SCIFEST NATIONAL FINAL 2024

Enhancing Data Security in AI Training using Zero-Knowledge Proofs for Secure Methodology Development.

Addison Carey

Celbridge Community School

Stand Number: 6

Project Report Book

Contents

Abstract (250 words)

Ensuring the security and privacy of sensitive data during AI and machine learning model training has become a critical challenge. Training AI models on datasets containing financial, medical, or other regulated information requires robust encryption methods and strict adherence to privacy regulations. As AI becomes increasingly integrated into essential services, the need for secure data-handling methodologies is more urgent than ever.

My project aims to address these concerns by integrating Zero-Knowledge Proofs (ZKPs) into the AI training process. ZKPs are cryptographic protocols that allow one party to prove possession of certain information to another party without revealing the information itself. By applying ZKPs, I aim to develop an algorithm that enables AI models to process encrypted data without exposing the raw data. This approach minimizes data exposure risks, ensuring privacy while maintaining model performance.

I started this project in **February 2024** and developed a basic ZKP algorithm that encrypts input data, which is then used through a simple linear regression model. My project also incorporates data visualization through a Flask-based interface. While my current model uses a small dataset and basic encryption, I am planning to scale to larger datasets and refining the algorithm using stronger encryption and cryptographic hash functions to enhance the zero-knowledge aspect further.

While ZKPs can be difficult to integrate into the world of AI, I believe my project demonstrates a novel application of ZKPs in AI, offering a potential solution to one of the biggest challenges in secure AI model training.

Introduction

My interest in Computer Science and Coding goes back to early Primary school. I started using Scratch and Micro:Bit but soon transitioned onto HTML/CSS and JS. Python was my introduction to the power of computer code to manipulate data.

Training AI models on sensitive data in a secure way is a significant challenge due to several reasons. First, handling sensitive data, such as personal or financial information, requires strict adherence to privacy regulations like GDPR or HIPAA, which adds complexity and legal considerations. Ensuring data security throughout its lifecycle - from collection and storage to processing and analysis, demands robust encryption, access controls, and secure data handling practices. The sheer volume of data needed for AI training means there's a higher risk of exposure or breaches, needing continuous monitoring and mitigation strategies. Lastly, the potential for adversarial attacks, where AI models can be manipulated or misled, underscores the importance of rigorous security measures in AI training pipelines. Balancing the benefits of AI with data protection and security remains a key challenge for organisations across various sectors.

With the rapid advancement of artificial intelligence technology, the usage of machine learning models is gradually becoming part of our daily lives. High-quality models rely not only on efficient optimisation algorithms but also on the training and learning processes built upon vast amounts of data and computational power. However, in practice, due to various challenges such as limited computational resources and data privacy concerns, users in need of models often cannot train machine learning models locally (Xing, et al. 2023).

My project involves the development of a ZKP algorithm in Python, to help ensure AI models can train on data securely, without access to the raw data through zero-knowledge. ZKPs are proof systems with two parties: a prover and a verifier, along with a challenge that gives users the ability to publicly share a proof of knowledge or ownership without revealing the details of it. In essence, they demonstrate knowledge of a fact without disclosing the details of that knowledge. They have shown that, it is feasible to demonstrate that some theorems are true without providing the slightest indication of why they are true.

For example, where a first party or a prover try to persuade the second party or verifier that the statement is true without sharing any hints or information to the verifier. An analogy for this is to imagine you have a magic trick where you can prove to your friend that you've solved a puzzle without revealing the solution. You show them the puzzle, cover it, and then reveal it solved without ever showing how you did it. ZKPs work similarly in cryptography: they allow one party to prove to another that they know a secret without revealing the secret itself.

My Hypothesis is that using ZKPs to allow secure training of AI models on sensitive data can prevent the disclosure of raw data to AI models. Using this method with advanced encryption and mathematical aspects could have to potential to revolutionise the world of AI, especially in areas like healthcare or finance where dealing with sensitive data is very common. My ZKP will be able to encrypt data ensuring zero-knowledge properties, and a simple Linear Regression model will be able to train on that encrypted data, with no disclosure or access to the original data.

Explanations of Components and Terms

In this section of my report book, you will find explanations of each primary component of the Report Book. These Primary Components include Zero-Knowledge Proofs, Artificial Intelligence, Linear Regression Models, Encryption & Hashing and the mathematical aspects of ZKPs in AI.

Zero-Knowledge Proofs

A Zero-Knowledge Proof (ZKP) is a method of proving the validity of a statement without revealing anything other than the validity of the statement itself. It is a proof system with a prover, a verifier, and a challenge that gives users the ability to publicly share a proof of knowledge or ownership without revealing the details of it. In essence, they demonstrate knowledge of a fact without disclosing the details of that knowledge. As their name suggests, ZKPs are cryptographic protocols that do not disclose the data or secrecy to any eavesdropper during the protocol. This concept was first introduced by MIT scientists Shafi Goldwasser, Silvio Micali, and Charles Rackoff in the 1980s. ZKPs have since found applications in various fields, including cryptography, privacy, and blockchain technology (Hasan 2019).

ZKPs must satisfy three characteristics to have evidence of zero-knowledge:

- **Completeness:** If the statement is true, an honest verifier will be convinced by an honest prover.
- **Soundness:** If the statement is false, no dishonest prover can convince the honest verifier. The proof systems are truthful and do not allow cheating.
- **Zero-Knowledge:** if the statement is true, no verifier learns anything other than the fact that the statement is true.

There are several types of ZKPs, and they can be categorised based on the nature of the statement being proven. Some of the more common types include:

1. **Interactive Zero-Knowledge Proofs (iZKPs):** In an iZKP, the prover and verifier engage in a series of communication rounds to establish the validity of the statement. The prover convinces the verifier without revealing the actual information.
2. **Non-Interactive Zero-Knowledge Proofs (NIZKPs):** Unlike iZKPs, NIZKPs require only a single round of communication. The prover can provide a proof that can be verified by the verifier without any further interaction.
3. **Statistical Zero-Knowledge Proofs (SZKPs):** These proofs allow a small, negligible probability of error. The verifier is convinced with high probability that the statement is true.
4. **Perfect Zero-Knowledge Proofs (PZKPs):** In a PZKP, the verifier learns absolutely nothing about the statement being proven.

In a real cryptographic ZKP, complex mathematical algorithms and protocols are used to ensure that the proof is sound and secure. ZKPs are used in various applications such as authentication systems, digital signatures, and privacy-preserving protocols in blockchain technology (such as Zcash's zk-SNARKs). They provide a way to establish trust and verify information without revealing sensitive data, making them a powerful tool for enhancing security and privacy in digital interactions.

ZKPs can be used in authentication systems where a user can prove their identity without revealing sensitive information such as passwords or biometric data. This is particularly useful in scenarios where privacy is paramount, such as healthcare, finance, and voting systems. They enable selective disclosure of information. This means that a party can prove they possess certain data or meet specific criteria without revealing the actual data itself. This is valuable for complying with regulations (such as GDPR) while still allowing for data analysis and verification (Hasan 2019).

ZKPs play a crucial role in blockchain technology to ensure transaction privacy and data confidentiality. For example, cryptocurrencies like Zcash and Monero use zero-knowledge proofs (zk-SNARKs and Ring Signatures, respectively) to provide anonymous

and private transactions while still maintaining a public ledger. In decentralized and trustless systems like decentralized finance (DeFi), ZKPs enable users to verify transactions and smart contract executions without relying on a central authority. This promotes transparency and reduces the need for blind trust in intermediaries.

Zk-SNARKs (zero-knowledge Succinct Non-Interactive Arguments of Knowledge) are ZKPs that is a way to prove some computational facts about the data without disclosure of the data. They are the cryptographic instrument that underlies Zcash and Hawk, both building ZKP blockchains. These SNARKS are used in the case of Zcash to verify transactions, and they are also used in the case of Hawk to verify smart contracts.

Overall, ZKPs are a powerful cryptographic tool with broad implications for enhancing digital security and privacy. Their ability to prove knowledge without revealing sensitive information has revolutionized authentication, blockchain technology, data sharing, and cybersecurity. ZKPs provide a secure and transparent way to verify information, making them a key component in building trust and privacy in digital interactions across various domains (Hasan 2019).

Artificial Intelligence

Artificial Intelligence (AI) refers to the development of computer systems or machines that perform tasks that would typically require human intelligence. It creates intelligent systems that can perceive, learn, reason, and make decisions similar to how humans do. It involves developing algorithms and computer programs that learn from and make predictions or decisions based on data. These algorithms can be trained on large datasets, enabling them to recognise patterns and make predictions or decisions that are accurate and consistent.

At its core, AI is about enabling machines to mimic certain aspects of human intelligence, such as:

- Perception: AI systems perceive and interpret information from their environment. This includes tasks like image recognition, speech recognition, natural language processing, and understanding sensor inputs.

- Learning: AI learns from data and experiences to improve their performance. They can identify patterns, extract insights, and adapt their behaviour based on feedback.
- Natural Language Processing: AI understands and generates human language. This includes tasks like language translation, chatbots, sentiment analysis, and speech synthesis.
- Robotics and Automation: AI can be applied to robotics, enabling machines to perform physical tasks autonomously or assist humans in various domains. This includes industrial automation, autonomous vehicles, and robot assistants.

There are several types of AI, including rule-based systems, machine learning, and deep learning. Rule-based systems rely on a set of predefined rules to make decisions or predictions. Machine Learning algorithms use statistical models to learn from data and improve their performance over time. Deep learning is a subset of machine learning that uses artificial neural networks to learn from data and make decisions or predictions.

Machine learning is a branch of computer science that broadly aims to enable computers to learn without being directly programmed. It has origins in the artificial intelligence movement of the 1950s and emphasises practical objectives and applications, particularly prediction and optimisation. Computers learn in machine learning by improving their performance at tasks through experience (Bi, et al. 2019).

AI is used in a wide range of applications, from virtual assistants like Siri and Alexa to self-driving cars, medical diagnosis, fraud detection, and more. As AI continues to advance, it has the potential to revolutionise many aspects of our lives and transform industries across the board.

Linear Regression Models

Linear regression is a statistical method for modeling the relationship between a dependent variable and one or more independent variables, assuming a linear relationship. It is typically represented as a straight line, $y = mx + c$, where m is the slope and c is the intercept, it predicts y based on changes in x.

In simple linear regression, one independent variable is used, while multiple linear regression includes several. The method is valued for its interpretability, simplicity, and clear assumptions like linearity, constant variance, and independence. It highlights feature importance through coefficients, aids prediction, and is computationally efficient.

However, it assumes a linear relationship, which may not suit all data. It is sensitive to outliers and multicollinearity. When these limitations arise, advanced techniques like polynomial regression or machine learning models may be more appropriate.

In summary, linear regression is a foundational yet versatile tool in data analysis and predictive modeling, though its assumptions and limitations should be carefully considered in complex scenarios (Tranmer 2008).

Encryption & Hashing

Encryption is the process of converting normal data or plaintext to something incomprehensible or cipher-text by applying mathematical transformations or formulae. These mathematical transformations or formulae used for encryption processes are called algorithms (Bhanot and Rahul 2015). Only authorized parties who have the correct key can decrypt the ciphertext back into plaintext.

