de los argumentos con los que se ha llamado a la transacción, pues depende del estado del storage en el momento de ejecutar de la transacción. Por otro lado, para los casos en los que no interviene el storage es posible aproximar una función de coste a partir de los parámetros de entrada, no dependiendo de ningún agente externo.

## Palabras Clave:

Ethereum, Ethereum Virtual Machine, Contratos Inteligentes, Cadena de Bloques, Gas, Aprendizaje Automático, Bytecode, Grafo de Control de Flujo

# Agradecimientos

A los Profesores Elvira Albert y Pablo Gordillo, por haber sido el motor principal del trabajo; por motivarnos a enfrentar nuestras barreras de conocimiento; por su confianza, tiempo, paciencia y dedicación para que nuestro esfuerzo tuviese los mejores resultados.

A nuestros compañeros, por acompañarnos de la mejor manera posible durante el transcurso de la carrera y hacer de sus días más divertidos.

A nuestras familias y amigos, por su cariño y confianza, pero sobretodo por su apoyo incondicional. Gracias por acompañarnos en esta travesía, por celebrar nuestros éxitos como si fueran propios y por estar presentes en cada paso del camino.

Y en especial, agradecernos el uno al otro, por estar ahí en los momentos difíciles; por ayudarnos e incentivarnos a ser mejores personas; y por las buenos recuerdos que hemos creado juntos y que nunca olvidaremos. Esta experiencia no habría sido la misma sin el apoyo mutuo y el compromiso compartido que hemos demostrado.

Finalmente, estos cuatro años son una prueba del esfuerzo colectivo y colaboración que nos ha acompañado durante este viaje. A todos los que de alguna manera han contribuido a este logro, les extendemos nuestro más sincero agradecimiento.

# Índice general

# Índice de figuras

# Capítulo 1

# Introducción y objetivos

## 1.1. Contexto

En la era digital actual, los avances tecnológicos han transformado de forma notoria la manera en la que interactuamos con lo que nos rodea. Dos de los campos que más han avanzado en esta coyuntura, y por ende más impacto han tenido han sido el *Machine Learning* y la tecnología *Blockchain*[1].

La evolución digital ha otorgado acceso a una cantidad masiva de datos. El procesado y estudio correcto de estos datos permiten obtener tendencias y predecir resultados. En este contexto el *Machine Learning*, una rama de la inteligencia artificial, ha demostrado ser uno de los pilares tecnológicos más importantes en la evolución de sistemas capaces de aprender gracias al acceso a grandes volúmenes de datos, siendo indispensable el aumento en velocidad de cómputo. Esta tecnología se emplea en múltiples sectores por su capacidad para predecir patrones complejos, como por ejemplo: en diagnósticos médicos, detección de fraude y robótica entre otros.

Como todo gran cambio requiere tener la capacidad de adaptarse, la era digital, en particular, ha puesto en entredicho los modelos centralizados. Estos modelos son susceptibles de manipulaciones y ataques cibernéticos, que con el aumento en la capacidad de cómputo suponen una gran amenaza. En respuesta, las redes *blockchain* (cadena de bloques) surgen como una tecnología disruptiva que promete revolucionar la forma en la que se gestionan las transacciones digitales, demostrando ser un un claro competidor de la banca convencional.

A diferencia de los sistemas tradicionales, que almacenan datos en servidores centrales, la *blockchain* es descentralizada, es decir, distribuye los datos a través de una red. Asimismo, las transacciones son transparentes y pueden ser auditadas por cualquier usuario, reduciendo así el coste al eliminar la figura del intermediario y aumentar la eficiencia mediante procesos más automatizados y transparentes[1].

La inmutabilidad es otro concepto clave para contextualizar este modelo, pues las transacciones una vez añadidas a la cadena de bloques, no pueden ser alteradas o borradas, garantizando así la integridad de los registros, sin necesidad de una autoridad centralizada[1].

## 1.2. Blockchain de Ethereum

*Ethereum*[2] es una de las plataformas de cadena de bloques más populares, distinguida por su capacidad para ir más allá de las simples transacciones, como las que realiza *Bitcoin*[3], y permitir la ejecución de código a través de una máquina virtual *cuasi-Turing-completa*, la *Ethereum Virtual Machine* –conocida como ***EVM***–, descrita en el *Yellow Paper*[4]. Esto hace de *Ethereum* un entorno computacional completo y descentralizado que ha abierto nuevas posibilidades permitiendo la ejecución de programas, conocidos como contratos inteligentes (*smart contracts*).

> *What Ethereum intends to provide is a blockchain with a built-in fully fledged Turing-complete programming language that can be used to create çontracts"that can be used to encode arbitrary state transition functions, allowing users to create any of the systems described above, as well as many others that we have not yet imagined, simply by writing up the logic in a few lines of code.*
>
> —*Vitalik Buterin, inventor of Ethereum [5]*

Es pionera en introducir conceptos como ***gas***, siendo la unidad de medida para el coste computacional propio de la ejecución del contrato inteligente. Así, el *gas* nace por la necesidad de calcular el coste de cada ejecución, pues la implementación de una máquina virtual *Turing-completa* que permita la ejecución de distintas instrucciones, resulta en que no todas las ejecuciones consuman lo mismo. El modelo de coste del *gas* será explicado en la Sección 1.2.4, siendo indispensable la comprensión de este para el desarrollo y seguimiento del presente proyecto.

### 1.2.1. Contratos inteligentes

El concepto de ***smart contract*** fue definido por Nick Szabo, en 1996 [6]. En particular, en *Ethereum*, representa un programa que se ejecuta en la *blockchain*. Así, un conjunto de código y datos que se almacenan en una cuenta de *Ethereum*, pueden ser objetivo de transacciones pero no están controladas por un usuario, actúan de acuerdo al código, estableciendo reglas que automáticamente se ejecutan.

Las variables globales del contrato inteligente residen en el ***storage***, almacenamiento distribuido persistente perteneciente a la *blockchain*. El *storage* está estructurado en un diccionario donde cada clave referencia a un *slot*, conjunto de 32 bytes donde se almacenan las variables.

El ***ABI***, siglas de "*Application Binary Interface*", es el estándar que define cómo se interactúa con el contrato. En él, se detalla las funciones disponibles, sus argumentos, los tipos de datos y cómo deben ser codificados para la interacción con el contrato. Cada función se identifica mediante la signatura, una codificación de los cuatro primeros bytes usando la función *hash Keccak-256* sobre el nombre de la función y el tipo de sus argumentos.

A continuación, se muestra la construcción de la signatura para una función bien conocida de interfaz *ERC20*, la función "transfer(address _to, uint256 _value)" a la que se le quitan los nombres de las variables y los espacios entre los paréntesis:

$$Keccak\text{-}256(transfer(address, uint256)) = 0xA9059CBB\ldots$$

Al coger del resultado anterior los cuatro primeros bytes (8 caracteres en hexadecimal) obtenemos la signatura resultante para esa función, signatura la cual se tendrá que referenciar en el *input* de una transacción para poder ejecutarla.

Numerosos contratos implementan el patrón *proxy*, empleando un contrato intermedio, con la declaración de funciones, donde delegan las llamadas. Esta implementación es utilizada para separar los datos de la lógica de negocio. Esto supone un problema a la hora de procesar el *input* de la transacción, pues el *ABI* del *proxy contract*, no recoge las funciones que define el contrato de "implementación", llamando a la función particular *fallback* que se ejecuta cuando se llama a una función no declarada.

### 1.2.2. Transacciones

Las transacciones son el elemento mediante el cual los usuarios interactúan con la red, actualizando el estado de esta. En *Ethereum* se distinguen tres tipos diferentes de transacciones: regular (una transacción desde una cartera a otra), creación de contrato y ejecución de contrato.

Las transacciones poseen los siguientes campos comunes:

| Columna | Descripción |
|---|---|
| type | EIP-2718 transaction type. |
| nonce | A scalar value equal to the number of transactions sent by the sender. |
| gasLimit | A scalar value equal to the maximum amount of gas that should be used in executing this transaction. This is paid up-front, before any computation is done and may not be increased later. |
| to | The 160-bit address of the message calls recipient or, for a contract creation transaction, ∅ |
| value | A scalar value equal to the number of Wei to be transferred to the message call's recipient or, in the case of contract creation, as an endowment to the newly created account. |

Tabla 1.1: Campos comunes de las transacciones en *Ethereum* [4]

Las transacciones de creación de contrato, independientemente del tipo de transacción, contienen un campo *init* con un array de bytes ilimitado con las instrucciones de *EVM* para inicializar el contrato. En cambio, las transacciones de llamada a contratos, poseen un campo *data*, un array de bytes ilimitado para especificar el *input* de la llamada. En particular, el proyecto se centra en las segundas[4].

Los primeros cuatro bytes del campo *input_data* especifican la función que se está llamando. Este valor hace referencia al concepto definido con anterioridad de signatura.

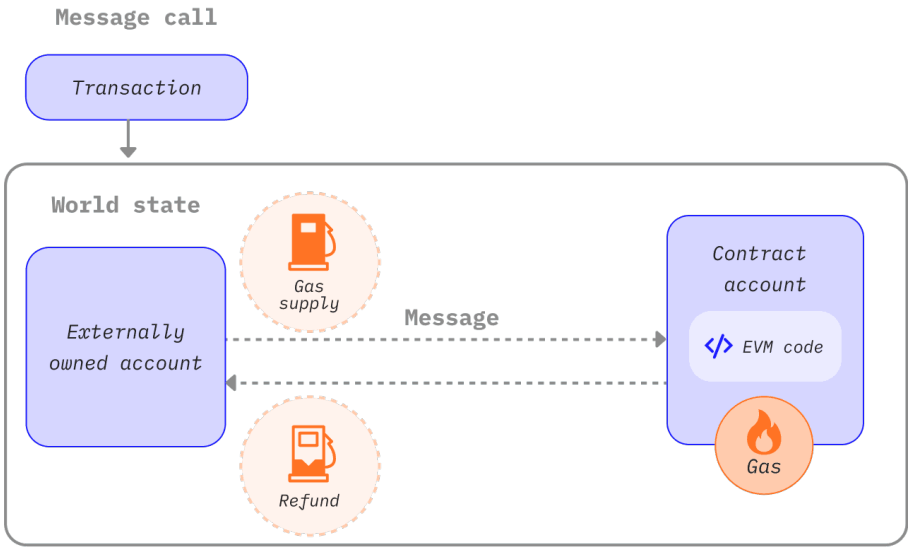



Figura 1.1: Diagrama de transacción *Ethereum*[7]

Para que una transacción se realice de forma satisfactoria la cantidad de *gas* enviado por el usuario debe ser mayor o igual al *gas* necesario para ejecutar las instrucciones pertinentes. Así, la cantidad de *gas* no utilizada será reembolsada al usuario.

### 1.2.3. Bloques

Un bloque en la *blockchain* de *Ethereum* es una colección de transacciones que han sido verificadas y consolidadas en un grupo. Cada bloque además tiene el *hash* del bloque anterior que garantiza la seguridad y consistencia de la *blockchain*.

La interconexión de los bloques a través del *hash* del bloque anterior forma una cadena continua y segura. Cada nuevo bloque añadido a la cadena confirma y respalda aún más la integridad, fortaleciendo la red y asegurando que funcione como una plataforma descentralizada y confiable.

Para validar un bloque y añadirlo a la cadena de bloques *Ethereum* utiliza la conocida como Prueba de participación (***Proof of Stake***, ***PoS***) [8], donde los usuarios bloquean parte de sus fondos para tener el derecho de validar transacciones. De este modo, los validadores son elegidos al azar, en función de la cantidad de tokens que tienen bloqueados.

### 1.2.4. Modelo de coste

El modelo de coste viene definido por el concepto de *gas*, que mide la proporción de recursos necesarios para ejecutar instrucciones en la *EVM*. Así cualquier fragmento de código tiene un coste asociado en términos de *gas*.

Las transacciones tiene un valor de ***gasLimit***, que los emisores pagan en Ether, ajustado por el precio del *gas*, variando según la congestión de la red y las prioridades de los usuarios. El sistema también incluye tarifas básicas y propinas para incentivar a los validadores a incluir transacciones. El objetivo del modelo es equilibrar la demanda de recursos y fomentar el uso eficiente del almacenamiento.

La cantidad de *gas* por instrucción es constante para la mayoría de instrucciones y está estipulada en la documentación de *Ethereum*, donde se enumeran los costes para cada tipo de operación.

| Opcode | Nombre | Descripción | Gas |
|---|---|---|---|
| 0x01 | ADD | Suma | 3 |
| 0x02 | MUL | Multiplicación | 3 |
| 0x57 | JUMPI | Salto condicional | 10 |

Tabla 1.2: Subcojunto opcodes *EVM* [9]

En *Ethereum* existen dos tipos de almacenamiento, el volátil (*memory*) que son las variables locales de una función y el *storage*, también conocido como variables de estado, que son las globales y persisten tras la ejecución de una llamada. Es de suma importancia distinguirlas pues el coste computacional es muy elevado en el caso del *storage*.

Las operaciones de memoria, *MLOAD* y *MSTORE* tienen un coste de 3 unidades de *gas* y adicionalmente una cantidad que depende de la expansión de memoria en bytes respecto al estado anterior. Otras operaciones como *RETURN*, *CALLDATACOPY*, etc. también agregan a su coste estático el asociado a la expansión de memoria. El coste de expansión de memoria está determinado por las siguientes fórmulas [4]:

$$\text{mem\_size\_words} = \left\lceil \frac{\text{mem\_byte\_size} + 32}{32} \right\rceil$$

$$\text{mem\_cost} = \left\lfloor \frac{\text{mem\_size\_words}^2}{512} \right\rfloor + (3 * \text{mem\_size\_words})$$

$$\text{mem\_expansion\_cost} = \text{new\_mem\_cost} - \text{last\_mem\_cost}$$

Los costes de las operaciones de memoria representan un coste insignificante en comparación con las operaciones que interactúan con el *storage*. Esto se debe a que el almacenamiento del *storage* es persistente y se establece en la *blockchain*.

La cantidad de *gas* en operaciones de escritura en *storage* es compleja dependiendo de varios factores, como si se trata de la primera vez que se almacena un valor en una dirección, si se está actualizando un valor existente, si se escribe un cero o sobre un cero. Adicionalmente, hay un *refund* asociado con el borrado de valores para mantener la *blockchain* limpia y eficiente. El coste de la operación *SSTORE* viene definido por el siguiente pseudocódigo:

```
/* orig_val: value of the storage slot before the current transaction
current_val: current value of the storage slot new_val: value to be set in
    the
storage slot cold access: first access to a storage slot via SLOAD or SSTORE
     in
the current transaction */

gas_cost = 0
gas_refound = 0

if (context_addr, target_storage_key) not in touched_storage_slots
    // cold access
    gas_cost += 2100
if new_val == current_val
    // no op
    gas_cost += 100
else
    if current_val == orig_val
        // clean slot, not yet updated in current execution context
        if orig_val == 0
            // slot started zero, being changed to nonzero
            gas_cost += 20000
        else
            // slot started nonzero, being changed
            gas_cost += 2900
            if new_val == 0
                gas_refound += 4800
    else
        // dirty slot, already updated in current execution context
        gas_cost += 100
        if orig_val != 0
            // execution context started with a nonzero value in slot
            if current_val == 0
                // slot started zero, being changed to nonzero
                gas_refound -= 4800
            else if new_val == 0
                // slot started nonzero, being changed to zero
                gas_refound += 4800
        if new_v44al == orig_val
            // slot is reset to the value it started with
            if orig_val == 0
                // slot started zero, being reset to zero
                gas_refound += 1990
            else
                //slot started nonzero, now reset to orig
                gas_refound += 2800
```

Código 1.1: Pseudocódigo para calcular el coste del opcode *SSTORE* [10]

Por otro lado, para la operación de lectura *SLOAD* la lógica es menos compleja, distinguiendo dos casos: el primero cuando se realiza un *cold access* siendo 2100 unidades de *gas*, o en caso contrario 100 unidades[10].

Aparte de los costes de modificación y lectura del *storage*, y de las instrucciones, existen otros suplementos de *gas* que se añaden a la transacción y que son dignos de mencionar, como los consumos de llamar a una transacción de 21000 unidades de *gas* y otros que dependen del *input*, donde cada byte a 0 tiene un coste de 4 unidades frente a las 16 unidades que consume cuando el byte es distinto de 0 [4].

## 1.3. Objetivo

En 2023 más de diez millones de transacciones resultaron fallidas, representando el 5 % la falta de cantidad de *gas* establecida.[1] Esto da lugar a cuestionarse el origen y el motivo de esta causa.

Las ejecuciones en Ethereum requieren establecer una cantidad de gas, que deberá de ser mayor o igual al consumo de dicha transacción para sus correcto desarrollo. La cota superior obtenida por el compilador de Solidity no permite a los usuarios prever con exactitud la cantidad de gas necesaria, llegando a devolver infinito en determinados casos. Dado que el consumo de gas por operación esta definido por el modelo de coste –expuesto en la Sección 1.2.4–, resulta posible predecir el consumo de una transacción.

Tal y como se ha referido en el contexto previamente expuesto, el objetivo de estudio es predecir a partir del *input* de la transacción la cantidad *gas* necesaria para ejecutar una transacción en *Ethereum*, basándose en técnicas de *machine learning*. Para ello, se utilizará la información pública disponible en la *blockchain* sobre las transacciones y *gas* consumido para construir modelos de *machine learning* que permitan predecir el *gas* necesario para ejecutar nuevas transacciones.

En la actualidad se cuenta con herramientas desarrolladas usando métodos formales, como *GASTAP*[11], que son capaces de calcular una cota superior en el consumo de *gas*, siendo muy útiles pues en el caso de que una transacción se quede sin *gas* no será ejecutada y ese *fee* será perdido. Estas, sin embargo, difieren del objetivo expuesto en el presente proyecto al devolver siempre el *gas* máximo, siendo el objetivo de este trabajo el tratar de predecir la cantidad de *gas* más ajustada. Ambas aplicaciones deberían complementarse pues resuelven distintos problemas.

Cabe mencionar que durante el transcurso del estudio se encontraron algunos estudios de aplicación de técnicas de aprendizaje automático en *blockchain*, pero la mayoría de ellos se centraban en predecir el precio, tanto de criptomonedas como del *gas*, pero no se encontró ninguno que se centre en la predicción de la cantidad de *gas* por instrucción.

[1] Dato obtenido de BigQuery el 2 de mayo de 2024.

# Capítulo 2

# Obtención y preparación de los datos

## 2.1. Búsqueda de fuentes

A fin de construir una base sólida se debe cumplir con la premisa de obtener un conjunto de datos veraz y con las columnas necesarias para poder realizar un modelo efectivo. En añadido a la obtención del dataset, se ha optado por apoyarse en herramientas para una verificación de datos rápida y con la que acceder a una interfaz visual. Este entorno presenta un ecosistema idóneo, al disponer de los datos de forma pública y veraz por definición, evitando la manipulación de datos erróneos o incompletos.

### 2.1.1. Google Cloud BigQuery

Tras valorar distintas plataformas con distintos datasets se ha optado por *Google Cloud BigQuery*[12], una plataforma de almacenamiento y análisis de datos administrada. BigQuery cuenta con un conjunto de datos, crypto_ethereum[13], el cual se mantiene constantemente actualizado, y que cuenta con distintas tablas, que contienen datos suficientes para la creación del modelo de *machine learning*.

A partir del conjunto de datos crypto_ethereum, se ha utilizado la tabla de contratos para obtener los más utilizados en *Ethereum* y, la tabla de transacciones para obtener todos los datos de las transacciones asociadas a estos contratos más utilizados.

Este conjunto dispone de las siguientes tablas más destacables:

| Tabla | Descripción |
|---|---|
| balances | Ether balances of all addresses, updated daily. |
| blocks | A set of all blocks in the blockchain and their attributes. |
| contracts | A subset of Ethereum addresses that contain contract byte-code, as well as some basic analysis of that byte-code. |
| logs | Generally useful for reporting on any logged event type on the Ethereum blockchain. |
| token_transfers | The subset of those transactions and has further processed and denormalized the data to make it easier to consume for analysis of token transfer events. |
| transactions | A set of all transactions from all blocks, and contains a block identifier to get associated block-specific information associated with each transaction. |

Tabla 2.1: Tablas del conjunto de datos crypto_ethereum de BigQuery [13]

### 2.1.2. Etherscan

Con el objetivo de extender la información sobre los contratos, se ha encontrado en *Etherscan*[14] la opción para trabajar con su API y poder obtener así el *ABI* de cada contrato para posteriormente poder decodificar el input de cada transacción.

A su vez, se ha utilizado *Etherscan* como plataforma de consulta rápida, debido a la facilidad para acceder y visualizar transacciones y su consumo de gas e *input* asociado, así como detalles como el código y *ABI* del contrato.

## 2.2. Extracción

Para el estudio de los experimentos se ha contado con una máquina local proporcionada por el grupo de investigación COSTA[15] de la Facultad de Informática de la UCM con la capacidad suficiente para la realización de todos los experimentos, y, a su vez permitiendo evitar los altos costes de realizar un estudio en plataformas como *Google Cloud*, preparadas para el manejo de grandes volúmenes de datos.

Para la extracción de los datos se ha creado una cuenta tanto en BigQuery como en *Etherscan*, se han obtenido dos tokens utilizados como credenciales a la hora de hacer las extracciones vía código para extraerlos directamente a la máquina local mencionada anteriormente y procesar los datos desde allí.

Por último, se ha optado por utilizar lenguajes de programación como Python para realizar los fragmentos de código destinados a la extracción y SQL para realizar las consultas a las tablas en BigQuery.

### 2.2.1. Contratos

En el contexto del estudio se ha decidido trabajar con los contratos más utilizados –aquellos con el mayor número de transacciones– con la finalidad de contar con la mayor cantidad de datos posibles y que exista una mayor información sobre los mismos.

Han sido escogidos mediante la selección de los 100 contratos que acumulaban mayor número de transacciones desde el día 01-01-2023 hasta el día 01-03-2024, y que a su vez tengan su *ABI* verificado en *Etherscan* ya que es condición necesaria para el posterior procesamiento del *input*.

Esta extracción ha sido realizada en BigQuery a partir de su tabla de contratos:

| Columna | Descripción |
|---|---|
| address | Address of the contract. |
| bytecode | Bytecode of the contract. |
| function_sighashes | 4-byte function signature hashes. |
| is_erc20 | Whether this contract is an ERC20 contract. |
| is_erc721 | Whether this contract is an ERC721 contract. |
| block_timestamp | Timestamp of the block where this contract was created. |
| block_number | Block number where this contract was created. |
| block_hash | Hash of the block where this contract was created. |

Tabla 2.2: Tabla de contratos BigQuery [16]

Posteriormente se ha procedido con la extracción del *ABI* de cada contrato, esta tarea se ha realizado a partir de la API de *Etherscan*.

A la hora de realizar el decode del *input* de las transacciones surgió un problema donde la función ejecutada no estaba en el *ABI* del contrato. Este fue resuelto al descubrir de que el contrato hacía uso del patrón *proxy*. Este problema se repitió en un gran número de ocasiones. Finalmente, se optó por realizar una extracción manual del *ABI* de los contratos de implementación para aquellos que tuvieran uno asociado y por ende solucionar el problema.

La información de los contratos se ha almacenado en un fichero csv resultando con su identificador, *ABI*, y *ABI* del contrato de implementación asociado si así lo tuviera.

### 2.2.2. Transacciones

Tras obtener el conjunto de contratos válidos, se ha procedido a extraer las transacciones válidas, que han sido ejecutadas, asociadas a sus respectivos contratos.

Debido a la gran cantidad de transacciones que existen para cada contrato en el transcurso de un año, se ha optado por esparcir en el tiempo la extracción de las muestras, realizándolas los días catorce y veintiocho de cada mes desde el 14-01-2023 hasta el 28-02-2024.

Esta extracción ha sido realizada en BigQuery a partir de su tabla de transacciones, previamente se ha realizado un filtro de algunas columnas de la tabla origen debido a la nula correlación con el contexto del trabajo. A continuación se muestran subrayadas las variables seleccionadas.

| Columna | Descripción |
|---|---|
| **hash** | Hash of the transaction. |
| nonce | The number of transactions made by the sender prior to this one. |
| transaction_index | Integer of the transactions index position in the block. |
| from_address | Address of the sender. |
| **to_address** | Address of the receiver. null when it's a contract creation transaction. |
| value | Value transferred in Wei. |
| gas | Gas provided by the sender. |
| gas_price | Gas price provided by the sender in Wei. |
| **input** | The data sent along with the transaction. |
| receipt_cumulative_gas_used | The total amount of gas used when this transaction was executed in the block. |
| **receipt_gas_used** | The amount of gas used by this specific transaction alone. |
| receipt_contract_address | The contract address created, if the transaction was a contract creation, otherwise null. |
| receipt_root | 32 bytes of post-transaction stateroot (pre Byzantium). |
| receipt_status | Either 1 (success) or 0 (failure) (post Byzantium). |
| **block_timestamp** | Timestamp of the block where this transaction was in. |
| block_number | Block number where this transaction was in. |
| block_hash | Hash of the block where this transaction was in. |
| max_fee_per_gas | Total fee that covers both base and priority fees. |
| max_priority_fee_per_gas | Fee given to miners to incentivize them to include the transaction. |
| transaction_type | Transaction type. |
| receipt_effective_gas_price | The actual value per gas deducted from the sender's account. |

Tabla 2.3: Tabla de transacciones BigQuery [17]

Se ha extraído un total de 10.764.324 de transacciones almacenadas en veintiocho ficheros csv procedentes de la extracción de cada día resultando con su *hash*, contrato ejecutado, entrada de la función ejecutada, fecha de ejecución de su bloque y cantidad de gas consumido.

## 2.3. Procesamiento y modelado

La gran cantidad de datos que se espera extraer y procesar es bastante alto, lo que llevaría a tiempos de ejecución muy elevados tanto para el tratamiento de los datos como el entrenamiento y evaluación de los modelos de *machine learning*, por lo que la programación monoproceso convencional no es acertada. Optando por emplear programación multiproceso, paralelizando los distintos trabajos en diferentes hilos para repartir equitativamente la carga y poder reducir al máximo el tiempo de ejecución de los programas.