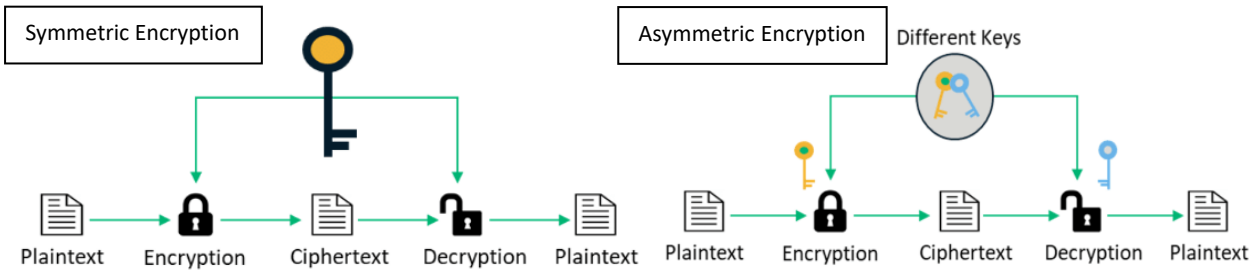
There are two primary types of encryption:

1. **Symmetric Encryption:** The same key is used for both encryption and decryption. It's fast and suitable for large amounts of data, but the key distribution process can be challenging since both parties need access to the same key. Common symmetric encryption algorithms include AES (Advanced Encryption Standard) and DES (Data Encryption Standard).

2. **Asymmetric Encryption (Public-Key Encryption):** Uses two keys: a public key (for encryption) and a private key (for decryption). The public key is openly shared, while the private key is kept secret. This method is slower but very secure and eliminates the need for both parties to have the same key. Examples include RSA (Rivest-Shamir-Adleman) and ECC (Elliptic Curve Cryptography)

Encryption algorithms take plaintext and process it with a cryptographic key to produce ciphertext. The security of encrypted data relies on the strength of the algorithm and the secrecy of the key.

Encryption ensures confidentiality, meaning that only authorized individuals or systems can access the protected information. It's widely used in various applications, like securing online transactions, protecting emails, and safeguarding sensitive data stored on devices.



Affine Order-Preserving Encryption (OPE) is a form of encryption that maintains the order of the plaintext data even after encryption. This means that if $P_1 < P_2$ for two plaintexts (P_1) and (P_2), their corresponding ciphertexts (C_1) and (C_2) will also satisfy $C_1 < C_2$. Affine OPE is a specific type of OPE where the encryption function is an affine transformation (Alexandra, Nathan and Adam 2011):

$$C = aP + b$$

Where:

- (a) is the scale factor (multiplicative constant),
- (b) is the offset (additive constant),

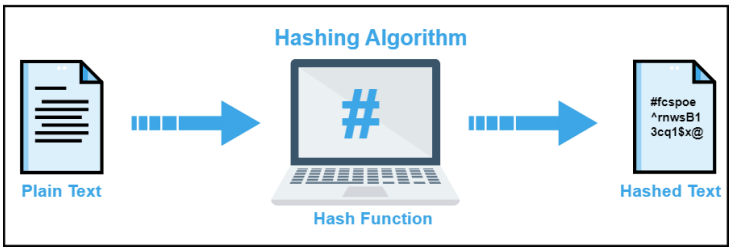
Affine transformations ensure a linear mapping while preserving the relative order of plaintext values. This property is particularly useful for scenarios where operations like sorting or range queries on encrypted data are required.

Many AI models, rely on relative ordering for meaningful predictions or transformations. Affine OPE ensures the model can operate effectively without access to raw data. Incorporating HMAC-based cryptographic commitments further ensures data integrity and prevents tampering, complementing the Affine OPE's order-preserving feature.

Affine OPE has applications in various fields where data confidentiality and usability must coexist:

1. Database Security: Allows encrypted data in databases to support operations like sorting or filtering without decryption.
2. Cloud Computing: Enables secure processing of data in outsourced environments, such as machine learning on encrypted datasets.
3. IoT and Embedded Systems: Used where lightweight encryption schemes are needed due to hardware constraints.

Hashing is a cryptographic process that takes an input (or message) and returns a fixed-size string of characters, which is typically a digest that represents the data. Unlike encryption, hashing is a one-way process: it cannot be reversed, meaning you cannot retrieve the original data from the hash. Hashing techniques have also evolved from simple randomization approaches to advanced adaptive methods considering locality, structure, label information, and data security, for effective hashing (Chi and Xingquan 2017).



The characteristics of Hash Functions include:

- Deterministic: The same input will always produce the same output.
- Fixed Output Size: Regardless of the size of the input, the output hash will always have a fixed length (e.g., SHA-256 always produces a 256-bit hash).
- Fast Computation: Hash functions are designed to quickly compute the hash value of an input.
- Pre-image Resistance: It's infeasible to determine the original input given the hash.
- Collision Resistance: It's extremely difficult to find two different inputs that produce the same hash.
- Avalanche Effect: A small change in the input results in a significantly different hash.

Some of the popular hashing algorithms include:

- MD5 (Message Digest Algorithm 5): Produces a 128-bit hash, but it's no longer considered secure due to vulnerabilities.
- SHA-1 (Secure Hash Algorithm 1): Produces a 160-bit hash but is also deprecated due to security weaknesses.
- SHA-256 and SHA-3: Both provide higher security and are widely used today in blockchain and other cryptographic applications.

Hashing serves several purposes in cryptography and computer science:

1. **Data Integrity:** Hashing verifies that data hasn't been altered. By comparing hash values, you can check if data has been tampered with or corrupted.
2. **Password Storage:** Instead of storing plaintext passwords, systems store hashed passwords. When a user logs in, the system hashes the entered password and compares it to the stored hash.
3. **Digital Signatures and Certificates:** Hashing is used to validate the authenticity and integrity of digital documents.

In cryptographic systems, both encryption and hashing play essential roles. For example, in this project encryption allows to secure sensitive data during model training, while hashing can ensure that the data remains unchanged throughout processing.

HMAC, or Hash-based Message Authentication Code, is a method that combines a hash function with a secret key to provide both data integrity and authenticity. In simpler terms, it's a way to verify that a message has not been tampered with and that it's coming from a trusted source (Krawczyk, Bellare and Ran 1997).

Here is how HMAC works in three simple steps:

1. **Message + Secret Key:** The sender combines the original message with a secret key, known only to the sender and the receiver.
2. **Hashing**: The combined message and key are processed through a hash function (like SHA-256).
3. **MAC Generation:** This process produces a unique output called the Message Authentication Code (MAC).

The MAC is sent along with the original message. The receiver can then perform the same process with their copy of the secret key to check if the MAC matches. If it does, it means:

- The message has not been tampered with (integrity).
- The message is from someone who knows the secret key (authenticity).

HMACs are widely used in secure communications (e.g., HTTPS, API requests) because they're quick and help ensure data hasn't been altered by an attacker.

The Mathematical Aspects of ZKPs in AI

When using ZKPs in linear regression AI, we are typically concerned with proving certain computations or properties without revealing the underlying data or intermediate values.

After some initial research, I discovered that it can be extremely difficult to implement zero-knowledge into AI for multiple reasons. Considering my knowledge on these proofs, that makes it a lot harder to grasp the concept and create a fully working ZKP that completely prevents disclosure of data. Below I've demonstrated the mathematics behind my original ZKP function from an earlier iteration in my project.

Example :

```
# Define the secret value and a related public value
secret_value = 5.67
public_value = secret_value * secret_value
```

- Let *secret_value = 5.67* be the secret value that has to be proven. This can be any integer or float (decimal)
- The related public value is calculated as $public\_value = secret\_value^2$

```
# Define symbols for the zero-knowledge proof
x = symbols('x')
```

- Here, *symbols('x')* creates a symbolic variable x that is used in the equation

```
# Define the equation for the zero-knowledge proof
eq = Eq(x * x, public_value)
```

- This equation represents:

$$x^2 = public\_value$$

```
# Solve the equation to find possible secret values
possible_secret_values = solve(eq, x)
```

- This solves the equation $x^2 = public\_value$ for *x*, giving us possible values of *x* (which correspond to the possible secret values)

```
# Verify the zero-knowledge proof
verified = False
for val in possible_secret_values:
    if val == secret_value:
        verified = True
        break
```

- Here, it checks if any of the possible values obtained from solving the equation match the actual secret value. If a match is found, the verification is successful.
- Overall the code is essentially solving the equation $x^2 = public\_value$ to find possible values of *x*, and then checking if any of these values match the secret value.

Linear Regression in Maths:

Let's say we have a set of data points (x_i,y_i) where x_i represents the independent variable (such as years of experience) and y_i represents the dependent variable (such as salary).

Hypothetical data points:

$$\text{Years of Experience } (x_i): [1, 2, 3, 4, 5]$$

$$\text{Salary } (y_i): [30000, 35000, 40000, 45000, 50000]$$

The goal of linear regression is to find the best-fit line that represents the relationship between *x* and *y*. We assume a linear relationship of the form:

$$y = mx + c$$

Linear regression calculates the values of *m* and *c* that minimize the sum of squared differences between the actual *y* values and the predicted *y* values on the line. Using simple mathematical formulas, the slope *m* and y-intercept *c* can be calculated as:

$$m = \frac{n(\sum xy) - (\sum x)(\sum y)}{n(\sum x^2) - (\sum x)^2}$$

$$c = \frac{\sum y - m(\sum x)}{n}$$

where *n* is the number of data points, $\sum xy$ is the sum of the products of *x* and *y*, $\sum x$ is the sum of *x* values, $\sum y$ is the sum of *y* values, and $\sum x^2$ is the sum of squared *x* values.

Calculate necessary sums:

- $\sum x = 1 + 2 + 3 + 4 + 5 = 15$
- $\sum y = 30000 + 35000 + 40000 + 45000 + 50000 = 200000$

- $\sum xy = (1 \times 30000) + (2 \times 35000) + (3 \times 40000) + (4 \times 45000) + (5 \times 50000)$
- $\sum x^2 = (1^2) + (2^2) + (3^2) + (4^2) + (5^2) = 55$
- $n = 5$ (number of data points)

Calculate the slope (*m*) and y-intercept (*y*):

$$m = \frac{(5 \times 965000) - (15 \times 200000)}{(5 \times 55) - (15^2)} = 5000$$

$$c = \frac{200000 - (5000 \times 15)}{5} = 25000$$

Use Linear regression to predict:

Now that we have the values of $m = 5000$ and $c = 25000$, we can predict the salary for someone with e.g. 6 years of experience:

$$y = mx + c = (5000 \times 6) + 25000 = 30000 + 25000 = 55000$$

Therefore, according to this linear regression model, a person with 6 years of experience is predicted to have a salary of €55,000.

Concluding from this, the overall aim of my project is to see if both of these can be integrated together into an algorithm using zero-knowledge that can prevent data disclosure to a linear regression model. Whether I will or won't be able to do this, I believe that in the future this could be a big step in the world of AI, and the prevention of data disclosure to AI models should really be considered especially in areas where sensitive data is commonly used in AI models.

My Code / Experimental Methods

In this section, you will find a breakdown of the algorithms used in my code to create the AI model with ZKP algorithm integration, and mathematic aspects of integrating Zero-Knowledge protocols into a Linear Regression Model.

I decided to create my project using the principles of User-centred design (UCD). This is a design approach that focuses on creating products, services, and experiences that are centred around the needs and desires of the people who will use them. To achieve this, UCD relies on a set of principles that guide the design process. These principles include:

- Empathy: Understanding the needs, goals, and motivations of the users and putting oneself in their shoes.
- Collaboration: Involving users in the design process and collaborating with them to co-create solutions.
- Iteration: Prototyping and testing ideas quickly and repeatedly to refine and improve the design.
- Inclusivity: Designing for the diverse needs and abilities of all potential users.
- Accessibility: Creating products and services that are accessible and usable by people with disabilities.
- User testing: Gathering feedback from users and using it to inform and improve the design.

I decided to apply UCD using the Agile Framework. Agile methodology is a project management approach that enables flexibility and empowers practitioners to engage in rapid iteration. It is based on the Agile Manifesto (Fowler 2001), a set of values and principles for software development that prioritize the needs of the customer, the ability to respond to change, and the importance of delivering working software regularly. Agile methodologies, such as Scrum and Lean, are designed to help teams deliver high-quality products in a fast-paced and dynamic environment by breaking down complex projects into smaller, more manageable chunks and continually reassessing and adjusting the project plan as needed. Agile teams are typically self-organizing and cross-functional and rely on regular communication and collaboration to achieve their goals.

While agile is best used in a team setting, it can be adapted to use by individual developers. I managed my Agile system using Kanban (Fig.1)

Fig.1

Kanban is a project management method that originated in Japan in the 1950s and is based on the principles of lean manufacturing. It is designed to help teams visualize and optimize their workflows by breaking down tasks into smaller, more manageable chunks and tracking their progress through a series of stages. Kanban uses a visual board to represent the distinct stages of a project, with cards representing individual tasks and columns representing the various stages of the workflow. The goal of Kanban is to increase efficiency and transparency by making it easier for team members to see where work is in the process, identify bottlenecks and inefficiencies, and adjust as needed. Kanban is often used in conjunction with other agile methodologies, such as Scrum, to help teams manage their work more effectively. (Corona 2013). Please scan here for my full Kanban journey.

Iteration 1:

Planning Phase

I started my project by creating a mind map (Fig. 2) of all the potential ideas that I could pursue through the lens of ZKPs. I looked towards a more topical issue that could enable the use of zero-knowledge to ensure privacy.

Fig.2

I settled on the problem of AI being trained on sensitive data and the creation of an algorithm implementing zero-knowledge to encrypt the data and ensure there is no disclosure of the raw data to the AI model (Fig 3.).

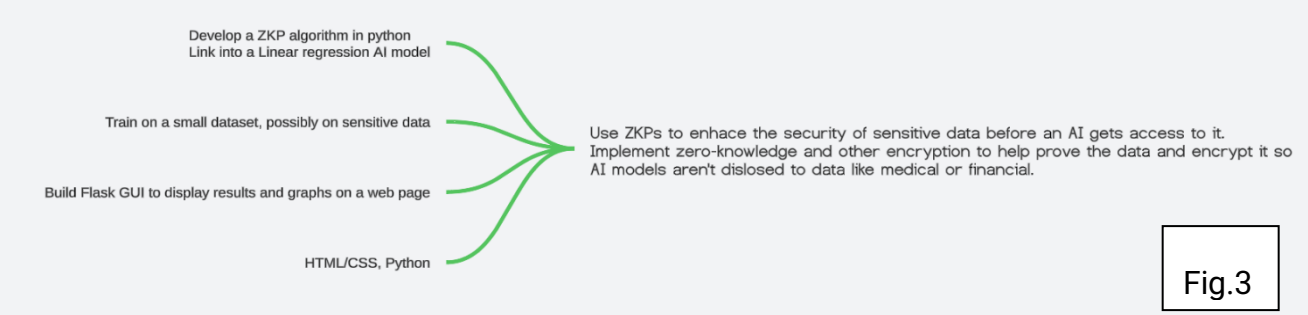


Fig.3

Design Phase

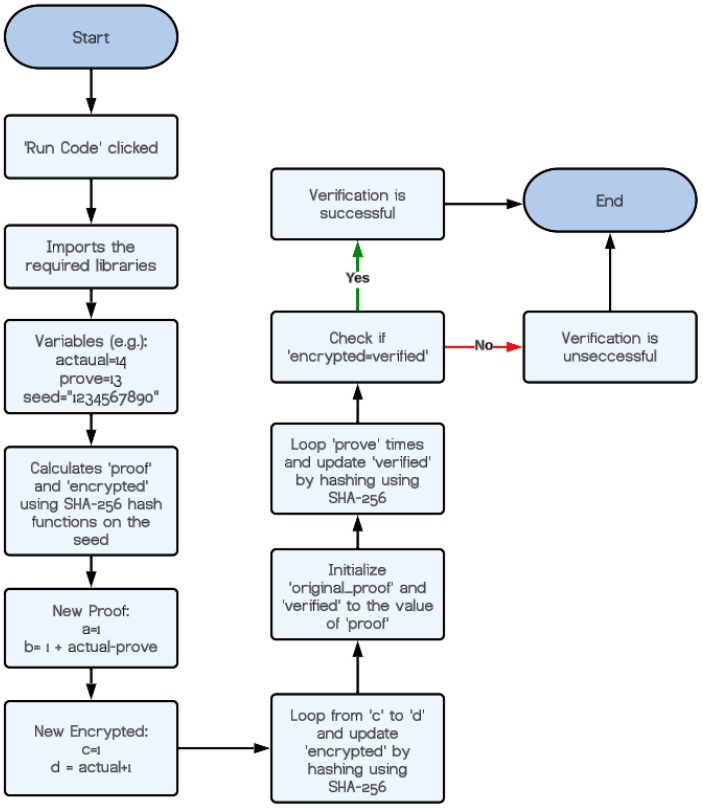


Fig.4

The basic algorithm seen above (Fig.4) for my first iteration was a simple algorithm that encrypts a piece of data (e.g. age) utilising zero-knowledge to ensure nothing is disclosed during the encryption. I found that mapping out the data flow before I coded it, helped me to plan for any tricky coding elements such as where I would need to put a conditional statement or a function.

Development Phase – Model Development Steps

In this section of the Iteration 1 Development Phase, you will find the steps it took to develop a basic ZKP algorithm to encrypt some sort of input data. The following steps are important in this development process.

1. Imports:

```
1  import hashlib;
2  import passlib.hash;
3  import sys;
```

Fig.5

These lines (1-3) import the necessary modules (hashlib, pass.hashlib and sys) for cryptographic hashing and system-related operations (Fig. 5)

2. Initialization:

```
5  # Verifier does not know the age
6  age_actual = 14
7  age_to_prove = 13
```

Fig.6

Here, *actual_age* represents the actual age, and *age_to_prove* represents the age that needs to be proven. E.g. you are 14 years old, but you want to prove you are at least 13 years or older. (Fig.6)

3. Seed and Proof initialization:

```
9  # This is seed is a random value generated by a Third Party, to ensure security
10 seed = "12345678910111213141516171819 20"
11
12 # This is a digital signature proof by Trusted Entity
13 proof = hashlib.sha256(seed.encode()).hexdigest()
14 print("Proof is " + proof)
15
16 # Proof and Encrypted have different values for distinct purposes
17 encrypted_age = hashlib.sha256(seed.encode()).hexdigest()
18 print("Encrypted Age (initialized with seed) is " + encrypted_age)
```

Fig.7

The *seed* is a random value used for cryptographic operations. *proof* is initialized by hashing the *seed* using SHA-256. SHA-256 (Secure Hash Algorithm 256-bit) is a cryptographic hash function that generates a fixed-size 256-bit (32-byte) hash value, providing a highly secure and unique representation of input data. The values of *proof* and *encrypted_age* are printed out. (Fig. 7)

4. Updating proof:

```
21 # If b < a, we don't loop, so we don't hash
22 a = 1
23 b = 1 + age_actual - age_to_prove
24 for i in range(a, b):
25     proof = hashlib.sha256(proof.encode()).hexdigest()
```

Fig.8

Here, *proof* is updated by hashing it multiple times based on the difference between *actual_age* and *age_to_prove*, in this case, is 1. This value is also printed out. (Fig.8)

5. Updating Encrypted Age:

```
# Prover hashes age_actual times his age to secure it
c = 1
d = age_actual + 1
for i in range(c, d):
    encrypted_age = hashlib.sha256(encrypted_age.encode()).hexdigest()
print("New Encrypted Age is " + encrypted_age)
```

Fig.9

Here, *encrypted_age* is updated by hashing it multiple times based on *actual_age*, in this case is 14. (Fig. 9)

6. Verifying Age:

```
# Verified Age will be Hashed age_to_prove times
original_proof = proof
verified_age = proof
for i in range(0, age_to_prove):
    verified_age = hashlib.sha256(verified_age.encode()).hexdigest()
```

Fig.10

The code verifies the age by repeatedly hashing *verified_age* using SHA-256 *age_to_prove* times. (Fig. 10)

```
# We check if Encrypted Age (By Prover) and Verified_Age (By Verifier)
if (encrypted_age == verified_age):
    print("You have proven your age!")
else:
    print("You have NOT proven your age!")
```

Fig.11

Finally, it checks is *encrypted_age* matches *verified_age* and print out the result based on if it's true or not. All of the other values are also printed out. (Fig. 11)

Testing Phase

```
Proof is ac0f6c5ad9f795ea5b958fedb929eec2201bbb7af8b65d393d7efe51275ae254
Encrypted Age (initialized with seed) is ac0f6c5ad9f795ea5b958fedb929eec2201bbb7af8b65d393d7efe51275ae254


New Proof is 7b66e8cfe268191daf348c59138a36c977a01d4187c10de47decd4dd7694f935
New Encrypted Age is 8461e3a101d583335acbda82b9e0cd24b051cde7f8f3c79cb196b8b0a075f1dc


Verified Age is 8461e3a101d583335acbda82b9e0cd24b051cde7f8f3c79cb196b8b0a075f1dc
Actual Age:             14
Age to prove:           13
....
Proof:          7b66e8cfe268191daf348c59138a36c977a01d4187c10de47decd4dd7694f935
Encrypted Age:  8461e3a101d583335acbda82b9e0cd24b051cde7f8f3c79cb196b8b0a075f1dc
Verified Age:   8461e3a101d583335acbda82b9e0cd24b051cde7f8f3c79cb196b8b0a075f1dc


You have proven your age!
```

Fig.12

Above is what the output of the ZKP code is (Fig. 12). You can see that if the encrypted age matches the verified age then the ZKP evaluates successful. I got parts of this code from GitHub, but it wasn't entirely working so I had to modify parts of it.

After trying it with different values, it overall worked fairly well. I had to start thinking about incorporating an algorithm similar to this but with a proper dataset and link it to a linear regression AI model. This ZKP algorithm encrypts the data using SHA-256 hashing, and with a linear regression model I wasn't sure how to make that work. That was one of the big factors I had to think about for the next iteration taking that into consideration.

Overall, this iteration was mainly about getting myself comfortable and explore a few different ways to use ZKPs in Python. Once I was getting good results from the code, I was ready to move on and start developing the algorithm further.

Iteration 2:

Planning Phase

My plan for iteration 2 was to start developing the real algorithm and try implement it into a simple linear regression model as well. My main focus was to get the architecture correct before I start experimenting with adding it into a Flask GUI or using different datasets. I aimed to just focus on the algorithm and linear regression because I knew this would be the most difficult and time consuming part of the development.

Design Phase

After some research, I realised that it can be extremely difficult to implement ZKPs into the encryption of data fir AI models on a simple level, for various reason that impacted my design choices including the complexity of ZKP protocols, data preprocessing, and security elements. Taking these factors into account, I decided to take a different approach to this (Fig. 13).