Esta paralelización se ha realizado a nivel de un hilo para cada signatura de cada contrato, es importante entender este concepto ya que muchos contratos comparten signaturas debido a los mismos nombres de funciones, pero se separan debido a que ejecutan distinto código Solidity[18].

Una vez agrupadas las transacciones para cada signatura de cada contrato se ha procedido con el decode del *input*. Como se ha mencionado anteriormente el *input* es una cadena en hexadecimal donde los ocho primeros caracteres pertenecen a la signatura, parte por la que se ha agrupado para cada contrato, y los siguientes dígitos son los parámetros de entrada con los que se ha ejecutado la función en dicha transacción. Resultando un dataframe individual para cada agrupación.

En primera instancia se ha optado por tratar el *input* devolviendo la longitud del mismo, con la hipótesis de que debiera existir una relación lineal entre la longitud de éste y el número de instrucciones ejecutadas.

En una segunda instancia se ha optado por incorporar a los datos, el *input* procesado y separado para cada variable. Esto se puede realizar una vez obtenida la signatura, donde gracias al $ABI$ se conocen cuáles son los parámetros de entrada de la función, y por ende hacer un decode de la entrada en hexadecimal y separarla en los distintos parámetros de la función.

Este proceso se ha automatizado con web3_input_decoder[19], una librería en Python que contiene una función la cual recibe como parámetros el $ABI$ del contrato y el input de la transacción a la que realizar el decode, devolviendo una lista de tuplas con su tipo, nombre de parámetro y valor:

$$decode(ABI, input) \Rightarrow [(type, name, value)]$$

A la hora de escoger el *ABI* a tratar se ha priorizado el propio del contrato y en caso de fallo se procede a utilizar el del contrato de implementación si existe. En algunos casos aun utilizando ambos *ABIs* se ha encontrado el problema de que la signatura no existe, en dichos casos se ha procedido a tan solo contar con la longitud del *input*.

Otro problema afrontado ha sido debido a que no es capaz de adaptar el *input* a los parámetros proporcionados por el *ABI*, lo que hace suponer que se utiliza la región de memoria *CALLDATALOAD* para el paso de estos parámetros. Al igual que en el anterior caso, se ha optado por entrenar el modelo tan solo con la longitud del *input*. Dejándose ambos casos abiertos a posibles mejoras de este trabajo, tratando de abordarlos ampliando el espectro de datos y poder realizar modelos más completos.

Una vez se tiene el *input* decodeado para cada variable con su valor se ha procedido en función del tipo de variable de la siguiente forma:

- **address**: no se procesa ya que para este tipo se entiende que no guarda una relación con la cantidad de gas a consumir debido a que este campo hace referencia a un identificador único a una cuenta o contrato dentro de la red de *Ethereum*, no a un posible valor sobre el que realizar alguna acción en concreto.
- **bool**: se devuelve el valor del booleano como 0 (false) y 1 (true).
- **list**: se devuelve el número de elementos de la lista, además se actúa recursivamente de forma que si se tratase de una matriz, devolvería el número total de elementos y no solo el número de filas.
- **int** / **uint**: en ambos casos se realizan dos tratamientos distintos debido a un limitación de la librería pandas ya que solo tiene enteros de hasta 64 bits y en Solidity se tratan enteros de hasta 256 bits. Para los casos que sean de hasta 64 bits se devuelve el valor del entero y para los mayores de 64 bits se devuelve el orden de magnitud del valor. Al aplicar la regla sobre el tipado de las variables, todas ellas trabajarán bajo las mismas unidades.

Para el resto de elementos no reconocidos se devuelve la longitud de la cadena hexadecimal quitando los ceros de la izquierda.

En una tercera y última aproximación se decidió añadir un campo booleano adicional relacionado con los enteros sobre si su valor es 0, ya que solo en los casos donde el entero fuese de hasta 64 bits se podía conocer dicho valor, y los demás enteros más grandes tomaba el valor 1, ya que ese es su orden de magnitud. Esta aproximación surge con la finalidad de interpretar los resultados obtenidos más adelante, al observar saltos condicionales que verificaban si el valor era 0.

En todas las aproximaciones, los valores obtenidos se han añadido al conjunto de datos individual de cada signatura como una columna nueva cuyo nombre es de la forma: *input_len* para la longitud del *input*, *type_name* para cada variable y *name_is_zero* para comprobar si son 0.

# Capítulo 3

# Modelos predictivos

## 3.1. Aproximación

Tal y como se ha expuesto previamente, el objetivo a tratar en este TFG es la predicción de la cantidad de *gas* que consumirá una transacción a partir del *input* de la misma, tratándose así de un problema de regresión.

A partir de esta información, la primera aproximación que se ha realizado para este problema es la de aplicar dos modelos distintos. Un modelo de regresión lineal, sencillo, fácil de implementar y cuya interpretabilidad es muy alta. Y otro que emplea la técnica *Gradient Boost*, añadiendo más complejidad lo que puede permitir obtener resultados más precisos pero en cambio disminuye su interpretabilidad.

Los modelos utilizados pertenecientes a la librería de Python Scikit-learn han sido los siguientes:

| Modelo | Descripción |
|---|---|
| Linear Regression | Fits a linear model with coefficients w = (w1, . . . , wp) to minimize the residual sum of squares between the observed targets in the dataset, and the targets predicted by the linear approximation. |
| Gradient Boosting Regressor | This estimator builds an additive model in a forward stage-wise fashion; it allows for the optimization of arbitrary differentiable loss functions. In each stage a regression tree is fit on the negative gradient of the given loss function. |

Tabla 3.1: Modelos de Scikit-learn [20][21]

El modelo de Linear Regression devuelve una función de coste que permite reproducir el comportamiento del modelo. La función de coste obtenida es:

$$gas(X) = \omega_0 + \omega_1 X_1 + \cdots + \omega_m X_m$$

Tras un análisis exhaustivo de los resultados obtenidos –tal y como se explicará en detalle en el Capítulo 4–, se ha decidido optar por no añadir más modelos de aprendizaje automático, considerando que no va a aportar más información relevante a las conclusiones de este estudio.

## 3.2. Entrenamiento

Al trabajar con magnitudes de datos muy elevadas, el tratamiento, y en particular el entrenamiento de los modelos resulta muy costoso, por lo que al igual que en la extracción, se ha paralelizado el entrenamiento de los distintos modelos a partir de las agrupaciones resultantes de la extracción de los datos, separando por signatura y contrato.

En la aproximación realizada se ha optado por dividir el conjunto de datos para cada agrupación en dos subconjuntos, uno con el 80 % de los datos destinado al entrenamiento de los modelos, y otro con el 20 % restante para poder evaluar los modelos resultantes. Se ha optado por no añadir otro subconjunto de validación, debido a que en una primera instancia no existe la intención de optimizar los hiperparámetros de los modelos.

Esta técnica de división en entrenamiento y prueba *(Train Test Split)* es fundamental aplicarla en la construcción de los distintos modelos de machine learning, entre varias razones, para evitar el sobreajuste de los modelos y por tanto evitar resultados muy buenos para ese conjunto, pero que no se ajustarían a otros conjuntos de datos.

Una vez finalizados los entrenamientos y pruebas de los modelos han sido guardados en un diccionario de la forma *dict[contrato][signatura]* donde se almacenan los dos modelos resultantes, el tamaño del conjunto de datos y los parámetros de evaluación. Esta estructura concede un rápido acceso a los distintos modelos y demás parámetros, permitiendo así poder evaluar y probar todos los modelos con datos nuevos de forma sencilla.

Para el posterior análisis de los modelos, en el mismo diccionario se ha guardado un subconjunto de transacciones del conjunto de prueba para poder realizar análisis más cómodos pues ya se cuenta con los datos procesados para cada uno de ellos. Este subconjunto cuenta con un límite de 500 registros.

## 3.3. Evaluación

La evaluación de los modelos supone un reto, pues se enfrenta a diferentes conjuntos de datos con distintas magnitudes, es decir, las funciones tienen consumos de *gas* de diferentes órdenes de magnitud. Con la finalidad de poder evaluar y comparar sin importar la cantidad de *gas*, se descartaron las métricas más comunes como MAE y RMSE, pues resultaban más grandes simplemente porque los datos en el conjunto están en una escala mayor. Al principio se consideró que el error porcentual absoluto medio (MAPE) podría ser una buena opción, pero finalmente se acordó que dada la naturaleza del problema interesaba más conocer el error sin importar la relación con la cantidad de *gas* que consuma la transacción. Finalmente se optó por dos métricas principales: el coeficiente de determinación ($R^2$) y el error absoluto medio (MAE).

El coeficiente de determinación puede interpretarse como la proporción de la varianza de la variable dependiente que es predecible a partir de las variables independientes[22].

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

Donde $y_i$ son los valores reales, $\hat{y}_i$ los valores predichos por el modelo y $\overline{y}$ el promedio de los valores reales. Un ajuste perfecto daría lugar a $R^2 = 1$ y por contrario un modelo constante que devuelva la media devolvería $R^2 = 0$, pudiendo tomar valores negativos.

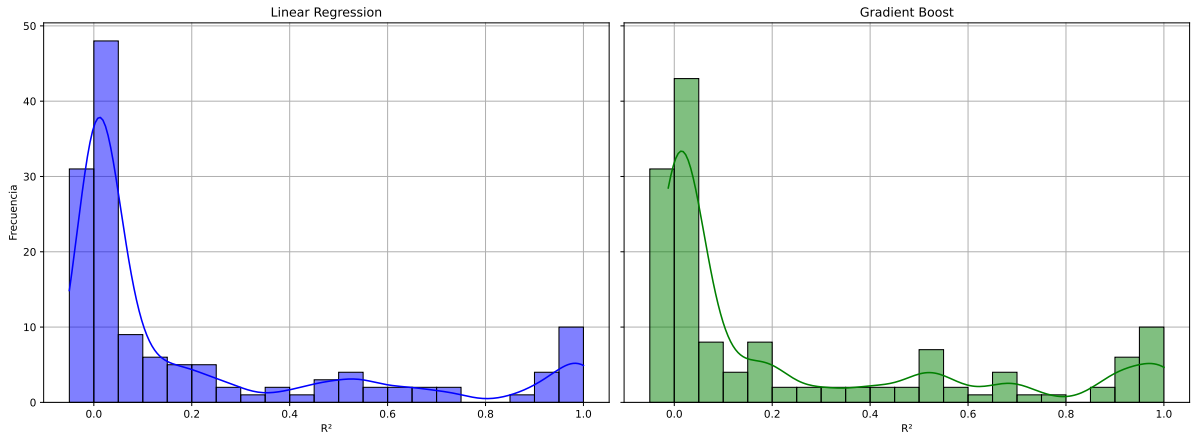



Figura 3.1: Resultados $R^2$

El grupo mayoritario se sitúa en métricas cercanas a 0, lo que resulta un ajuste muy negativo. Contrariamente, un subconjunto de 15 funciones tienen una métrica considerablemente acertada.

La métrica MAE representa la diferencia absoluta promedio de las predicciones frente a los valores reales.

$$MAE = \frac{\sum_{i=1}^{n} |y_i - x_i|}{n}$$

Donde $y_i$ es la predicción y $x_i$ es el valor verdadero. Por tanto, valores menores reflejan un mejor entrenamiento del modelo.

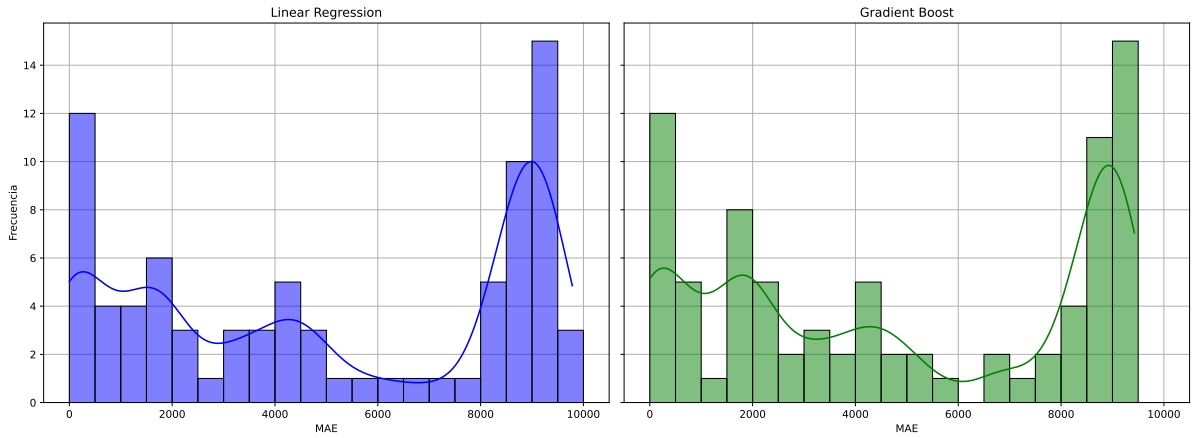



Figura 3.2: Resultados MAE $\leq$ 10.000

Para el análisis se ha representado el histograma de la métrica MAE cuyo valor es menor o igual a 10.000 pues los errores se distribuyen en diferentes órdenes de magnitud y se ha optado por interpretar los resultados en dos secciones, para representar los resultados mayores que 10.000, se ha empleado una distribución logarítmica en base 10 que permita una visión detallada, –véase Figura A.1–.