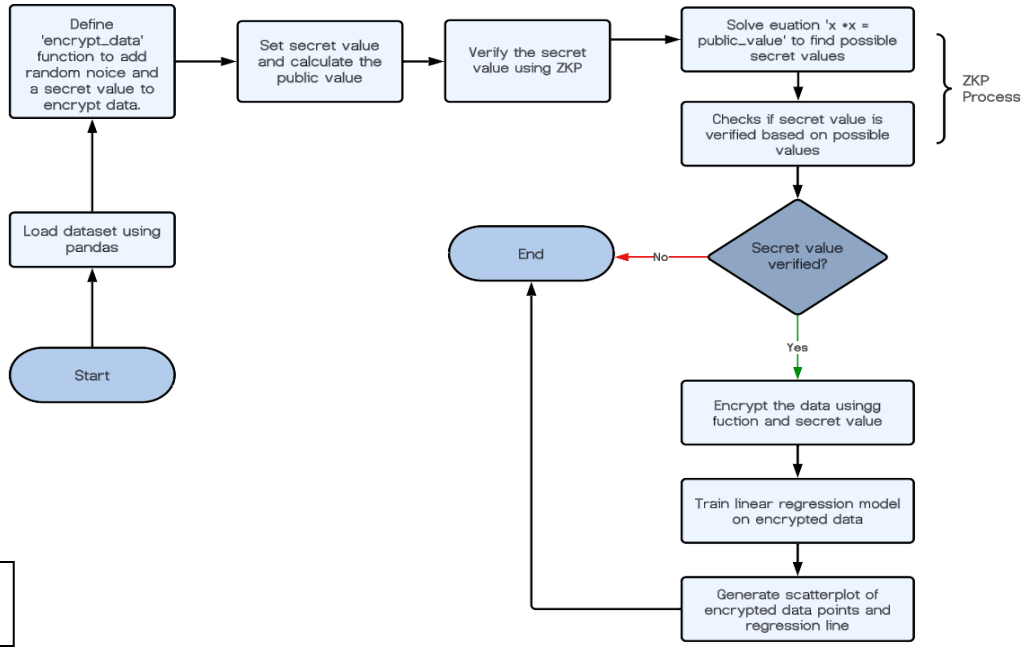


Fig. 13

Development Phase

The development phase during this iteration was particularly important. Following the flow chart above, I was able to create a working encryption algorithm along with a simple linear regression model.

1. Import Libraries:

```
1  import numpy as np
2  import pandas as pd
3  from sklearn.linear_model import LinearRegression
4  from sympy import symbols, Eq, solve
5  import random
```

Fig. 14

Import the necessary libraries (numpy, pandas, sklearn, sympy, and random) (Fig. 14)

2. Read Data and Define Values

```
7  # Read data from CSV file using pandas
8  df = pd.read_csv('6april.csv')
9
10 # Define the secret value and a related public value
11 secret_value = 5.67
12 public_value = secret_value * secret_value
```

Fig. 15

Next, it reads data from a csv file (in this case, a csv with two columns: house size and price) into a DataFrame *df* using pandas. After, the secret value *secret_value* is defined and calculates the related *public_value* for the ZKP. I created this csv file with only a few rows of data, and it was only used for testing purposes. (Fig. 15)

3. ZKP setup

```
14 # Define symbols for the zero-knowledge proof
15 x = symbols('x')
16
17 # Define the equation for the zero-knowledge proof
18 eq = Eq(x * x, public_value)
19
20 # Solve the equation to find possible secret values
21 possible_secret_values = solve(eq, x)
```

Fig. 16

Now it defines a symbol *x* for the ZKP. It sets up the equation *eq* for the ZKP as '*x* $*x = public\_value$' ($x^2 = public\ value$). It then solves the equation to find possible secret values. (Fig. 16)

4. Verification of ZKP

```
# Verify the zero-knowledge proof
verified = False
for val in possible_secret_values:
    if val == secret_value:
        verified = True
        break
```

Fig. 17

It iterates through the *possible_secret_values* to verify is any matches the *secret_value*. Next, its sets *verified* based on whether the proof is successful or not. (Fig. 17)

5. Model Training (if verified)

```
if verified:
    # Updated encrypt_data function with random noise
    def encrypt_data(data):
        encrypted_data = []
        for x in data:
            noise = random.uniform(-0.5, 0.5)  # Generate random noise between -0.5 and 0.5
            encrypted_value = int(x[0]) + secret_value + noise  # Add noise to the encrypted value
            encrypted_data.append([encrypted_value])
        return encrypted_data

    # Decrypt the data after model training
    def decrypt_data(data):
        if isinstance(data, float):
            return data - secret_value  # Handle single value
        else:
            return [x - secret_value for x in data]  # Handle list of values
```

Fig. 18

If the proof is verified, then it can proceed to the model training. It first defines an updated *encrypt_data* function to add random noise to data for encryption. Next it defines a *decrypt_data* function to decrypt data after model training. (Fig. 18)

```
    # Mask the input data using a more secure encryption (with random noise)
    X_encrypted = encrypt_data(df[['size']].values.tolist())

    # Print the encrypted data
    print("Encrypted data:")
    print(X_encrypted)

    y = df['price']

    # Train a linear regression model using the encrypted dataset
    model = LinearRegression()
    model.fit(X_encrypted, y)

    # Print the coefficients of the trained model
    print("Trained model coefficients:", model.coef_[0])

    # Predict prices for new house sizes (example)
    new_sizes_encrypted = encrypt_data([[1800], [2200]])  # Encrypt new data
    predicted_prices_encrypted = model.predict(new_sizes_encrypted)
    predicted_prices = decrypt_data(predicted_prices_encrypted)  # Decrypt predictions
```

Fig. 19

The input data, in this case is house size (*x* or independent variable) is encrypted using the *encrypt_data* function. A simple linear regression model is trained on the encrypted dataset and the coefficients are printed out. Next, new data is encrypted to predict

prices using the trained model and then they are decrypted after they have been predicted. (Fig. 19)

6. Output

```
    # Print the encrypted predictions
    print("Predicted prices for new house sizes (encrypted):")
    print(predicted_prices_encrypted)

    # Print the decrypted prices
    print("Predicted prices for new house sizes (decrypted):\n", predicted_prices)

else:
    print("Cannot proceed with model training due to failed zero-knowledge proof.")
```

Fig. 20

The encrypted data, model coefficients, encrypted predictions and decrypted predications are printed based on the process. If the ZKP is unsuccessful, the model wont train and encrypt the data. (Fig. 20)

In the code, the ZKP is used to verify a secret value *secret_value* without revealing it directly. The ZKP involves setting up an equation where *x* represents a potential secret value, and *public_value* is a known value derived from the secret value. It then solves this equation to find possible secret values. This ZKP benefits the rest of the code by ensuring that sensitive information, represented by the *secret_value,* is securely validated before proceeding with critical operations like model training and predictions.

By verifying the *secret_value* without revealing it directly, the ZKP adds a layer of security and privacy to the process. This is particularly important in scenarios involving encrypted data or confidential computations where the actual values must remain hidden but still need to be validated for correctness. The successful verification of the ZKP (i.e., *verified* is True) guarantees that subsequent operations can be safely executed, such as encrypting and decrypting data, training machine learning models, and making predictions, with confidence in the integrity and authenticity of the secret value without exposing it unnecessarily. Thus, the ZKP enhances the overall security and trustworthiness of the code's functionality involving sensitive information handling.

```
Zero-knowledge proof verified successfully.
Encrypted data:
[[1005.76333000478], [1505.2505086964336], [2005.8451713061581], [2505.4072677012855], [3005.193362047414], [3505.887468218O7], [4005.769281783408], [4505.2866
752073105], [5006.031297609827]]
Trained model coefficients: 199.65644277551928
Predicted prices for new house sizes (encrypted):
[409352.97602818 489250.75963146]
Predicted prices for new house sizes (decrypted):
 [409347.30602817825, 489245.0896314629]
```

Fig. 21

This method incorporating a ZKP worked a lot better. The code from iteration 1 would've been difficult to incorporate into linear regression because the inputs weren't numerical, making it difficult to make predictions. As well as the ZKP verification, it adds random noise to each piece of data, increasing its overall security once it's gone through the AI model. There is also a decrypt function, so the data can be previewed as both encrypted and decrypted in the output. (Fig. 21)

Testing phase

Testing in iteration 2 was important as I had added new functions and features to the code. I experimented with the code by changing the different values involved in the ZKP and some of the other values. I used a small, simple dataset with two columns for now, and for the test data I used two values that aren't included in this csv. (Fig. 22)

	size	price
1	size	price
2	1,000	250000
3	1,500	350000
4	2,000	450000
5	2,500	550000
6	3,000	650000
7	3,500	750000
8	4,000	850000
9	4,500	950000
10	5,000	1050000
11		

Fig. 22

Everything was working quite well during this iteration and, I figured that next I should focus on some of the visual aspects and possibly using a different dataset. The regular Thonny/VS Code interface didn't help with the visual aspects either so I decided that in the next iteration I could try incorporate a simple GUI.

Iteration 3:

Planning phase

For this iteration, I needed to focus on completing a few main things. I needed to think a method to create a GUI and develop it. Although it was not the main focus of my project, I decided to add a bit of CSS to make the GUI look cleaner and more organised. Adding extra CSS or HTML wouldn't be too difficult as a few of my past projects involved a lot of CSS/HTML.

I also planned to train the AI on a proper dataset. The program works well on a small dataset so using a larger dataset shouldn't cause too many problems. Additionally, If there is any parts of the code that need to be modified then I would complete that in this iteration.

Design phase

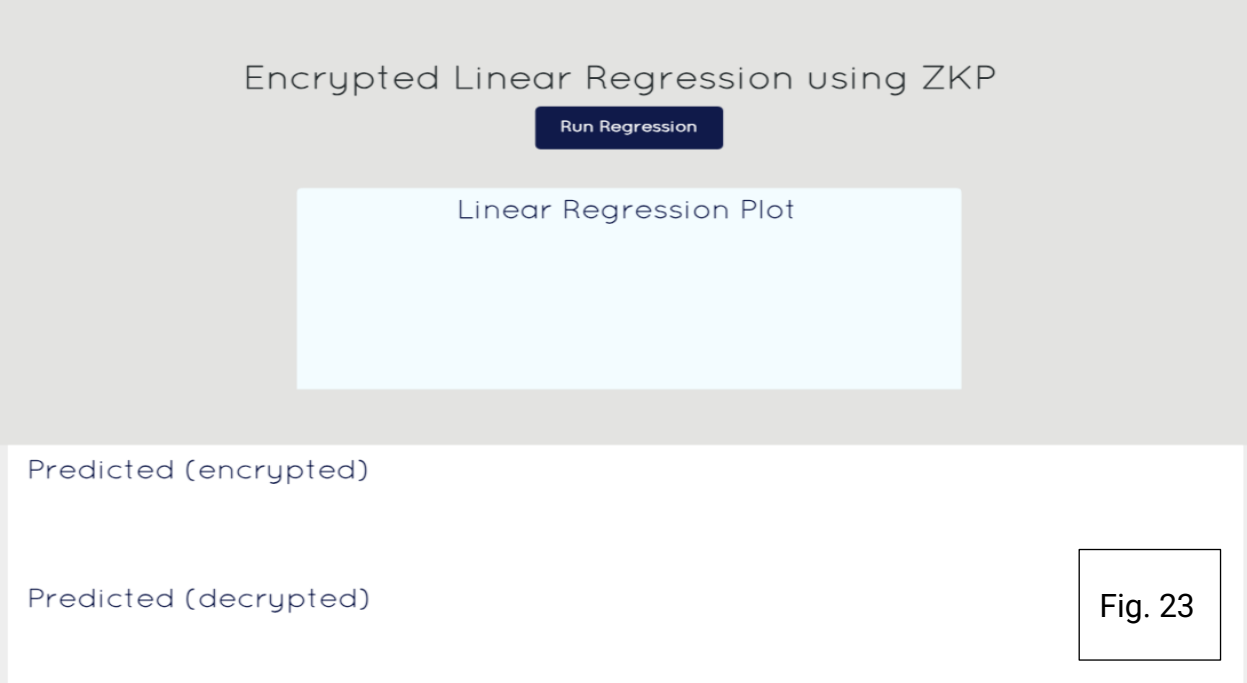


Fig. 23

I designed a simple wireframe of what I wanted the GUI to look like (Fig. 23) if I could achieve that level of customisation. I wasn't entirely sure if I would have the time or if I would be able to get something like that. If I ended up being able to get a GUI that looks similar, I would additionally change things like the colour and font styles. Again, this was not a main component of my project.