Como se puede apreciar en ambas métricas se cuenta con un grupo de funciones que el modelo consigue predecir la cantidad de *gas* de manera satisfactoria, mientras que para el grupo mayoritario no logra predecirlo. A continuación, se desglosan estos resultados realizando diversos experimentos para poder analizar y evaluar los modelos.

# Capítulo 4

# Experimentos y resultados

Ante la obtención de resultados muy desfavorables en las predicciones, se ha priorizado realizar experimentos que buscan la explicabilidad, con la finalidad de poder realizar un entendimiento de los resultados y llegar a unas conclusiones veraces e interpretables.

Como se mencionó anteriormente en el modelo de coste, modificar el *storage* –operaciones *SSTORE*– supone unos costes muy dispares, que dependen de condiciones imposibles de conocer "a priori", lo que puede hacer que los modelos se confundan y por ende este es un buen camino por el que comenzar los experimentos.

## 4.1. Análisis de la utilización de opcodes SSTORE

Como primera aproximación se ha optado por un análisis más sencillo observando el número de apariciones de la operación *SSTORE* en el bytecode de los contratos. Este análisis realiza una primera aproximación, pues no se distingue por función, pero permite tener una visión general sobre si los contratos, en algún punto, modifican o no el *storage*.

La extracción del bytecode se realizó a traves de la plataforma *Etherscan*, y para la posterior transformación se empleo el compilador oficial de *Solidity*.

Como resultado de este experimento se obtuvo que todos los contratos hacían uso de la operación *SSTORE* en múltiples ocasiones, dando a entender que en distintos puntos de todos los contratos se modifica el *storage*, pudiendo ser esta la causa de los malos resultados.

## 4.2. Estudio de signaturas que pertenecen a IERC20

Partiendo del resultado anterior y para entender los resultados se ha querido realizar un análisis del código de las funciones de cada signatura. Ante la gran cantidad de estas, se ha tratado de reducir el espectro encontrando patrones similares, resultando en la interfaz *ERC20*, una interfaz que define el estándar para la implementación de *tokens* en *smart contracts*, y que gran parte de los contratos que están en la *blockchain* implementan[23].

Esta interfaz cuenta con pocas funciones fáciles de analizar y sobre las que se conoce que realizan accesos y modificaciones del *storage*, por lo que la hipótesis es que los resultados de los modelos para estas signaturas sean desfavorables.

Para la búsqueda de las signaturas se han utilizado las siguientes bien conocidas:

```
SIGNATURES_ERC20 = {
    '0x095ea7b3': 'approve',
    '0xa9059cbb': 'transfer',
    '0x23b872dd': 'transferFrom',
    '0xd505accf': 'permit',
    '0x42966c68': 'burn',
    '0x79cc6790': 'burnFrom',
    '0x40c10f19': 'mint',
    '0xa457c2d7': 'decreaseAllowance',
    '0x39509351': 'increaseAllowance',
    '0x2f4f21e2': 'depositFor',
    '0x205c2878': 'withdrawTo'
}
```

Código 4.1: Signaturas IERC20

Cabe recalcar que cada signatura tiene su propia implementación pese a que extienda dicha interfaz, por lo que las instrucciones de mismas signaturas pueden no ser idénticas.

Una vez realizada la búsqueda se han encontrado 41 signaturas que pertenecen a la interfaz *ERC20*. Explorando los resultados obtenidos se observa que efectivamente la hipótesis era correcta, con la excepción de un único caso que se predice de manera satisfactoria, el cual se procede a analizar.

### 4.2.1. Análisis y comparación de caso favorable

Para poder interpretar el resultado y con el objetivo de comprender el motivo de su predicción, se optó en primera instancia por representar un gráfico de dispersión de los valores reales frente a los valores predichos.

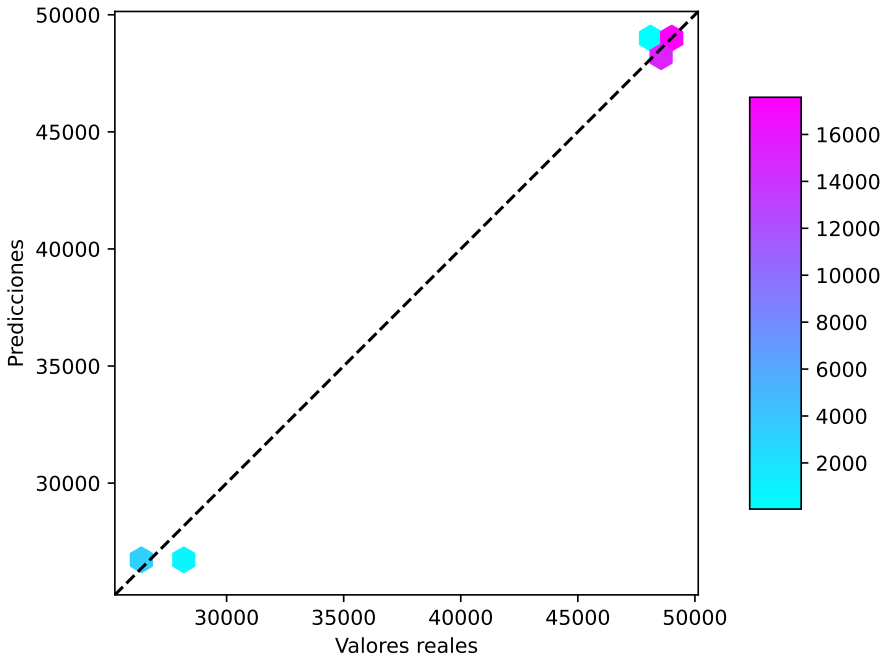



Figura 4.1: Gráfica dispersión caso favorable

En esta gráfica se observa cómo el modelo devuelve un valor menor para ciertos casos donde los valores reales toman dos posibles valores entre 25.000 y 30.000, que el modelo no es capaz de discernir. Por otro lado los casos de más de 45.000 lo predice de forma satisfactoria con un cierto error. Para poder comprender la predicción se procedió a examinar el código de *Solidity* de la signatura.

```
1 // etherscan.io/address/0xdac17f958d2ee523a2206206994597c13d831ec7#code
2 function approve(address _spender, uint _value)
3         public onlyPayloadSize(2 * 32) {
4
5     require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));
6
7     allowed[msg.sender][_spender] = _value; Approval(msg.sender,
8     _spender,_value);
9 }
```

Código 4.2: Caso favorable

En primer lugar valida dos condiciones, una dependiente de la entrada y otra del *storage*, y en caso de no cumplirse no realiza la escritura en el *storage*. Por ello se consideró que en función del parámetro de entrada *_value*, si era 0 o no, el modelo estaba prediciendo en el grupo correspondiente. Esta hipótesis fue respaldada al comprobar los pesos asignados por los modelos a la condición *is_zero* de la variable *_value* en esta signatura.

### 4.2.2. Análisis y comparación de caso desfavorable

Para el análisis de los casos desfavorables se ha optado por examinar detalladamente otro signatura del método *approve*, pero en este caso en particular uno que el modelo no sea capaz de predecir correctamente.

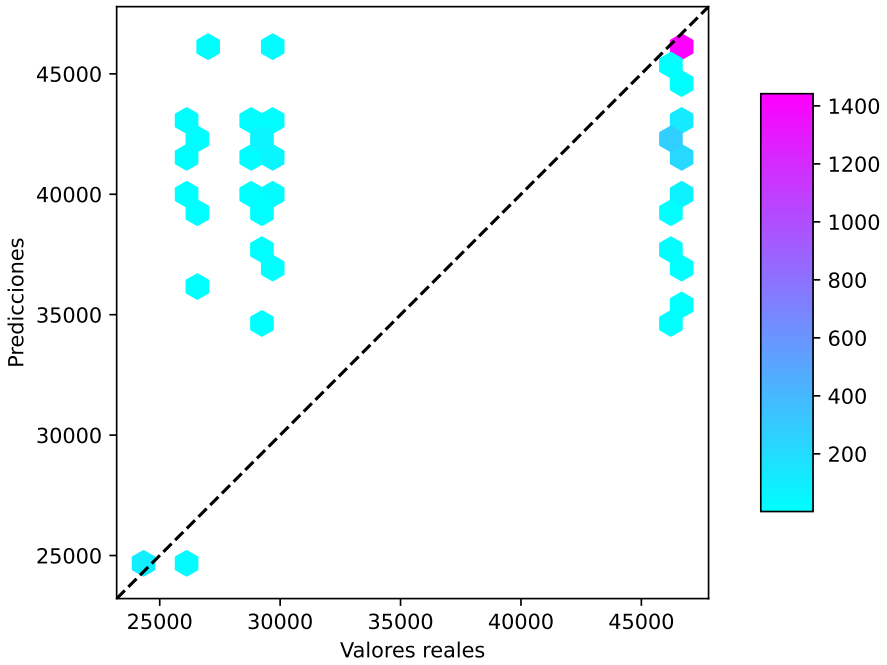



Figura 4.2: Gráfica dispersión caso desfavorable

En la gráfica de dispersión se aprecian cinco diferentes valores y cómo el modelo no es capaz de ajustar los resultados en función del parámetro *amount*. Para la comprensión del resultado se procede a analizar el código correspondiente a esta función.

```
// etherscan.io/address/0x6982508145454ce325ddbe47a25d4ec3d2311933#code
function approve(address spender, uint256 amount)
        public virtual override returns (bool) {
    _approve(_msgSender(), spender, amount);
    return true;
}

function _approve(address owner, address spender, uint256 amount) internal
    virtual {
    require(owner != address(0), "ERC20: approve from the zero address");
    require(spender != address(0), "ERC20: approve to the zeroaddress");

    _allowances[owner][spender] = amount; emit Approval(owner, spender,
    amount);
}
```

Código 4.3: Caso desfavorable

A diferencia del caso anterior, no realiza la comprobación de si el valor a escribir es 0, tan solo valida los direcciones recibidas por los parámetros de entrada. Realizando la escritura para todos los valores. Esto se traduce en todas las posibles casuísticas definidas con anterioridad en el modelo de coste, que dan como resultado esos grupos en la gráfica de dispersión y la falta de capacidad para predecirlos de manera adecuada, por la ausencia de conocimiento del estado del *storage*.

## 4.3. Análisis y evaluación de modelos favorables

Así, mientras la evaluación de los modelos reportaba una serie de funciones que se estaban prediciendo correctamente, en esta sección se procederá al estudio y comprensión de la predicción, investigando sobre la relación lineal con el tamaño de la entrada, el peso asignado a los parámetros de entrada y si fuese necesario el código fuente de la función.

Tras el análisis se han observado distintos patrones, lo que ha permitido la siguiente clasificación de los modelos:

### 4.3.1. Modelos constantes

En el primer análisis exploratorio se encontraron dos modelos cuyas métricas resultaban $R^2 = 1$ y $MAE = 0$, dieron lugar a modelos donde evaluaban signaturas con consumo de *gas* constante y resultando en un modelo trivial.

En la siguiente figura se muestra el consumo de *gas* por ejecución para un caso concreto y donde se puede ver un consumo constante a lo largo del tiempo. El otro modelo constante observado sigue la misma distribución.

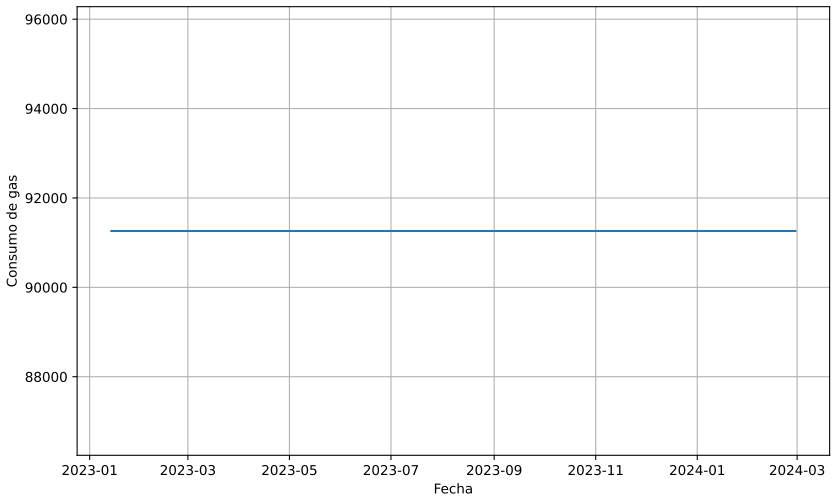



Figura 4.3: Gráfica temporal caso constante

Este consumo constante proviene de una serie de instrucciones, cuyo coste asignado a su *opcode* es constante, y siempre se ejecutan el mismo número de ellas. Por ende se puede descartar el empleo de instrucciones con consumo de *gas* dinámico, como pueden ser de acceso o modificación del *storage*.

### 4.3.2. Modelos lineales con el tamaño de entrada

Tras el análisis de los modelos constantes, se ha procedido a un análisis del peso de las variables donde se puede ver el protagonismo de la variable *input_len* tomando casi el 100 % del peso en los distintos modelos. Para otros, la longitud de la entrada no toma todo el peso apoyándose en otras variables para realizar la predicción.

A continuación se muestra a la izquierda la gráfica de dispersión de los valores reales frente a las predicciones, siendo muy fiel a la gráfica de identidad, que resultaría un ajuste perfecto. A la derecha se muestra la relación lineal entre el valor real y la longitud de la entrada, donde se aprecia un correlación significativamente alta.

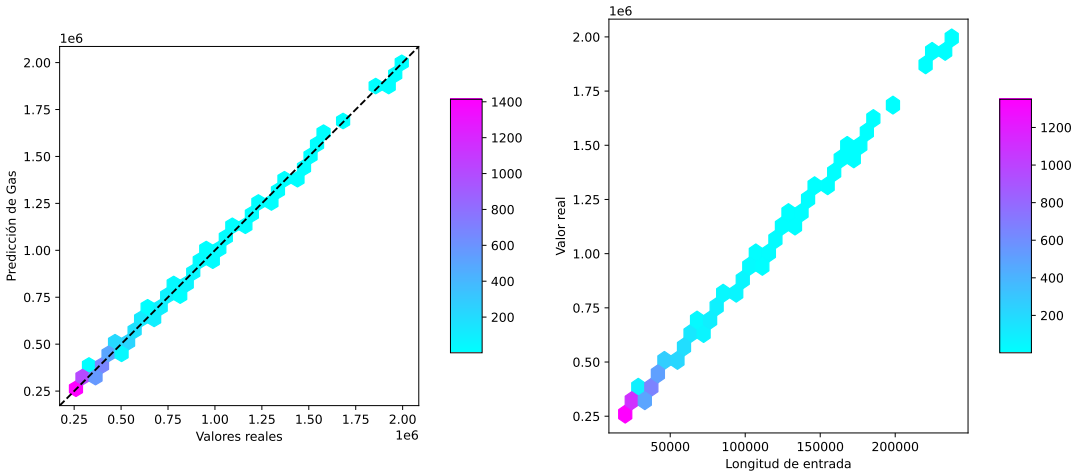



Figura 4.4: Grafícas de dispersión y relación *input*

En este caso en particular llama la atención que pese a trabajar con magnitudes de *gas* muy dispares, la predicción es muy certera obteniendo un $MAE = 1794$ lo que supone a nivel porcentual un error del 0,4 %.

Por lo expuesto se ha realizado un análisis del código de la función. En este caso no hace uso de parámetros de entrada, si no que utiliza la zona de memoria *CALLDATALOAD*, de la cual extrae los datos del *input* a partir de un offset. Se puede observar cómo realiza un bucle en función del número de elementos y posteriormente una modificación de una variable almacenada en el *storage*. Se han omitido instrucciones que no se consideran relevantes.

```
// etherscan.io/address/0x5e4e65926ba27467555eb562121fac00d24e9dd2#code
function appendSequencerBatch() external {
    uint40 houldStartAtElement;
    uint24 totalElementsToAppend;
    uint24 numContexts;
    assembly {
        shouldStartAtElement := shr(216, calldataload(4))
        totalElementsToAppend := shr(232, calldataload(9))
        numContexts := shr(232, calldataload(12))
    }

    ...

    BatchContext memory curContext;
    for (uint32 i = 0; i < numContexts; i++) {
        BatchContext memory nextContext = _getBatchContext(i);
        curContext = nextContext;
        numSequencerTransactions += uint32(curContext.
    numSequencedTransactions);
        nextQueueIndex += uint40(curContext.numSubsequentQueueTransactions);
    }

    ...

    // Update the _nextQueueIndex storage variable.
    _nextQueueIndex = nextQueueIndex;
}
```

Código 4.4: Caso dependiente del tamaño de la entrada

Los puntos relevantes que propician estos errores tan positivos son en primer lugar que en el código principal de la función –el bucle–, no se realizan modificaciones al *storage*, solo se ejecutan instrucciones cuyo consumo de *gas* es constante. Por otro lado, no se realizan múltiples escrituras en el *storage*, tan solo se realiza una al final de la función.

En diversos casos se ha observado que cuando se realizan operaciones iterativas en función de los parámetros de entrada, la cantidad de *gas* consumido por estas opaca el consumo por modificar el *storage*, obteniendo una aproximación al valor real pero no certera.

## 4.4. Análisis bytecode de un smart contract

Una vez realizados los anteriores experimentos se quiere replicar los comportamientos identificados para cerciorarse de la veracidad de los mismos. Para ello se va a desarrollar un contrato –con las casuísticas relevantes–, y mediante el compilador oficial de Solidity obtener el bytecode asociado al contrato. Por otro lado se va a lanzar el mismo contrato en una máquina virtual de Remix[24], que simula la conducta de la *blockchain* para la versión Shanghai de *Ethereum*[25].

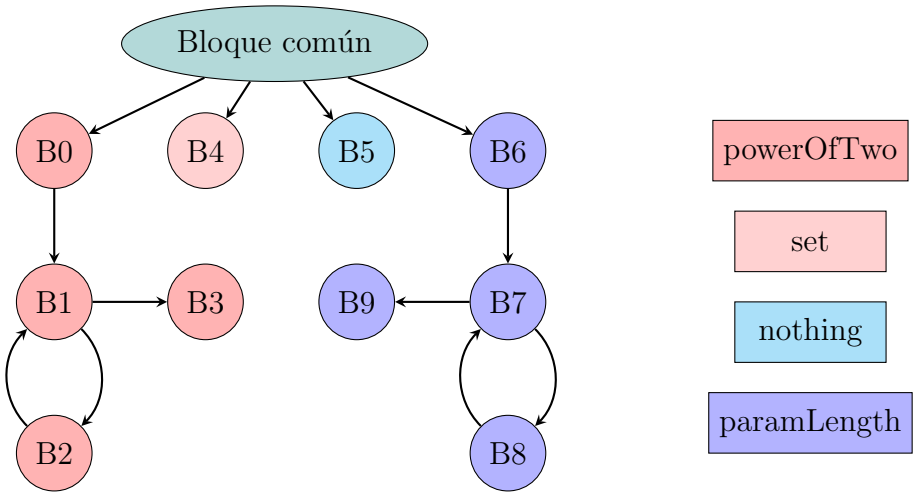
El objetivo de este experimento es analizar y contrastar de manera experimental, mediante la máquina virtual de Remix, las funciones de coste de los métodos públicos definidos en el contrato. Para ello se ha utilizado el *framework* de *EthIR*[26] para obtener el *CFG (Control Flow Graph)* del contrato a partir del bytecode obtenido.

```
1  pragma solidity 0.8.20;
2
3  contract GasUsageTest {
4
5      // Private variables, this are saved in storage
6      uint private storedData;
7
8      // Function that writes in storage
9      function set(uint x) public {
10         storedData = x;
11     }
12
13     // Function that does only changes in the stack
14     function nothing() public pure {
15         uint example = 100;
16         example = 0;
17     }
18
19     // Function that calculates power of 2
20     function powerOfTwo(uint x) public pure returns (uint) {
21         uint sum = 1;
22         for (uint i = 0; i < x; i++) {
23             sum += sum;
24         }
25         return sum;
26     }
27
28     // Function that depends on the length of the params
29     function paramLength(uint[] memory list) public pure returns (uint) {
30         return list.length;
31     }
32 }
```

Código 4.5: GasUsageTest.sol

Desarrollar una función de coste tiene un inconveniente con las operaciones de coste dinámico, no siendo posible determinar el coste exacto de dicha operación sin ser conocedor del estado del *storage* y de la ejecución. Es por ello que para el desarrollo de las funciones de coste se va a utilizar el consumo base de la operación, omitiendo el asociado a la expansión de memoria o acceso a *storage*, para el posterior análisis aislado. A su vez se va a omitir el coste asociado al número de bytes ceros y no ceros de la entrada siendo trivial su cálculo –descrito en la Sección 1.2.4–.

La siguiente figura representa una adapatación esquemática del *control-flow graph* obtenido al analizar el bycode resultante con el *framework* de *EthIR*. Cada nodo representa un bloque de código que se ejecuta secuencialmente, para una mayor interpretabilidad se ha optado por esquematizarlo y reducirlo agrupando bloques. Como se puede observar para las funciones *powerOfTwo* y *paramLength* aparecen ciclos en el grafo que representan las operaciones iterativas, descritas en el Código 4.5. Para el resto de casos, la ejecución es secuencial, no dependiendo el número de instrucciones a ejecutar por el parámetro de entrada. El árbol binario original se puede encontrar en el repositorio, junto al bytecode.



### 4.4.1. Coste constante

El primer caso a analizar es de una transacción con consumo de *gas* constante, representado por la función *nothing*(). A continuación se desglosan los opcodes, número de apariciones, y su coste asociado:

| Opcode | Consumo base | Número de apariciones | Consumo total |
|---|---|---|---|
| JUMPI | 10 | 4 | 40 |
| PUSH2 | 3 | 6 | 18 |
| JUMP | 8 | 2 | 16 |
| PUSH1 | 3 | 5 | 15 |
| DUP1 | 3 | 3 | 9 |
| POP | 2 | 4 | 8 |
| PUSH4 | 3 | 2 | 6 |
| PUSH0 | 2 | 3 | 6 |
| SWAP1 | 3 | 2 | 6 |
| EQ | 3 | 2 | 6 |
| JUMPDEST | 1 | 4 | 4 |
| ISZERO | 3 | 1 | 3 |
| MSTORE | 3 | 1 | 3 |
| LT | 3 | 1 | 3 |
| CALLDATALOAD | 3 | 1 | 3 |
| SHR | 3 | 1 | 3 |
| CALLVALUE | 2 | 1 | 2 |
| CALLDATASIZE | 2 | 1 | 2 |
| STOP | 0 | 1 | 0 |
| Total | | | 153 |

Tabla 4.1: Opcodes *nothing*()

Empíricamente la llamada en Remix resultaba en un consumo de 162, difiriendo en 9 unidades de *gas* de la función de coste obtenida.

El primer bloque de instrucciones del contrato, realiza un *MSTORE* con offset 64, que supone un coste de expansión de memoria de 9 unidades de *gas*. Este consumo de *gas* se repetirá en los demás casos de estudio, pues el primer bloque lo ejecutan todos ellos.

$$mem\_size\_words = \lceil \frac{mem\_byte\_size + 32}{32} \rceil = \lceil \frac{64 + 32}{32} \rceil = 3$$

$$mem\_cost = \lfloor \frac{mem\_size\_words^2}{512} \rfloor + (3 * mem\_size\_words) = \lfloor \frac{3^2}{512} \rfloor + (3 * 3) = 9$$

$$mem\_expansion\_cost = new\_mem\_cost - last\_mem\_cost = 9 - 0 = 9$$

Sumando el consumo dinámico a la función de coste resultante, se obtiene una convergencia entre el experimento empírico y el formal, resultando en:

$$dynamic\_gas = 9$$

$$gas(nothing()) = 153 + dynamic\_gas = 162$$

### 4.4.2. Coste en función del parámetro de entrada

El segundo caso a replicar es el de una transacción que su consumo de *gas* esté relacionado con un número de operaciones realizadas a partir de algún parámetro de entrada. La función $powerOfTwo(x)$ recibe como parámetro de entrada un entero sin signo, que determina el número de iteraciones de un bucle for, y dentro de este se realiza una operación simple.

A continuación se va a diferenciar los opcodes en dos grupos, el primero de ellos hace referencia a la parte no iterativa del código, es decir lo que se ejecuta una única vez, lo que representaría en el esquema anterior el "Bloque común", "B0", "B1" y "B3". El segundo grupo denota la parte iterativa del código, la secuencia "B1" y "B2', que se repetirá x veces. Véase las tablas resultantes en Tabla A.1 y Tabla A.2.

La función de coste obtenida a partir del análisis del bytecode es:

$$gas(powerOfTwo(x)) = 600 + 367 * x$$

De manera experimental se ha observado en las ejecuciones de Remix, que el consumo de *gas* de la parte iterativa es 367, coincidiendo con el obtenido en el análisis formal. Mientras que el estático es de 615, como en el caso anterior, se cuenta con 9 unidades de *gas* adicional correspondiente al *MSTORE* del primer bloque. Las seis unidades de *gas* restantes provienen de la expansión de dos palabras, resultado de la operación *MSTORE* sobre la posición 128.

$$mem\_size\_words = \lceil \frac{128 + 32}{32} \rceil = 5$$

$$mem\_cost = \lfloor \frac{5^2}{512} \rfloor + (3 * 5) = 15$$

$$mem\_expansion\_cost = new\_mem\_cost - last\_mem\_cost = 15 - 9 = 6$$

### 4.4.3. Coste en función del tamaño de la entrada

En el tercer caso se intentará replicar un caso muy repetido relacionado con el tamaño de la entrada y la cantidad de *gas* consumido. La función $paramLength(list)$ recibe una lista como parámetro de entrada y solo realiza la devolución de la longitud de la misma.

Este caso es muy similar al expuesto con anterioridad, pues a nivel de bytecode se distingue un conjunto de instrucciones iterativas, frente al otro conjunto secuencial. Véase las tablas resultantes en Tabla A.3 y Tabla A.4.