Development phase

In one of my recent projects, I initially went with Tkinter for a GUI. Tkinter is a Python library used for creating graphical user interfaces (GUIs) with widgets like buttons, labels, and entry fields, making it easier to build interactive applications. Tkinter doesn't allow for a deep level of customization, and it's also caused problems for me in the past. After some thinking, I disregarded the Tkinter GUI approach due to time constraints and focused on creating a simple Flask based GUI interface (Fig. 24). I had initially thought about using a Django based GUI, but my teacher, concerned for the time necessary to become competent in Django, suggested that I try an online course in beginner web applications using Flask. I used this tutorial to help (Loeber 2021).

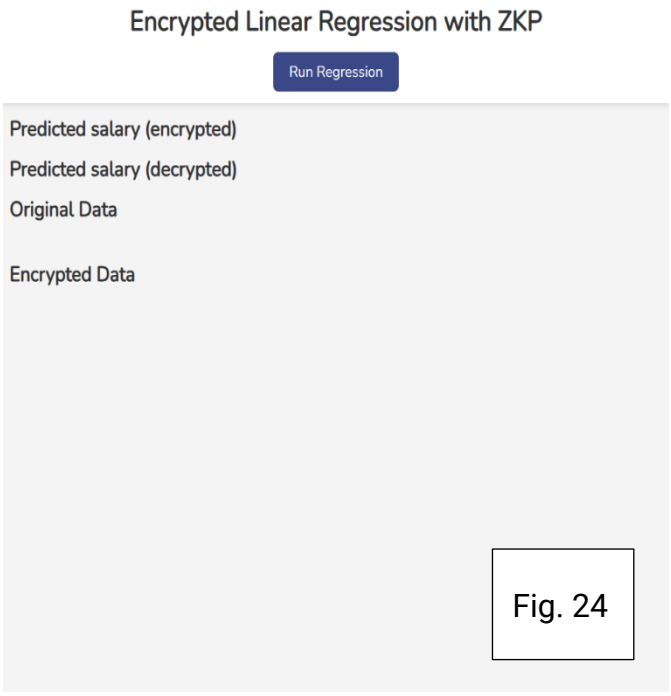


Fig. 24

Using Flask allowed me to use my HTML/CSS/JavaScript coding experience once I was able to deploy the Flask module.

Flask also allowed me to incorporate the matplotlib scatterplot (Fig. 25).

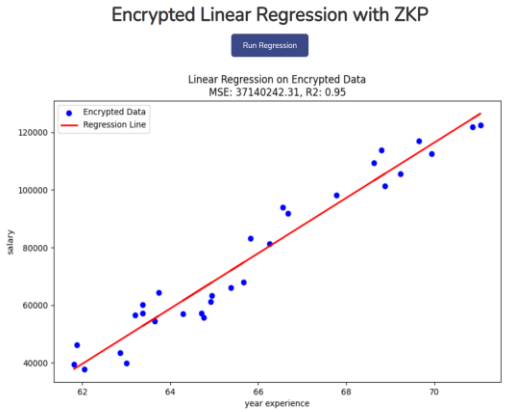


Fig. 25

I was also able to display some of the predicted outputs both encrypted and decrypted on the same page. (Fig. 26)

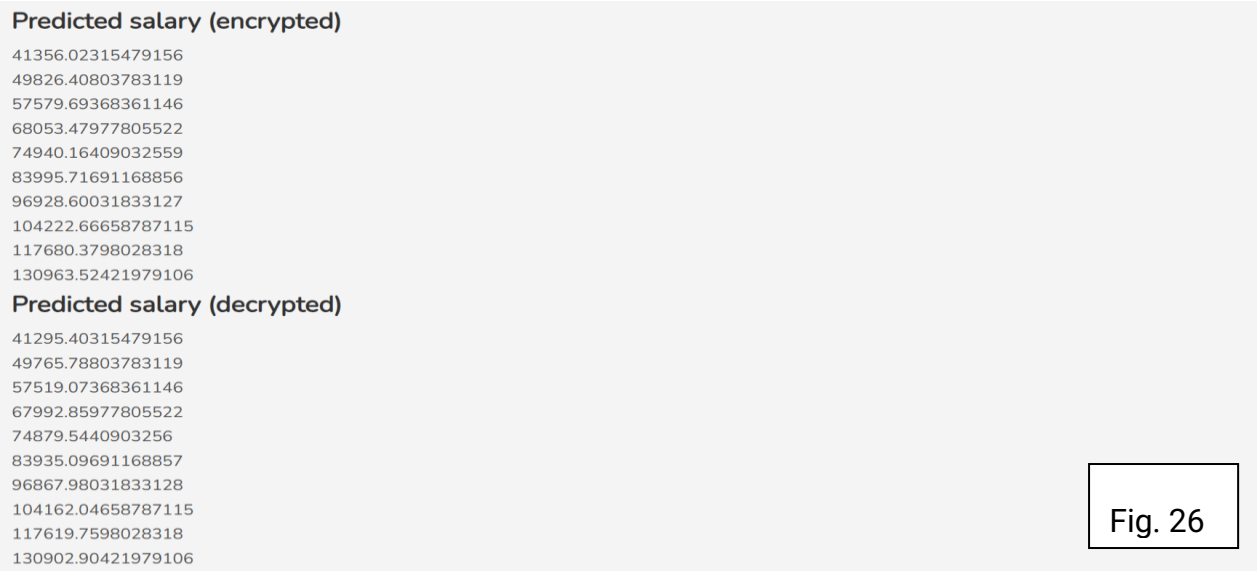


Fig. 26

This involved some simple HTML where it creates a list `<*ul*>` of the encrypted predicted salaries. It uses a loop (line 26, 32) to go through each encrypted salary (line 27, 33) and display it on the screen. (Fig. 27)

```
23  <div class="data">
24    <h2>Predicted salary (encrypted)</h2>
25    <ul>
26        {% for item in predicted_sal_e %}
27        <p>{{ item }}</p>
28        {% endfor %}
29    </ul>
30    <h2>Predicted salary (decrypted)</h2>
31    <ul>
32        {% for item in predicted_sal %}
33        <p>{{ item }}</p>
34        {% endfor %}
35    </ul>
```

Fig. 27

Once the Flask GUI was built and working with the Python code, I focused on the final few bits with the AI and ZKP. I needed to find and train the AI on a more realistic dataset and adjust the code in any way that might fit more appropriately with the dataset.

My next step was to find a dataset to train the AI on. I searched on Kaggle to find a simple dataset that I could use for my project. I settled on a simple Salary Dataset (Abhishek 2023) which can be used for linear regression, as it only has two columns, which

is perfect for implementing it into my code. I wanted to choose a dataset that contained some sort of data that would be considered 'sensitive' in real world scenarios.

Sensitive data refers to any information that, if disclosed or accessed without authorization, could result in harm, discrimination, or privacy violations for individuals or organizations. This can include personal information such as financial data, health records, biometric data, and confidential business information.

In the context of AI, sensitive data is crucial because AI systems often rely on vast amounts of data to learn and make decisions. Handling sensitive data in AI involves ensuring proper security measures, ethical considerations, and legal compliance to protect individuals' privacy and prevent misuse or unauthorized access. I chose this salary dataset because it is an example of financial data, which in real life would be considered sensitive.

Luckily enough, it was a small enough dataset, so I didn't have to do any data cleaning as there weren't any NaN or null columns. Although, the training data (Fig. 28) was the only csv that came with the file, so I had to create my own test data (Fig. 29) for the model to make predictions based on.

	YearsExpe	Salary
0	1.2	39344
1	1.4	46206
2	1.6	37732
3	2.1	43526
4	2.3	39892
5	3	56643
6	3.1	60151
7	3.3	54446
8	3.3	64446
9	3.8	57190
10	4	63219

Fig. 28

	YearsExpe	Salary
0	1.9	43567
1	2.5	52787
2	3.5	63993
3	4.5	73992
4	5.5	83123
5	6.5	94783
6	7.5	105793
7	8.5	116783
8	9.5	128923
9	10.5	139872

Fig. 29

I updated the code to read in these two csv viles using pandas (Fig. 30).

```
17  # Load datasets
18  df = pd.read_csv('Salary_dataset.csv')
19  df1 = pd.read_csv('Salary_dataset_test.csv')
```

Fig. 30

There were a few more places I had to update so that it would encrypt the correct columns of data. Below shows that once the proof is verified, the training data is encrypted and the model is able to train (Fig. 31).

```
75      if verified:
76          # Encrypt the training data
77          X_encrypted = encrypt_data(df[['YearsExperience']].values.tolist(), secret_value)
78          y = df['Salary']
79
80          # Train the linear regression model
81          model = LinearRegression()
82          model.fit(X_encrypted, y)
83
84          # Encrypt the test data
85          test_data = df1[['YearsExperience']].values.tolist()
86          new_years_encrypted = encrypt_data1(test_data, secret_value)
```

Fig. 31

Another area I had to update was where once the *YearsExperience* column in the test data is encrypted, it will be able to predict the salaries and produce the MSE and R-squared. (Fig. 32)

```
88          if new_years_encrypted:
89              # Predict salaries for new data
90              predicted_salary_encrypted = model.predict(new_years_encrypted)
91              predicted_salary = decrypt_data(predicted_salary_encrypted)  # Decrypt predictions
92
93              # Calculate Mean Squared Error (MSE) and R-squared (R2 score)
94              mse = mean_squared_error(y, model.predict(X_encrypted))
95              r2 = model.score(X_encrypted, y)
96
97              # Generate the plot
98              plt.figure(figsize=(10, 6))
99              # Update plot title with MSE and R2 score
100             plt.title(f'Linear Regression on Encrypted Data\nMSE: {mse:.2f}, R2: {r2:.2f}')
101             plt.scatter([x[0] for x in X_encrypted], y, color='blue', label='Encrypted Data')
102             plt.plot([x[0] for x in X_encrypted], model.predict(X_encrypted), color='red', linewidth=2, label='Regression Line')
103             plt.xlabel('year experience')
104             plt.ylabel('salary')
105             plt.legend()
```

Fig. 32

Once the dataset was in, I ran it a few times to make sure that it was working properly. Below is an image of a scatterplot produced by the model. (Fig. 33)

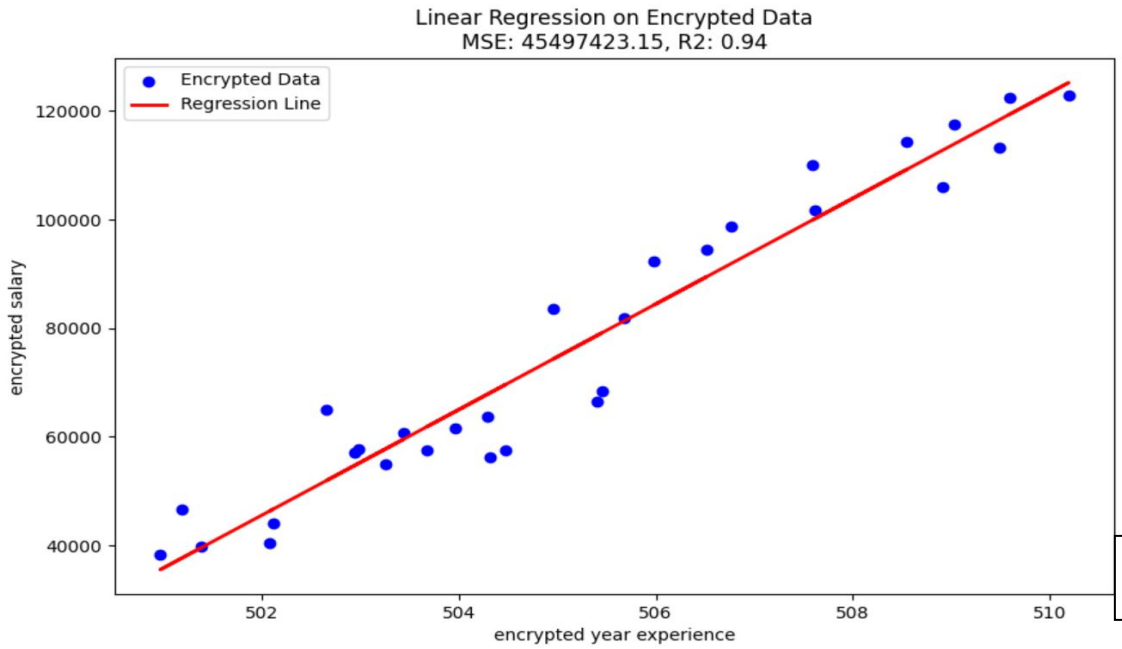


Fig. 33

Additionally, I wanted there to be some representation of the encrypted values on the HTML. I created a simple table to display the predicted salary encrypted, predicted salary decreypted (test data) as well as the years of experience encrypted and decrypted (test data). (Fig. 34)

```
<div class="data">
    <h2>Data Overview</h2>
    <table>
      <tr>
        <th>Predicted Salary (Encrypted)</th>
        <th>Predicted Salary (Decrypted)</th>
        <th>Test Data (Decrypted)</th>
        <th>Test Data (Encrypted)</th>
      </tr>
      {% for index in range(predicted_sal_e|length) %}
      <tr>
        <td>{{ predicted_sal_e[index] }}</td>
        <td>{{ predicted_sal[index] }}</td>
        <td>{{ original_data[index] }}</td>
        <td>{{ encrypted_data[index] }}</td>
      </tr>
      {% endfor %}
    </table>
  </div>
```

Fig. 34

This is what the code displays on the GUI. (Fig.35)

Data Overview

Predicted Salary (Encrypted)	Predicted Salary (Decrypted)	Test Data Years (Decrypted)	Test Data Years (Encrypted)
46548.59354917705	45662.18354917705	[1.2]	[887.768396324284]
48595.53968119994	47709.129681199935	[1.4]	[887.340206218319]
64195.52811775729	63309.11811775729	[1.6]	[887.0935422024731]
68800.1172472816	67913.70724728159	[2.1]	[888.6983351367086]
84292.38214113005	83405.97214113004	[2.3]	[888.0810631519189]
93847.8737158645	92961.46371586449	[3.0]	[889.694555649598]
96275.69433411583	95389.28433411583	[3.1]	[889.6459010386034]
108600.01767147705	107713.60767147705	[3.3]	[889.6630602840149]
118624.83867330477	117738.42867330476	[3.3]	[889.34191131
131229.43064370193	130343.02064370192	[3.8]	[889.84901566

Fig. 35

Testing phase

Once I had the real dataset complete, I completed a thorough testing of the model. There were a few small problems, for example if something in the python didn't match up with the HTML, it wouldn't show up on the screen. I had to be careful when adding in features like this to go and check to make sure that they are both connected from each file.

The new salary predictions dataset worked quite well, additionally, the test dataset also worked very well. Although the dataset it very small, I'm glad that it worked in the linear regression and salary values are able to be predicted based on the input (Fig. 36).

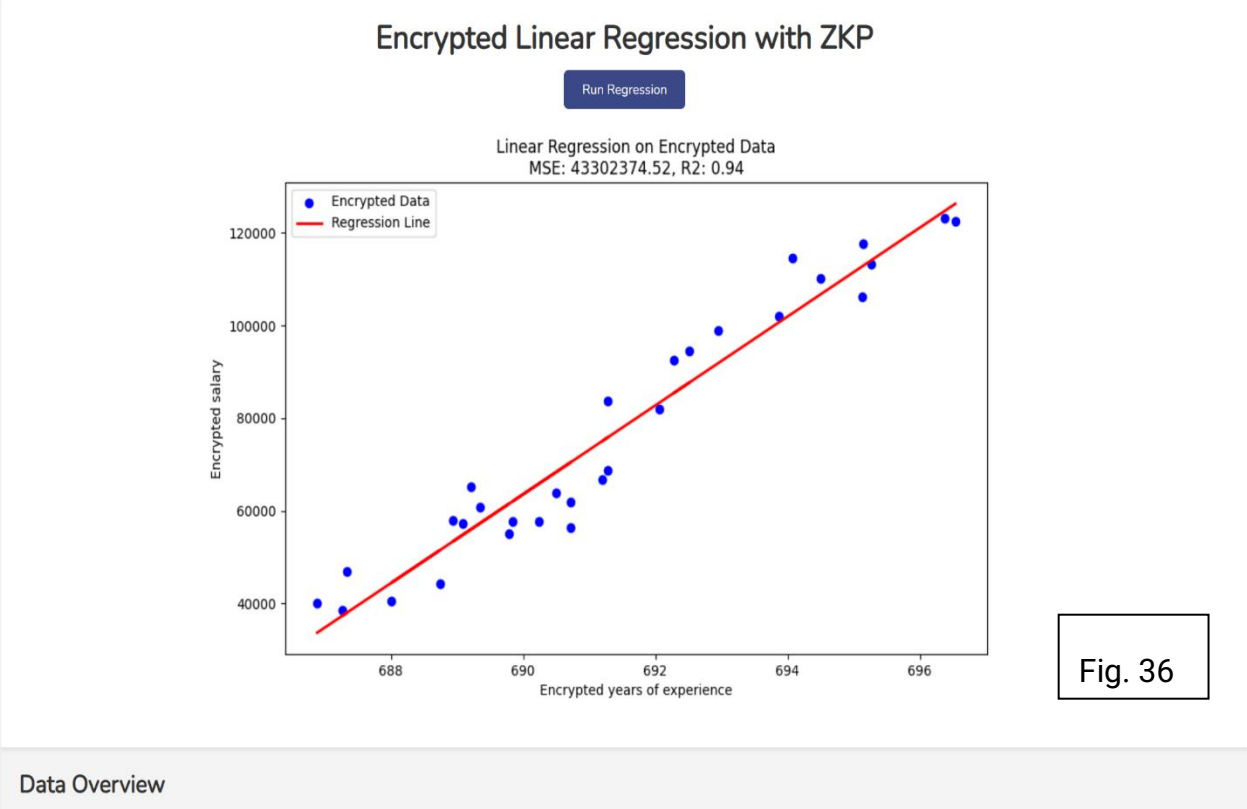


Fig. 36

Iteration 4:

Planning Phase

From reflecting on previous iterations of the code, there were a few elements that I figured could be improved to enhance the performance of the model and zero-knowledge security. Below is a few points on what I decided to implement during the last development iteration of my project:

- Improve the algorithm by integrating a stronger encryption process, possibly looking into more commonly used encryption methods.
- Enhance ZKP aspect by utilizing hash functions and commitment schemes to prove knowledge of the data without revealing the underlying values.
- Display encrypted model predictions and decrypted model predictions to visualise if the relationships between datapoints differ when the data has been manipulated.
- Test various datasets on the model to show results on different types of sensitive data compatible and possibly add more model metrics to evaluate performance.
- Update Flask GUI to display this updated information produced by the model.

Design Phase

Taking these updated features to include in this version of the algorithm, I reevaluated my design approach (Fig. 37):

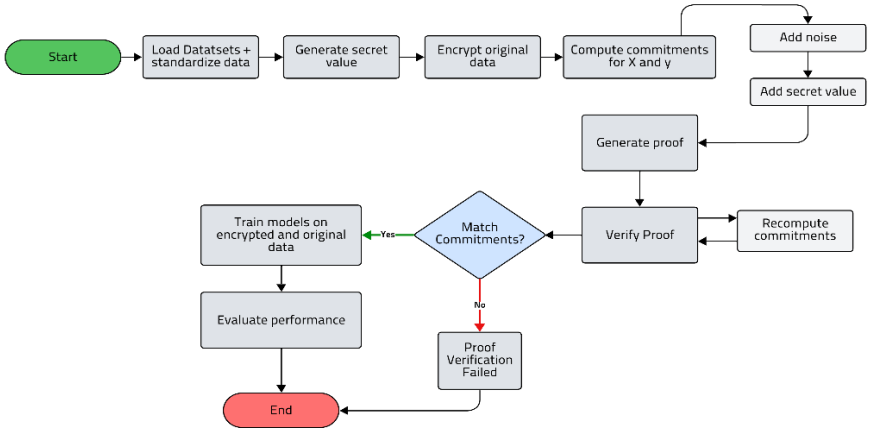


Fig. 37

Development Phase

Below is each of the steps and processes which take place in the code, including the data manipulation, ZKP commitments and model training.

1. Imports, Parameters and Dataset Loading:

```
1  import pandas as pd
2  import numpy as np
3  import random
4  import hashlib
5  import matplotlib.pyplot as plt
6  from sklearn.linear_model import LinearRegression
7  from sklearn.metrics import mean_squared_error
8  import logging
9  import hmac
10 import secrets
11
12 # Initialize logging with a more formatted output
13 logging.basicConfig(level=logging.INFO, format='%(message)s')
14
15 # Parameters for affine encryption-like order-preserving encryption
16 scale_factor = 2.4  # Adjust as needed to scale values
17 offset = secrets.randbelow(1000)  # Randomly generated offset for encryption
18
19 # Load datasets
20 df = pd.read_csv('Salary_dataset_train.csv')
21 df1 = pd.read_csv('Salary_dataset_test.csv')
```

The necessary libraries are imported for data manipulation, encryption, modeling, and plotting (lines 1-10). Logging is configured for structured and formatted output (line 13). Multiple parameters for encryption including *scale_factor* and *offset* (lines 16-17). The training and test datasets are then loaded (lines 20-21)

2. Data Manipulation Functions:

```
23 def standardize(data):
24     # Standardize the data.
25     return (data - np.mean(data)) / np.std(data)
```

This standardizes the data to have a mean of 0 and a standard deviation of 1 (lines 23-25).

```
27 def encrypt_data(data):
28     # Encrypt by scaling and adding an offset; preserves order.
29     encrypted_data = (data * scale_factor) + offset
30     return encrypted_data
```

The encryption applies an affine-style encryption to scale and offset the data using the relevant parameters (lines 27-30).

```
32 def decrypt_data(encrypted_data):
33     # Decrypt by reversing the scaling and offset.
34     decrypted_data = (encrypted_data - offset) / scale_factor
35     return decrypted_data
```

The decrypt function reverses the scaling and offset to decrypt data (lines 32-35).

```
37 def generate_commitment(data):
38     # Generate a cryptographic commitment to the data.
39     data_bytes = data.tobytes()
40     commitment = hashlib.sha256(data_bytes).hexdigest()
41     return commitment
```

The generate commitment function generates a cryptographic hash of the data for integrity verification using SHA-256.