La función de coste obtenida a partir del análisis del bytecode es:

$$gas(paramLength(list)) = 1081 + 223 * list.length$$

En la práctica, simulando transacciones con Remix, se ha observado que el consumo de *gas* de la parte iterativa es 226 y el estático de 1099. Para la parte iterativa, la diferencia de 3 unidades de *gas* proviene de la operación *MSTORE* que se realiza por cada elemento de la lista, ampliando la memoria una palabra –32 bytes– por cada elemento. En el caso estático, la ejecución expande la memoria hasta el byte 160, por lo tanto:

$$mem\_size\_words = \lceil \frac{160 + 32}{32} \rceil = 6$$

$$mem\_cost = \lfloor \frac{6^2}{512} \rfloor + (3 * 6) = 18$$

$$mem\_expansion\_cost = new\_mem\_cost - last\_mem\_cost = 18 - 0 = 18$$

Como la expansión de memoria se realiza de manera iterativa, en función del número de elementos de la lista, hay que tener en cuenta que el número de palabras en memoria puede variar, y por tanto $mem\_size\_words^2 >= 512$. Por ello, para obtener la función de coste para cualquier número de elementos, debemos calcular la expansión de memoria aparte:

$$gas(paramLength(list)) = 1081 + 223 * list.length + mem\_cost$$

$$mem\_size\_words = \lceil \frac{160 + 32 + 32 * list.length}{32} \rceil$$

$$mem\_cost = \lfloor \frac{mem\_size\_words^2}{512} \rfloor + (3 * mem\_size\_words)$$

### 4.4.4. Coste accediendo al storage

Como cuarto y último caso, se va a analizar el caso más observado durante este estudio, funciones que acceden y modifican el *storage*. La función $set(x)$ recibe como parámetro un entero sin signo, el cual se escribe en la variable del contrato, persistiendo en la red de *Ethereum*.

La evaluación del bytecode ha resultado en la siguiente función de coste, donde se puede apreciar que se desconoce el consumo dinámico de la operación *SSTORE*. Véase la tabla resultante en Tabla A.5.

$$gas(set(x)) = 427 + dynamic\_gas$$

El consumo dinámico de *gas* se compone por las 9 unidades anteriormente citadas, sumado al coste asociado a la operación *SSTORE*, que se desarrolla en los siguientes casos:

$$dynamic\_storage\_gas = \begin{cases} 2200 & \text{if } new\_val = orig\_val \\ 22100 & \text{if } orig\_val = 0 \\ 5000 & \text{if } orig\_val \neq 0 \wedge new\_val \neq 0 \\ 200 & \text{if } orig\_val \neq 0 \wedge new\_val = 0 \end{cases}$$

Esta función en particular solo realiza una escritura en *storage* sobre la misma variable, es por ello que conlleva la simplificación ($current\_val = orig\_val$) del coste de las operaciones *SSTORE*, descrito en Código B.1.

Empíricamente se han realizado pruebas a conciencia simulando los casos descritos. En todos ellos, la función de coste no determinista obtenida, se ajusta a la perfección. Este resultado sugiere que conociendo el estado del *storage* calcular el coste de *gas* de una transacción, es trivial.

# Capítulo 5

# Conclusión

Los presentes análisis permiten concluir que, el empleo de técnicas de aprendizaje automático en la predicción de consumo de *gas* en transacciones en *Ethereum* a partir del *input* de la transacción no es suficiente para determinar una predicción veraz en los casos que se modifica el *storage*.

La escritura en *storage* supone una incertidumbre que propicia unos consumos de *gas* que no son predecibles únicamente contando con el *input*, pues depende del estado del *storage* en el momento de ejecutar de la transacción, y esto no es predecible en un entorno distribuido como es el *blockchain*. El error obtenido no resulta admisible para los objetivos del proyecto.

Se ha determinado que para los casos en lo que no interviene el *storage*, es trivial devolver una función de coste a partir de los parámetros de entrada, pues ejecuta siempre el mismo número de instrucciones y con el mismo consumo para idénticos parámetros de entrada.

Por otro lado, cabe resaltar que para algunos casos donde se realizan múltiples operaciones sobre un *input* muy grande, adquiere mayor relevancia llegando a opacar en algunos casos a la cantidad de *gas* que supone las escrituras en *storage*. Para estos casos se asume un margen de error, es decir, no devuelve una función de coste fiel, pero sí realiza aproximaciones interesantes que pueden ser objetivos de estudio.

Otra potencial e interesante línea de estudio sería contemplar entrenar un modelo con la información del estado anterior del *storage*. Esto dotaría al modelo de la capacidad de determinar el consumo de operaciones de escritura y lectura en el *storage*, y por tanto, lograr el objetivo del presente estudio. Esto presenta dos retos en particular: la obtención de la información del estado anterior, y la condición de carrera que se daría al existir la posibilidad de que el estado se modifique después de la lectura, y por tanto carece de veracidad la predicción obtenida, pues el estado podría haber sido modificado desde que se lanza la predicción hasta la ejecución.

Es por ello que para este caso en particular una herramienta desarrollada utilizando métodos formales sería más adecuada, por predecir el coste computacional utilizando técnicas como la función de coste, el invariante, etc. –visto en Fundamentos de Algoritmia y Métodos Algorítmicos en Resolución de Problemas–

# Bibliografía


[1] Massimo Di Pierro. «What is the blockchain?» En: *Computing in Science & Engineering* 19.5 (2017), págs. 92-95.

[2] *Ethereum*. 8 de mayo de 2024. URL: https://ethereum.org/.

[3] Satoshi Nakamoto. «Bitcoin: A Peer-to-Peer Electronic Cash System». En: (2008).

[4] Gavin Wood. «Ethereum: A secure decentralised generalised transaction ledger». En: *Ethereum project yellow paper* (4 de mar. de 2024).

[5] Vitalik Buterin et al. «A next-generation smart contract and decentralized application platform». En: *Ethereum White Paper* 3.37 (2014).

[6] Nick Szabo. «Smart contracts: building blocks for digital markets». En: *EXTROPY: The Journal of Transhumanist Thought,(16)* 18.2 (1996).

[7] *Transacciones Ethereum*. Ene. de 2024. URL: https://ethereum.org/es/developers/docs/transactions/.

[8] *Proof-of-stake*. Mar. de 2024. URL: https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/.

[9] *Opcodes para la EVM*. Abr. de 2024. URL: https://ethereum.org/es/developers/docs/evm/opcodes/.

[10] Wolflo. *Appendix - Dynamic Gas Costs*. Nov. de 2021. URL: https://github.com/wolflo/evm-opcodes/blob/main/gas.md.

[11] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez y Albert Rubio. «*Don't run on fumes* - Parametric gas bounds for smart contracts». En: *J. Syst. Softw.* 176 (2021), pág. 110923. DOI: 10.1016/J.JSS.2021.110923.

[12] *Google Cloud BigQuery*. URL: https://cloud.google.com/bigquery.

[13] *BigQuery crypto_ethereum*. Dataset. Sep. de 2022. URL: https://console.cloud.google.com/bigquery?ws=!1m4!1m3!3m2!1sbigquery-public-data!2scrypto_ethereum.

[14] Warren E. Buffett. *The Superinvestors of Graham-and-Doddsville*. 1984. URL: https://etherscan.io/.

[15] Elvira Albert. *The Costa group*. 2024. URL: https://costa.fdi.ucm.es/web/.

[16] *BigQuery crypto-ethereum contracts table*. Dataset. 2024. URL: https://console.cloud.google.com/bigquery?ws=!1m5!1m4!4m3!1sbigquery-public-data!2scrypto_ethereum!3scontracts.

[17] *BigQuery crypto-ethereum transactions table*. Dataset. 2024. URL: https://console.cloud.google.com/bigquery?ws=!1m5!1m4!4m3!1sbigquery-public-data!2scrypto_ethereum!3stransactions.

[18] Solidity Team. *Solidity*. 2024. URL: https://soliditylang.org/.

[19] Weiliang Li. *web3-input-decoder python library*. 2024. URL: https://pypi.org/project/web3-input-decoder/.

[20] *Scikit-learn Linear Regression*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html.

[21] *Scikit-learn Gradient Boosting Regressor*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingRegressor.html#sklearn.ensemble.GradientBoostingRegressor.

[22] Sewall Wright. «Correlation and causation». En: *Journal of agricultural research* 20.7 (1921).

[23] Weili Chen, Tuo Zhang, Zhiguang Chen, Zibin Zheng y Yutong Lu. «Traveling the token world: A graph analysis of ethereum erc20 token ecosystem». En: *Proceedings of The Web Conference 2020*. 2020, págs. 1411-1421.

[24] *The Native IDE for Web3 Development*. 2024. URL: https://remix.ethereum.org/.

[25] *Remix Docs*. 2024. URL: https://remix-ide.readthedocs.io/en/latest/run.html#environment.

[26] Elvira Albert, Pablo Gordillo, Benjamin Livshits, Albert Rubio e Ilya Sergey. «Ethir: A framework for high-level analysis of ethereum bytecode». En: *International symposium on automated technology for verification and analysis*. Springer. 2018, págs. 513-520.

# Apéndice A

# Anexo

Todos los materiales desarrollados por los integrantes del trabajo se encuentran disponibles para su uso y visualización en la plataforma GitHub bajo el siguiente enlace: https://github.com/Charlisteeron/TFG. A continuación se facilita material que se considera relevante para la mejor comprensión del trabajo:

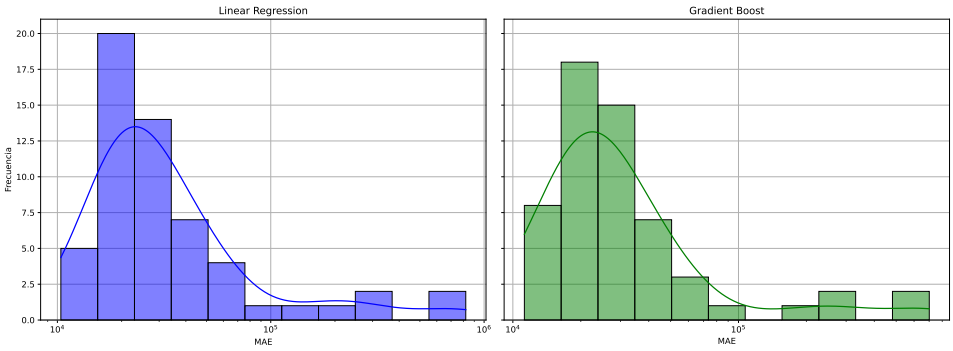



Figura A.1: Resultados MAE $\geq$ 1000

| Opcode | Consumo base | Apariciones | Consumo |
|---|---|---|---|
| JUMP | 8 | 16 | 128 |
| PUSH2 | 3 | 22 | 66 |
| JUMPI | 10 | 6 | 60 |
| POP | 2 | 23 | 46 |
| SWAP1 | 3 | 12 | 36 |
| SWAP2 | 3 | 11 | 33 |
| PUSH1 | 3 | 10 | 30 |
| DUP2 | 3 | 9 | 27 |
| JUMPDEST | 1 | 22 | 22 |
| PUSH0 | 2 | 10 | 20 |
| DUP1 | 3 | 6 | 18 |
| ADD | 3 | 4 | 12 |
| DUP3 | 3 | 4 | 12 |
| ISZERO | 3 | 3 | 9 |
| SUB | 3 | 3 | 9 |
| DUP5 | 3 | 3 | 9 |
| SWAP3 | 3 | 3 | 9 |
| MSTORE | 3 | 2 | 6 |
| LT | 3 | 2 | 6 |
| CALLDATALOAD | 3 | 2 | 6 |
| EQ | 3 | 2 | 6 |
| DUP4 | 3 | 2 | 6 |
| MLOAD | 3 | 2 | 6 |
| CALLDATASIZE | 2 | 2 | 4 |
| SHR | 3 | 1 | 3 |
| PUSH4 | 3 | 1 | 3 |
| SLT | 3 | 1 | 3 |
| DUP6 | 3 | 1 | 3 |
| CALLVALUE | 2 | 1 | 2 |
| RETURN | 0 | 1 | 0 |
| Total | | | 600 |

Tabla A.1: Opcodes $powerOfTwo(x)$ no iterativo

| Opcode | Consumo base | Apariciones | Consumo |
|---|---|---|---|
| JUMP | 8 | 11 | 88 |
| PUSH2 | 3 | 14 | 42 |
| POP | 2 | 17 | 34 |
| SWAP1 | 3 | 11 | 33 |
| JUMPI | 10 | 3 | 30 |
| SWAP2 | 3 | 10 | 30 |
| DUP3 | 3 | 8 | 24 |
| DUP2 | 3 | 5 | 15 |
| JUMPDEST | 1 | 13 | 13 |
| PUSH0 | 2 | 5 | 10 |
| DUP1 | 3 | 3 | 9 |
| DUP4 | 3 | 2 | 6 |
| ISZERO | 3 | 2 | 6 |
| SWAP3 | 3 | 2 | 6 |
| ADD | 3 | 2 | 6 |
| LT | 3 | 1 | 3 |
| GT | 3 | 1 | 3 |
| PUSH32 | 3 | 1 | 3 |
| SUB | 3 | 1 | 3 |
| PUSH1 | 3 | 1 | 3 |
| Total | | | 367 |