3. HMAC Commitment and Proof Generation:

```
43  # Generate a secret key for HMAC commitments
44  secret_key = secrets.token_bytes(32)  # 256-bit key for strong security
```

A secret key is generated for HMAC-based commitments using *secrets.token_bytes* (line 44).

```
46  def generate_hmac_commitment(data, secret_key, salt):
47      # Generate a cryptographic HMAC commitment to the data with a secret key and salt
48      data_bytes = data.tobytes()
49      salt_bytes = salt.encode('utf-8')  # Convert salt to bytes
50      hmac_object = hmac.new(secret_key, data_bytes + salt_bytes, hashlib.sha256)  # Combine data and salt
51      commitment = hmac_object.hexdigest()
52      return commitment
```

The function *generate_hmac_commitment(data, secret_key, salt)* creates HMAC-based cryptographic commitments using SHA-256. The function combines the data and a salt for added security (lines 46-52).

```
54  def generate_proof(X, y, X_encrypted, y_encrypted, secret_key, salt):
55      # Generate a ZKP with HMAC commitments to enhance verification security.
56      commitment_X = generate_hmac_commitment(X, secret_key, salt)
57      commitment_y = generate_hmac_commitment(y, secret_key, salt)
58
59      # Log the HMAC-based commitments
60      logging.info(f'\nCommitments Generated (HMAC):')
61      logging.info(f'  - HMAC Commitment for X: {commitment_X}')
62      logging.info(f'  - HMAC Commitment for y: {commitment_y}')
63
64      proof = {
65          'commitment_X': commitment_X,
66          'commitment_y': commitment_y
67      }
68
69      logging.info(f'\nProof Details:')
70      logging.info(f'\n  - Encrypted X Sample: {X_encrypted[:3]}...')
71      logging.info(f'\n  - Encrypted y Sample: {y_encrypted[:3]}...')
72
73      return proof
```

The function *generate_proof* is defined to create a proof using HMAC commitments for both X and y. It logs the commitments and encrypted samples for verification (line 54-73).

```
75  def verify_proof(proof, X, y, secret_key, salt):
76      # Verify the HMAC-based proof to check data integrity.
77      commitment_X_check = generate_hmac_commitment(X, secret_key, salt)
78      commitment_y_check = generate_hmac_commitment(y, secret_key, salt)
79
80      # Validate commitments
81      is_valid = (commitment_X_check == proof['commitment_X']) and \
82                 (commitment_y_check == proof['commitment_y'])
83
84      if is_valid:
85          logging.info('\n[Success] HMAC-based proof verification succeeded.')
86      else:
87          logging.error('\n[Error] HMAC-based proof verification failed.')
88
89      return is_valid
```

The *verify_proof* function validates the HMAC commitments. It then logs success or failure based on the match between the generated and provided commitments (lines75-89).

4. Data Preparation:

```
95  # Original data
96  X_train = standardize(df[['YearsExperience']].values)
97  y_train = standardize(df[['Salary']].values)
98
99  # Salt value for HMAC
100 salt = secrets.token_hex(16)  # Generate a 16-byte salt
```

The input data are standardized with the *standardize* function. A random salt (additional random value) is generated for the HMAC commitments (lines 95-100).

5. Encryption and Proof Verification:

```
105 # Encrypt the original data
106 X_encrypted = encrypt_data(X_train)
107 y_encrypted = encrypt_data(y_train).flatten()
108
109 # Generate and verify proof using the HMAC-based commitment
110 proof = generate_proof(X_train, y_train, X_encrypted, y_encrypted, secret_key, salt)
```

The input data is encrypted using the *encrypt* function (lines 106-107). A proof is generated based on the encrypted data (line 110). Then there is a conditional to check if the proof is verified. If the verification is successful, the training will begin.

6. Model Training:

```
112 # Verify the proof before training the model
113 if verify_proof(proof, X_train, y_train, secret_key, salt):
114     logging.info('\n[Model Training]')
115     # Proceed with model training
116
117     # Train the linear regression model on the encrypted data
118     model_encrypted = LinearRegression()
119     model_encrypted.fit(X_encrypted, y_encrypted)
120     logging.info(f'  - Trained model on encrypted data.')
121
122     # Train the linear regression model on the original (un-encrypted) data
123     model_original = LinearRegression()
124     model_original.fit(X_train, y_train)
125     logging.info(f'  - Trained model on original data.')
126
127     # Encrypt the test data
128     X_test = standardize(df1[['YearsExperience']].values)
129     X_test_encrypted = encrypt_data(X_test)
130
131     # Predict salaries for the test data using the model trained on encrypted data
132     y_pred_encrypted = model_encrypted.predict(X_test_encrypted)
133     y_pred_decrypted = decrypt_data(y_pred_encrypted)
134
135     # Predict salaries for the test data using the model trained on original data
136     y_pred_original = model_original.predict(X_test)
137
138     # Evaluate model performance on the training data (encrypted)
139     mse_encrypted = mean_squared_error(y_encrypted, model_encrypted.predict(X_encrypted))
140     r2_encrypted = model_encrypted.score(X_encrypted, y_encrypted)
141
142     # Evaluate model performance on the training data (original)
143     mse_original = mean_squared_error(y_train, model_original.predict(X_train))
144     r2_original = model_original.score(X_train, y_train)
```

Once the proof is successful, two linear regression models are trained on both encrypted and original data. The test data is also encrypted to make predictions. The model's performance is evaluated using Mean Squared Error (MSE) and R-squared (R^2)

7. Metrics and Plotting:

```
177    # Calculate additional metrics for encrypted data model
178    mae_encrypted = mean_absolute_error(y_encrypted, model_encrypted.predict(X_encrypted))
179    mape_encrypted = mean_absolute_percentage_error(y_encrypted, model_encrypted.predict(X_encrypted))
180    rmse_encrypted = np.sqrt(mean_squared_error(y_encrypted, model_encrypted.predict(X_encrypted)))
181
182    # Calculate additional metrics for original data model
183    mae_original = mean_absolute_error(y_train, model_original.predict(X_train))
184    mape_original = mean_absolute_percentage_error(y_train, model_original.predict(X_train))
185    rmse_original = np.sqrt(mean_squared_error(y_train, model_original.predict(X_train)))
```

Some additional model metrics are evaluated such as Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE) for both model to get a further insight into the performance on both encrypted and decrypted training. The plots with regression lines and data points are also outputted by the model at the end (Fig. 38).

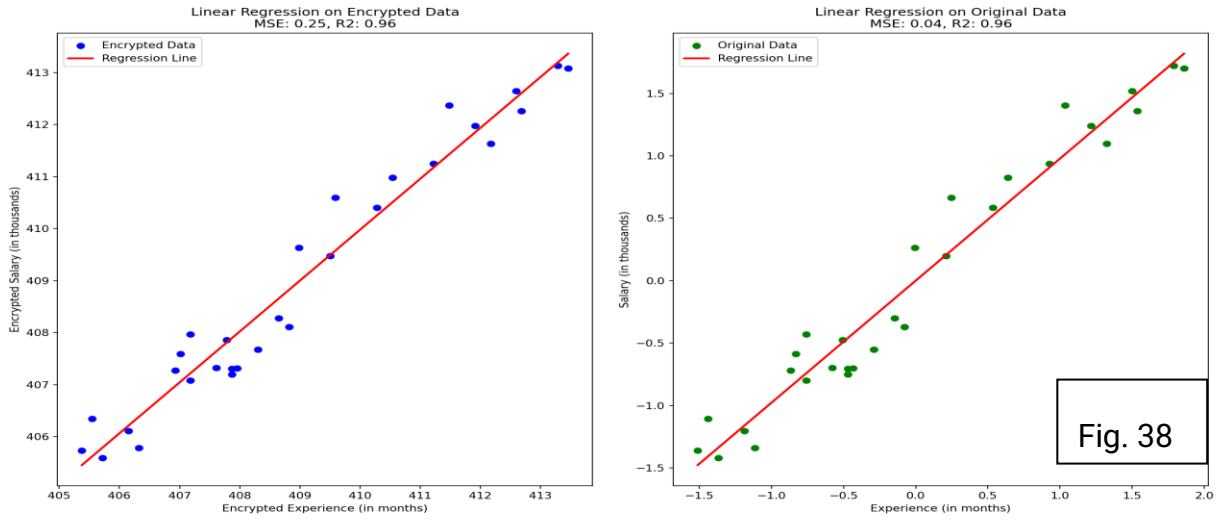


Fig. 38

From looking at these graphs produced by the model, we can see that the underlying relationships between the data whether encrypted or not, remain the same. This was an ultimate goal to achieve, and to prove that even though the raw data is unseen by the model, it can still evaluate predictions accurately while encrypted.

Once this code functioned correctly and ran smoothly, I updated the Flask GUI to display the new results and metrics evaluated by the model (Fig. 39).

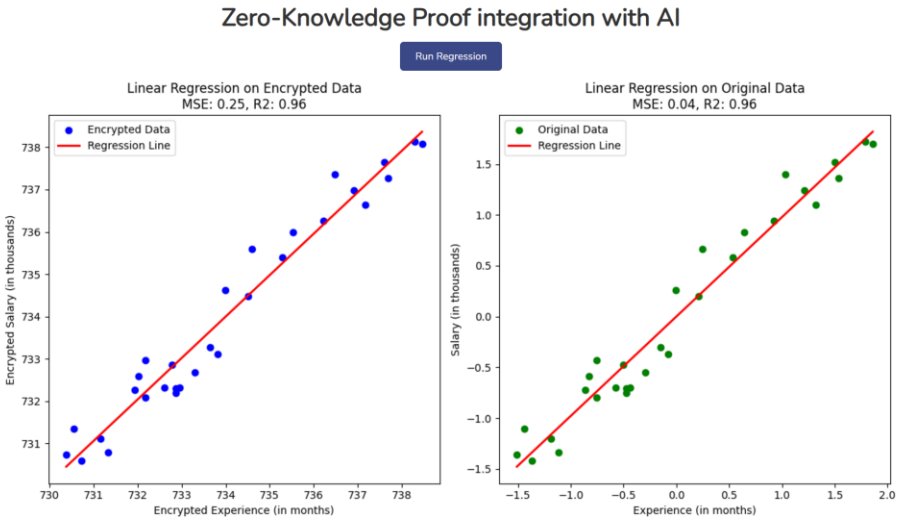


Fig. 39

I updated the Python and HTML code to display the new plots generated by the model (Fig. 39). I also added a new table to display the performance metrics evaluated by the model and split them into encrypted and decrypted columns (Fig. 40).

Model Performance Metrics

Metric	Encrypted Model	Original Model
MSE	0.247929614533398 3	0.04304333585649145
MAE	0.41352777937690243	0.17230324080703935
RMSE	0.49792531019561387	0.20746887924816929
R²	0.9569566641435072	0.9569566641435086
MAPE	0.0011755482192723095	0.26511719586767507

Fig. 40

I also updated the Data Overview plot to display the new values form the test data and predictions, both encrypted and decrypted (Fig. 41).