Tabla A.2: Opcodes $powerOfTwo(x)$ iterativo

| Opcode | Consumo base | Número de apariciones | Consumo total |
|---|---|---|---|
| JUMP | 8 | 24 | 192 |
| JUMPI | 10 | 13 | 130 |
| PUSH2 | 3 | 37 | 111 |
| POP | 2 | 37 | 74 |
| SWAP1 | 3 | 20 | 60 |
| PUSH1 | 3 | 19 | 57 |
| DUP3 | 3 | 15 | 45 |
| DUP2 | 3 | 13 | 39 |
| SWAP2 | 3 | 13 | 39 |
| ADD | 3 | 12 | 36 |
| JUMPDEST | 1 | 34 | 34 |
| DUP1 | 3 | 9 | 27 |
| PUSH0 | 2 | 13 | 26 |
| ISZERO | 3 | 7 | 21 |
| DUP5 | 3 | 6 | 18 |
| DUP4 | 3 | 6 | 18 |
| MSTORE | 3 | 4 | 12 |
| PUSH4 | 3 | 4 | 12 |
| EQ | 3 | 4 | 12 |
| GT | 3 | 4 | 12 |
| MLOAD | 3 | 4 | 12 |
| SWAP3 | 3 | 4 | 12 |
| MUL | 5 | 2 | 10 |
| LT | 3 | 3 | 9 |
| CALLDATALOAD | 3 | 3 | 9 |
| SUB | 3 | 3 | 9 |
| PUSH8 | 3 | 3 | 9 |
| SLT | 3 | 2 | 6 |
| DUP6 | 3 | 2 | 6 |
| CALLDATASIZE | 2 | 2 | 4 |
| SHR | 3 | 1 | 3 |
| DUP7 | 3 | 1 | 3 |
| NOT | 3 | 1 | 3 |
| AND | 3 | 1 | 3 |
| OR | 3 | 1 | 3 |
| SWAP4 | 3 | 1 | 3 |
| CALLVALUE | 2 | 1 | 2 |
| RETURN | 0 | 1 | 0 |
| Total | | | 1081 |

Tabla A.3: Opcodes $paramLength(list)$ no iterativo

| Opcode | Consumo base | Número de apariciones | Consumo total |
|---|---|---|---|
| JUMP | 8 | 7 | 56 |
| PUSH2 | 3 | 9 | 27 |
| DUP2 | 3 | 8 | 24 |
| JUMPI | 10 | 2 | 20 |
| POP | 2 | 9 | 18 |
| SWAP1 | 3 | 4 | 12 |
| JUMPDEST | 1 | 8 | 8 |
| SWAP2 | 3 | 2 | 6 |
| DUP5 | 3 | 2 | 6 |
| PUSH1 | 3 | 2 | 6 |
| ADD | 3 | 2 | 6 |
| PUSH0 | 2 | 2 | 4 |
| LT | 3 | 1 | 3 |
| ISZERO | 3 | 1 | 3 |
| DUP1 | 3 | 1 | 3 |
| DUP9 | 3 | 1 | 3 |
| DUP3 | 3 | 1 | 3 |
| CALLDATALOAD | 3 | 1 | 3 |
| EQ | 3 | 1 | 3 |
| SWAP3 | 3 | 1 | 3 |
| MSTORE | 3 | 1 | 3 |
| SWAP4 | 3 | 1 | 3 |
| Total | | | 223 |

Tabla A.4: Opcodes $paramLength(list)$ iterativo

| Opcode | Consumo base | Número de apariciones | Consumo total |
|---|---|---|---|
| JUMP | 8 | 10 | 80 |
| JUMPI | 10 | 7 | 70 |
| PUSH2 | 3 | 17 | 51 |
| POP | 2 | 13 | 26 |
| DUP2 | 3 | 7 | 21 |
| PUSH1 | 3 | 6 | 18 |
| DUP1 | 3 | 6 | 18 |
| SWAP1 | 3 | 6 | 18 |
| SWAP2 | 3 | 5 | 15 |
| JUMPDEST | 1 | 14 | 14 |
| PUSH0 | 2 | 6 | 12 |
| EQ | 3 | 4 | 12 |
| PUSH4 | 3 | 3 | 9 |
| ISZERO | 3 | 2 | 6 |
| CALLDATALOAD | 3 | 2 | 6 |
| SUB | 3 | 2 | 6 |
| ADD | 3 | 2 | 6 |
| DUP3 | 3 | 2 | 6 |
| DUP5 | 3 | 2 | 6 |
| SWAP3 | 3 | 2 | 6 |
| CALLDATASIZE | 2 | 2 | 4 |
| MSTORE | 3 | 1 | 3 |
| LT | 3 | 1 | 3 |
| SHR | 3 | 1 | 3 |
| SLT | 3 | 1 | 3 |
| DUP6 | 3 | 1 | 3 |
| CALLVALUE | 2 | 1 | 2 |
| SSTORE | 0 | 1 | ? |
| STOP | 0 | 1 | 0 |
| Total | | | 427 + ? |

Tabla A.5: Opcodes *set*()

# Apéndice B

# Introduction

## B.1. Context

In today's digital age, technological advances have dramatically transformed the way we interact with our surroundings. Two of the fields that have advanced the most at this juncture, and therefore have had the greatest impact, are *Machine Learning* and *Blockchain*[1] technology.

Digital evolution gives access to a massive amount of data. The correct processing and study of this data permits to obtain trends and predict results. In this context, *Machine Learning*, a branch of artificial intelligence, has proven to be one of the most important technological bases in the evolution of systems capable of learning thanks to the access to large volumes of data, being indispensable the increase in computational speed. This technology is used in multiple sectors due to its ability to predict complex patterns, such as medical diagnostics, fraud detection and robotics, among others.

As any major change requires the ability to adapt, the digital era, in particular, has challenged centralized models. These models are susceptible to manipulation and cyber-attacks, which with the increase in computing power pose a great threat. In response, *blockchain* networks are emerging as a disruptive technology that promises to revolutionize the way transactions are handled, proving to be a clear competitor to traditional banking.

Unlike traditional systems, which store data on central servers, the blockchain is decentralized, and it distributes data across a network. In addition, transactions are transparent and can be audited by user. Blockchain technology reduces costs by eliminating the middleman and increases efficiency through more automated and transparent processes[1].

Immutability is another key concept to contextualize this model, as transactions once added to the blockchain, cannot be altered or deleted, thus guaranteeing the integrity of the records, without the need for a centralized authority[1].

## B.2. Ethereum's blockchain

*Ethereum*[2] is one of the most popular blockchain platforms, distinguished by its ability to go beyond simple transactions, such as those performed by *Bitcoin*[3], and allows code execution via a *quasi-Turing-complete* virtual machine, the *Ethereum Virtual Machine* –known as ***EVM***–, described in the *Yellow Paper*[4]. This makes *Ethereum* a complete and decentralized computational environment that has opened up new possibilities by enabling the execution of programs, known as smart contracts.

> *What Ethereum intends to provide is a blockchain with a built-in fully fledged Turing-complete programming language that can be used to create çontracts"that can be used to encode arbitrary state transition functions, allowing users to create any of the systems described above, as well as many others that we have not yet imagined, simply by writing up the logic in a few lines of code.*
>
> —*Vitalik Buterin, inventor of Ethereum [5]*

It is a trailblazer in introducing concepts such as ***gas***, which is a the unit of measurement for the computational cost of executing a method of the smart contract on the blockchain. Thus, the *gas* is born from the need to calculate the cost of each execution, since the implementation of a *Turing-complete* virtual machine that allows the execution of different instructions, results in that not all executions consume the same amount of *gas*. The *gas* cost model will be explained in Section 1.2.4, as it is essential to understand this concept in order to have a deeper insight of the project

### B.2.1. Smart contracts

The concept of smart contract was defined by Nick Szabo, in 1996 [6]. In particular, in *Ethereum*, it represents a program that runs on the *blockchain*. Hence, a set of code and data that are stored in an *Ethereum* account, can be the target of transactions. However they are not controlled by a user, they act according to the code, setting rules that are executed automatically.

The global variables of a smart contract reside in the ***storage***, a persistent distributed storage belonging to the *blockchain*. The *storage* is structured in a dictionary where each key refers to a *slot*, a set of 32 bytes where the variables are stored.

The ***ABI***, which stands for "*Application Binary Interface*", is the standard that defines how to interact with the contract. It details the available functions, their arguments, the data types and how they should be coded to interact with the contract. Each function is identified by the signature, an encoding of the first four bytes using the *hash Keccak-256* function over the name of the function and the type of its arguments.

The following example is the construction of the signature for a well-known function of the *ERC20* interface, the function "transfer(address _to, uint256 _value)" with the variable names and the spaces between the parentheses removed:

$$Keccak\text{-}256(transfer(address, uint256)) = 0xA9059CBB\ldots$$

By taking the first four bytes (8 characters in hexadecimal) from the previous result, we obtain the resulting signature for that function, which will have to be referenced in the *input* of a transaction to be able to execute it.

Numerous contracts implement the *proxy* pattern, using an intermediate contract, with the declaration of functions, where they delegate the calls to. This implementation is used to separate the data from the business logic. This poses a problem when processing the transaction *input*, since the *ABI* of the *proxy contract* does not include the functions defined by the implementation contract, calling the particular function *fallback*, which is executed when an undeclared function is called.

### B.2.2. Transactions

Transactions are the element by which users interact with the network, updating its the state. Three different types of transactions are defined in *Ethereum*: regular (a transaction from one wallet to another), contract creation and contract execution.

Transactions have the following common fields:

| Column | Description |
|---|---|
| type | EIP-2718 transaction type. |
| nonce | A scalar value equal to the number of transactions sent by the sender. |
| gasLimit | A scalar value equal to the maximum amount of gas that should be used in executing this transaction. This is paid up-front, before any computation is done and may not be increased later. |
| to | The 160-bit address of the message calls recipient or, for a contract creation transaction, ∅ |
| value | A scalar value equal to the number of Wei to be transferred to the message call's recipient or, in the case of contract creation, as an endowment to the newly created account. |

Tabla B.1: *Ethereum* transaction's common fields [4]

Contract creation transactions, regardless of the transaction type, contain a *init* field with an unlimited byte array with the *EVM* instructions to initialize the contract. On the other hand, contract call transactions have a *data* field, an unlimited byte array to specify the *input* of the call. The project focuses on the second ones in particular[4].

The first four bytes of the *input_data* field specify the function being called. This value refers to the previously defined concept of signature.

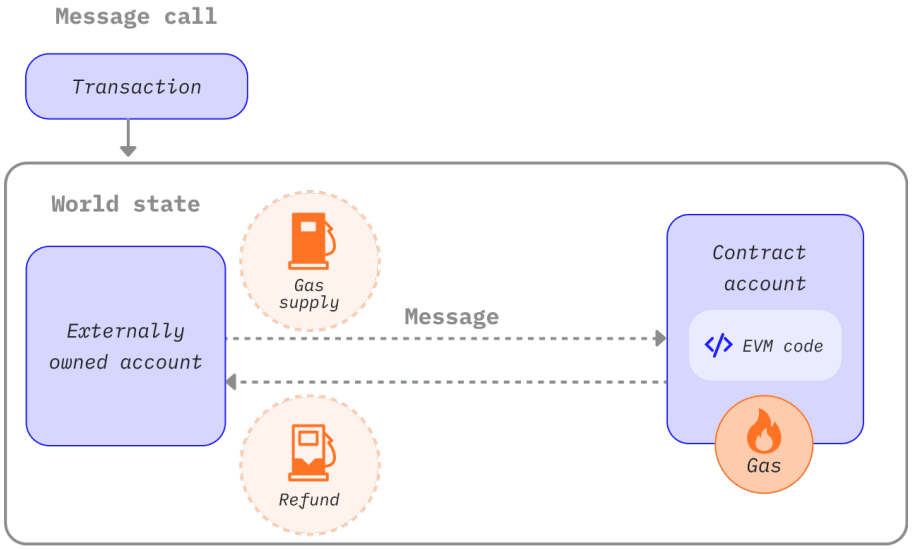



Figura B.1: *Ethereum* transition diagram[7]

For a transaction to be successful, the amount of *gas* sent by the user must be greater than or equal to the amount of *gas* required to execute the instructions. Thus, the amount of *gas* not used will be reimbursed to the user.

### B.2.3. Blocks

A block in the *Ethereum blockchain* is a collection of transactions that have been verified and consolidated into a group. Each block also has the *hash* of the previous block which guarantees the security and consistency of the *blockchain*.

The interconnection of the blocks through the *hash* of the previous block forms a continuous and secure chain. Each new block added to the chain further confirms and supports integrity, strengthening the network and ensuring that it works as a decentralized and trusted platform.

To validate a block and add it to the blockchain *Ethereum* uses the so-called ***Proof of Stake***, (***PoS***) [8], where users block part of their funds in order to have the right to validate transactions. In this way, validators are chosen at random, based on the number of tokens they have locked.

### B.2.4. Cost model

The cost model is defined by the concept of *gas*, which measures the proportion of resources required to execute instructions in the *EVM*. Thus, any code fragment of has a cost associated with it in terms of *gas*.

Transactions have a value of ***gasLimit*** which emitters pay in Ether, adjusted by the price of *gas*, which varies according to network congestion and user priorities. The system also includes basic fees and gratuities to encourage validators to include transactions. The objective of the model is to balance resource demand and encourage efficient use of storage.

The amount of *gas* per instruction is constant for most instructions and is stipulated in the documentation of *Ethereum*, where the costs for each type of operation are listed.

| Opcode | Name | Description | Gas |
|---|---|---|---|
| 0x01 | ADD | Addition | 3 |
| 0x02 | MUL | Multiplication | 3 |
| 0x57 | JUMPI | Condicional jump | 10 |

Tabla B.2: *EVM* opcodes subset [9]

In *Ethereum* there are two types of storage, the volatile one (*memory*), which includes some of the local variables of a function, and the *storage*, also known as state variables, which consist of the global variables and persist after the execution of a call. It is crucial to distinguish them as the computational cost is very high in the case of *storage*.

Memory operations, *MLOAD* and *MSTORE* cost 3 units of *gas* and additionally an amount that depends on the memory expansion in bytes with respect to the previous state. Other operations such as *RETURN*, *CALLDATACOPY*, etc. also add to their static cost the cost associated with memory expansion. The cost of memory expansion is determined by the following formulas [4]:

$$\text{mem\_size\_words} = \left\lceil \frac{\text{mem\_byte\_size} + 32}{32} \right\rceil$$

$$\text{mem\_cost} = \left\lfloor \frac{\text{mem\_size\_words}^2}{512} \right\rfloor + (3 * \text{mem\_size\_words})$$

$$\text{mem\_expansion\_cost} = \text{new\_mem\_cost} - \text{last\_mem\_cost}$$

The gas consumed by memory operations represent an insignificant cost compared to the operations that interact with the *storage*. This is because the *storage* is persistent and is set in the *blockchain*.

The amount of *gas* in write operations on *storage* is complex depending on several factors, such as whether it is the first time a value is stored at an address, whether an existing value is being updated, and whether the value is being written to or over a zero. Additionally, there is a *refund* associated with value deletion to keep the *blockchain* clean and efficient. The cost of the *SSTORE* operation is defined by the following pseudocode:

```
/* orig_val: value of the storage slot before the current transaction
current_val: current value of the storage slot new_val: value to be set in
    the
storage slot cold access: first access to a storage slot via SLOAD or SSTORE
     in
the current transaction */

gas_cost = 0
gas_refound = 0

if (context_addr, target_storage_key) not in touched_storage_slots
    // cold access
    gas_cost += 2100
if new_val == current_val
    // no op
    gas_cost += 100
else
    if current_val == orig_val
        // clean slot, not yet updated in current execution context
        if orig_val == 0
            // slot started zero, being changed to nonzero
            gas_cost += 20000
        else
            // slot started nonzero, being changed
            gas_cost += 2900
            if new_val == 0
                gas_refound += 4800
    else
        // dirty slot, already updated in current execution context
        gas_cost += 100
        if orig_val != 0
            // execution context started with a nonzero value in slot
            if current_val == 0
                // slot started zero, being changed to nonzero
                gas_refound -= 4800
            else if new_val == 0
                // slot started nonzero, being changed to zero
                gas_refound += 4800
        if new_v44al == orig_val
            // slot is reset to the value it started with
            if orig_val == 0
                // slot started zero, being reset to zero
                gas_refound += 1990
            else
                //slot started nonzero, now reset to orig
                gas_refound += 2800
```

Código B.1: Pseudocode for *SSTORE* opcode gas consumption [10]

On the other hand, for the *SLOAD* read operation, the logic is less complex, distinguishing two cases: the first one when a *cold access* is performed, being 2100 units of *gas*, or otherwise 100 units[10].

Apart from the costs of modifying and reading the *storage*, and of the instructions, there are other supplements of *gas* that are added to the transaction and that are worth mentioning, such as the consumptions of calling a transaction of 21000 units of *gas* and others that depend on the *input*, where each byte set to 0 has a cost of 4 units as opposed to the 16 units consumed when the byte is other than 0 [4].

## B.3. Goal

In 2023 more than ten million transactions resulted in failures, where 5 percent were due to the lack of the established amount of *gas*[1]. It gives rise to question the origin and reason for this cause.

Ethereum executions require the establishment of an amount of gas, which must be greater than or equal to the consumption of the transaction for its correct development. The upper bound obtained by the Solidity compiler does not allow users to accurately predict the amount of gas required, returning infinity in some cases. Since the gas consumption per transaction is defined by the cost model -explained in Section 1.2.4–, it is possible to predict the consumption of a transaction.

As mentioned in the above context, the objective of the study is to predict from the transaction *input* the amount of *gas* needed to execute a transaction in *Ethereum*, based on machine learning techniques. To do this, public information on the blockchain about transactions and consumed *gas* will be used to build machine learning models to predict the *gas* required to execute new transactions.

Nowadays, there are tools developed using formal methods, such as *GASTAP*[11], which are capable of calculating an upper bound on the consumption of *gas*, being very useful because in the event that a transaction runs out of *gas*, it will not be executed, that transaction will not be executed and that *fee* will be lost. The aim of these tools, however, differ from the objective stated in the present project since they always return the maximum amount of *gas* that can be consumed by the transactions always. The objective of this work is to try to predict the most adjusted *gas* quantity. Both applications should complement each other as they solve different problems.

It is worth mentioning that during the course of the study, some previous studies based on the application of machine learning techniques on *blockchain* were found, most of them focused on predicting price, both cryptocurrencies and *gas*, neither were found that focus on predicting the amount of *gas* per instruction.

[1]Data obtained from BigQuery on May 2, 2024.

# Apéndice C

# Conclusion

The described analyses allows to conclude that the use of machine learning techniques in the prediction of *gas* consumption in *Ethereum* transactions from the transaction *input* is not sufficient to determine a truthful prediction in cases where the *storage* is modified.

Writing to *storage* involves an uncertainty that leads to *gas* consumption that cannot be predicted by the *input* alone, since it depends on the state of the *storage* at the time the transaction is executed, and this is not predictable in a distributed environment such as the *blockchain*. The error obtained is not admissible for the objectives of the project.

It has been determined that for cases where *storage* is not involved, it is trivial to return a cost function from the *input* parameters, since it always executes the same number of instructions and with the same consumption for identical *input* parameters.

On the other hand, it should be noted that for some cases where multiple operations are performed on a very large input, it acquires greater relevance, as sometimes it overshadows the amount of *gas* involved in the writings in *storage*. For these cases a margin of error is assumed, i.e., it does not return a faithful cost function, but it makes interesting approximations that can be studied.

Another potential and interesting line of study would be to consider training a model with the information of the previous state of the *storage*. It would provide the model with the ability to determine the consumption of write and read operations in the *storage*, and therefore, achieve the objective of this study. It presents two challenges in particular: obtaining the information of the previous state, and the race issue that would be given by the possibility that the state is modified after the read, and therefore the prediction obtained lacks veracity, since the state could have been modified from the time the prediction is launched until the execution.

That is why for this particular case a tool developed using formal methods would be more appropriate, for predicting the computational cost using techniques that involved cost functions, invariants, etc. –seen in Fundamentals of Algorithms and Algorithmic Methods in Problem Solving–.

# Apéndice D

# Trabajo realizado por los integrantes del trabajo

Durante el desarrollo del trabajo ambos integrantes han participado a partes iguales y de manera conjunta, realizando todas las actividades de forma cooperativa. Es por ello que no se considera oportuno realizar una división de tareas en función del integrante, por lo que se procede a desglosar el conjunto de las actividades desarrolladas:

- Estudio e investigación sobre el paradigma Blockchain.
- Estudio e investigación sobre el paradigma de Machine Learning.
- Búsqueda de fuentes científicas para la comprensión del proyecto. Recopilando artículos, tesis y libros relevantes.
- Selección de los lenguajes de programación a utilizar en el proyecto. Evaluando diferentes lenguajes de programación y frameworks adecuados, tomando la compatibilidad con Ethereum y machine learning.
- Aprendizaje en el lenguaje de programación Solidity. Consumiendo tutoriales en línea para dominar la sintaxis.
- Desarrollar ejemplos prácticos de smart contracts para entender su funcionamiento y optimización.
- Aprendizaje de LaTex para la redacción de la memoria.
- Estudio de la Ethereum Virtual Machine. Analizando el funcionamiento interno de la EVM y cómo ejecuta los contratos inteligentes y estudiando la arquitectura de la EVM y cómo se relaciona con el consumo de gas.
- Análisis del modelo de coste para distintos opcodes. Revisando la documentación oficial de Ethereum sobre el coste de los opcodes y realizando experimentos para medir y comparar el consumo de gas de diferentes opcodes.
- Creación de un repositorio que permita control de versiones en la nube donde subir el material del proyecto (Git).
- Creación de un repositorio que permita control de versiones en la nube para compartir la memoria del proyecto con los tutores (Subversion).

- Preparación y diseño de la estructura de los repositorios.
- Creación de un entorno virtual para aislar las dependencias y poder trabajar en distintos equipos (Poetry).
- Búsqueda de fuentes para la extracción de los datos. Identificando APIs y bases de datos que proporcionen datos públicos de transacciones de Ethereum y evaluando la calidad y la relevancia de los datos disponibles para el proyecto.
- Selección de parámetros necesarios de cada tabla. Definiendo los parámetros clave que se extraerán de las tablas de datos.
- Búsqueda de herramientas para el procesado y análisis de datos masivos.
- Formación para la utilización de la máquina local del departamento de COSTA. Obtener acceso a la máquina local del departamento y configurar el entorno de trabajo.
- Búsqueda de herramientas para el seguimiento y visualización de la blockchain de Ethereum.
- Obtención de las claves necesarias para la utilización de las herramientas.
- Realización de un script en SQL para la extracción de los 100 contratos más usados en BigQuery.
- Realización de un script en Python para la validación de los contratos.
- Realización de un script en Python para la extracción de transacciones asociadas a los contratos más usados.
- Realización de un script en Python para la extracción los ABIs de cada contrato
- Actualización del script para obtener los ABIs de los contratos de implementación.
- Desarrollo del modelo aplicando principios SOLID que permitan un desarrollo eficiente, mantenible y escalable y que a su vez permita la programación multiproceso de forma paralela
- Evaluación de las distintas aproximaciones a la hora de decodear el input de las transacciones. Probando diferentes técnicas de decodificación de inputs de transacciones y comparando la eficiencia y precisión de cada técnica.
- Investigación de herramientas que permitan hacer un decode del input a partir de los datos disponibles.
- Preparación del código que realice las distintas aproximaciones del input.
- Análisis y exploración del problema.
- Búsqueda de modelos de aprendizaje automático.
- Búsqueda de métricas de evaluación.

- Preparación de los distintos conjuntos de datos aplicando la técnica Train Test Split. Dividiendo los datos disponibles en conjuntos de entrenamiento y prueba
- Preparación de un diccionario de acceso concurrente que permita guardar los modelos entrenados con la métrica de evaluación, y un conjunto de datos para la evaluación, ya procesados.
- Preparación del código que realice el entrenamiento de los distintos modelos y los almacene en el diccionario.
- Generación de gráficas para la evaluación de los modelos.
- Evaluación de los modelos para cada métrica.
- Análisis de los posibles motivos de los malos resultados.
- Creación de un script para la extracción del bytecode de los contratos.
- Compilación de los distintos bytecodes y búsqueda de instrucciones SSTORE.
- Búsqueda de las funciones y signaturas pertenecientes a la IERC20. Identificando y extrayendo las funciones y signaturas estándar definidas en la IERC20.
- Filtrado de signaturas pertenecientes a la IERC20 provenientes del conjunto de datos.
- Evaluación de las métricas de las signaturas resultantes.
- Análisis exhaustivo de las métricas y gráficas del modelo favorable y desfavorable perteneciente a la IERC20.
- Búsqueda y análisis del código ejecutado para el caso favorable y desfavorable.
- Selección y distinción de los modelos más efectivos, agrupando por características similares.
- Análisis de código para casos constantes.
- Análisis de código para casos con alta correlación entre el consumo y la longitud de la entrada.
- Desarrollo de un smart contract que reúna las casuísticas identificadas para poder analizarlas.
- Desplegado en máquina virtual de Remix. Utilizando la herramienta Remix para desplegarlo y probar el funcionamiento del contrato en un entorno de prueba controlado.
- Obtención del bytecode asociado.
- Generación de los opcodes a partir del bytecode.
- Obtención del grafo de control de flujo asociado al smart contract desarrollado, utilizando el framework de Ethir.

- Desarrollo de un script que cuenta el número de apariciones de cada opcode en una traza del grafo de control de flujo y el consumo total asociado a cada opcode.
- Análisis y evaluación de la función de coste obtenida a partir del grafo de control de flujo y comparación con los resultados empíricos. Para el caso constante.
- Análisis y evaluación de la función de coste obtenida a partir del grafo de control de flujo y comparación con los resultados empíricos. Para el caso que depende de un parámetro de entrada.
- Análisis y evaluación de la función de coste obtenida a partir del grafo de control de flujo y comparación con los resultados empíricos. Para el caso que depende del tamaño de la entrada.
- Análisis y evaluación de la función de coste obtenida a partir del grafo de control de flujo y comparación con los resultados empíricos. Para la escritura en el storage.
- Preparación de la estructura de la memoria. Diseñando la estructura y el índice.
- Redacción de los distintos apartados de la memoria. Revisando y corregiendo cada apartado para asegurar la calidad del contenido.
- Aplicar las correcciones y sugerencias de los tutores a la memoria.
- Establecer un formato visual en la memoria. Realizando una revisión completa de la memoria para identificar errores.