Data Overview

Test Data Years (Decrypted)	Predicted Salary (Decrypted)	Test Data Years (Encrypted)	Predicted Salary (Encrypted)
-1.472535882416383	-1.4404958848977856	544.4659138822007	544.5428098762453
-1.2591248849647332	-1.2317283653473985	544.9781002760847	545.0438519231662
-0.9034398892119836	-0.8837824994300547	545.8317442658912	545.8789220013679
-0.5477548934592342	-0.5358366335127112	546.6853882556978	546.7139920795695
-0.19206989770648472	-0.18789076759536746	547.5390322455045	547.5490621577711
0.16361509804626473	0.16005509832192882	548.392676235311	548.3841322359726
0.5193000937990142	0.5080009642393198	549.2463202251176	549.2192023141744
0.8749850895517637	0.8559468301566635	550.0999642149243	550.054272392376
1.2306700853045132	1.2038926960739598	550.9536082047308	550.8893424705775
1.5863550810572626	1.5518385619913033	551.8072521945375	551.7244125487791

Fig. 41

Mathematical Interpretation of the Data Encryption:

I've attempted to show an example of how the encryption process works using a suitable example. I've already explained the maths behind the ZKP process in the explanations of components and terms.

1. Data Encryption with Affine Order-Preserving Encryption (OPE)

Affine encryption uses a simple linear transformation:

$$E(x) = (x \times s) + o$$

Where:

- x: Original data.
- s: Scale factor (in this case, $s = 2.4$).
- o: Offset (randomly generated).

Decryption reverses this process:

$$D(E(x)) = \frac{E(x) - o}{s}$$

Example: Suppose $x = 5$, $s = 2.4$, and $o = 100$.

- Encrypted value: $E(5) = (5 \times 2.4) + 100 = 112$
- Decrypted value: $D(112) = \frac{112-100}{2.4} = 5$

Affine OPE ensures that the relative ordering of data is preserved.

If $x_1 < x_2$, $then\ E(x_1) < E(x_2)$.

2. HMAC-based Cryptographic Commitments

HMAC (Hash-based Message Authentication Code) is a keyed hash function. A commitment is a "fingerprint" of the data that cannot be altered without detection.

Commitment formula:

$$C = \text{HMAC}(K, data + salt)$$

Where:

- K: Secret key.
- data: Original data (in byte form).
- salt: Randomly generated value to add variability.

Example: Assume:

- Data: $x = 5$
- Salt: "$a1b2c3d4e5..$"
- Key: $K = 32$ random bytes

The process combines the data with the salt and applies a keyed hash using SHA-256.

$$Commit(x) = \text{SHA} - 256(K|5|\text{salt})$$

3. Zero-Knowledge Proofs (ZKPs)

A ZKP allows one party to prove that they know a value (e.g., x) without revealing it. The proof involves commitments:

1. Generate commitments for X and y (encrypted inputs and outputs).
2. Share the commitments.
3. Verify that commitments match the recomputed values.

The ZKP ensures data integrity and privacy.

Proof Generation:

1. Commit to X and y using HMAC:

$$Proof = \{\text{HMAC\_Commit}(X), \text{HMAC\_Commit}(y)\}$$

2. Encrypted data ($E(x), E(y)$) and proof are logged for verification.

Verification:

Verify the commitments match:

$$\text{HMAC\_Commit}_{\text{verify}}(\text{X}) = \text{Proof}[\text{X}_{\text{commit}}]$$

$$\text{HMAC\_Commit}_{\text{verify}}(\text{y}) = \text{Proof}[\text{y}_{\text{commit}}]$$

If both equations hold, the ZKP succeeds. This guarantees the encrypted data aligns with the original data.

4. Training and Prediction in Linear Regression:

Linear regression is a statistical method for modeling the relationship between a dependent variable (y) and one or more independent variables (x). The model is expressed as:

$$y = mx + c$$

Where:

- y: Predicted value (dependent variable).
- x: Input feature (independent variable).
- m: Slope of the line (weights in machine learning).
- c: Intercept (value of y when $x = 0$).

The goal of linear regression is to find m and c that minimize the error between the predicted and actual values.

I will show a sample walkthrough using these values in terms of the salary dataset:

- $X = \text{YearsExperience}$ (input feature)
- $y = \text{Salary}$: (target variable)

The training data consists of pairs of data points (X_i, y_i) where $i = 1,2,\ldots,n$ (number of data points).

Objective of Linear Regression:

Linear regression minimizes the Mean Squared Error (MSE):

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

The optimal values of m and b are calculated by differentiating the MSE function with respect to m and b, setting the derivatives to zero, and solving for m and b.

Optimal Coefficients:

The formulas for m (slope) and b (intercept) are derived as follows:

$$m = \frac{n\sum_{i=1}^{n}(X_i y_i) - \sum_{i=1}^{n}X_i \sum_{i=1}^{n} y_i}{n\sum_{i=1}^{n}X_i^2 - \left(\sum_{i=1}^{n}X_i\right)^2}$$

$$b = \frac{\sum_{i=1}^{n} y_i - m\sum_{i=1}^{n}X_i}{n}$$

Using the following training data sums:

- $n = 30$
- $\sum X_i = 76.1$
- $\sum y_i = 1669780.0$
- $\sum X_i^2 = 119.71$
- $\sum X_i y_i = 4580197.0$

We can calculate:

1. Slope (m):

$$m = \frac{30(4580197.0) - (76.1)(1669780.0}{30(119.71) - (76.1)^2}$$

$$m \approx 9449.962$$

2. Intercept (b):

$$b = \frac{1669780.0 - 9449.962(76.1)}{30}$$

$$b \approx 25792.200$$

Final Linear Regression Equation:

$$\text{Salary} = 9449.962 \cdot \text{YearsExperience} + 25792.200$$

Making Predictions:

Using the equation, predictions for any new YearsExperience can be made:

Example Predictions:

YearsExperience	Calculation	Predicted Salary
1.9	$9449.962 \cdot 1.9 + 25792.200$	≈ 43747.128
2.5	$9449.962 \cdot 2.5 + 25792.200$	≈ 49417.105
3.5	$9449.962 \cdot 3.5 + 25792.200$	≈ 58867.067
4.5	$9449.962 \cdot 4.5 + 25792.200$	≈ 68317.030

5. Metrics in Linear Regression:

1. Mean Squared Error (MSE)

Measures average squared difference between predicted and actual values:

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

2. Coefficient of Determination (R^2)

Indicates how well the regression model explains the variability of the dependent variable:

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$

Where:

- $SS_{res} = \sum(y_i - \hat{y}_i)^2$ (residual sum of squares)
- $SS_{tot} = \sum(y_i - \bar{y}_i)^2$ (total sum of squares)

Example Calculation:

Given:

- Actual values: $y = [3, -0.5, 2, 7]$,
- Predicted values: $\hat{y} = [2.5, 0.0, 2, 8]$.

1. MSE:

$$\text{MSE} = \frac{1}{4}((3 - 2.5)^2 + (-0.5 - 0)^2 + (2 - 2)^2 + (7 - 8)^2) = 0.375$$

2. R^2:

$$\bar{y} = \frac{3 - 0.5 + 2 + 7}{4} = 2.875,$$

$$SS_{tot} = (3 - 2.875)^2 + (-0.5 - 2.875)^2 + (2 - 2.875)^2 + (7 - 2.875)^2 = 37.5$$

$$SS_{res} = 0.375 \times 4 = 1.5$$

$$R^2 = 1 - \frac{1.5}{37.5} = 0.96$$

Testing Phase

Testing in iteration 4 was crucial as I made a number of changes compared to the previous iteration. Testing the code itself was important to ensure everything was working correctly and certain inputs were producing expected outputs.

I also ensured to test the hash commitment functions multiple times to ensure it was working as expected. This was important as the zero-knowledge aspect relies on this to work to ensure that the data is proven with no disclosure to the underlying values.

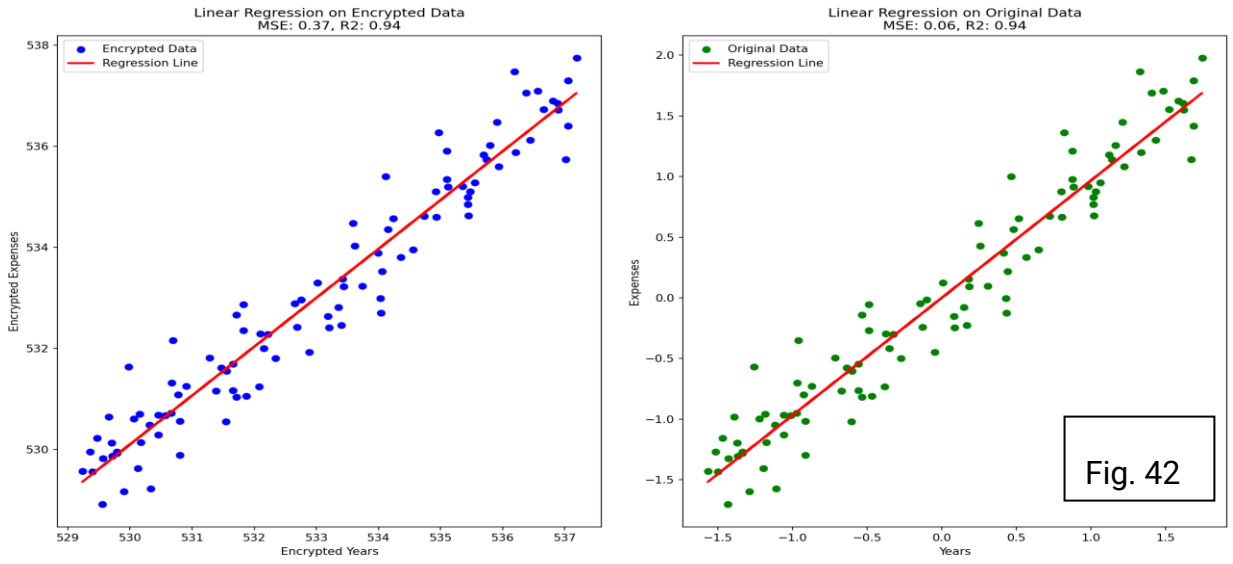
One major problem I experienced was integrating the final code into the Flask Application. This can cause problems frequently as if there is something missing or incorrect, it wouldn't run properly. In the end I was able to make a smooth transition from the code outputs displaying in the terminal onto the HTML page.

Final results from testing various aspects of the final model can be viewed on the next page. The link to the GitHub Repository with all of the code is located in the Appendices.

Results

Since I primarily trained the model on one dataset throughout the development process, I decided to trial a few other datasets to gain a wide variety of results from suing my algorithm. Below, you can see results from using my algorithm based on four other datasets consisting of ‘sensitive data’ which I generated using simple Python codes.

1. Healthcare Data:




```
[Model Evaluation Metrics - Encrypted Data]
  - MAE: 0.48
  - MAPE: 0.00%
  - RMSE: 0.61
  - MSE: 0.37
  - R2 Score: 0.94
```

```
[Model Evaluation Metrics - Original Data]
  - MAE: 0.20
  - MAPE: 1.42%
  - RMSE: 0.25
  - MSE: 0.06
  - R2 Score: 0.94
```

Both the encrypted and original models perform similarly, with a high R^2 (0.94) indicating that the model explains 94% of the variance in the data. However, the MSE for the encrypted data is slightly higher, suggesting that while encryption doesn't significantly degrade the model's ability to fit the data, there is a small loss in prediction accuracy due to the encryption process (Fig. 42).

2. Banking Data:

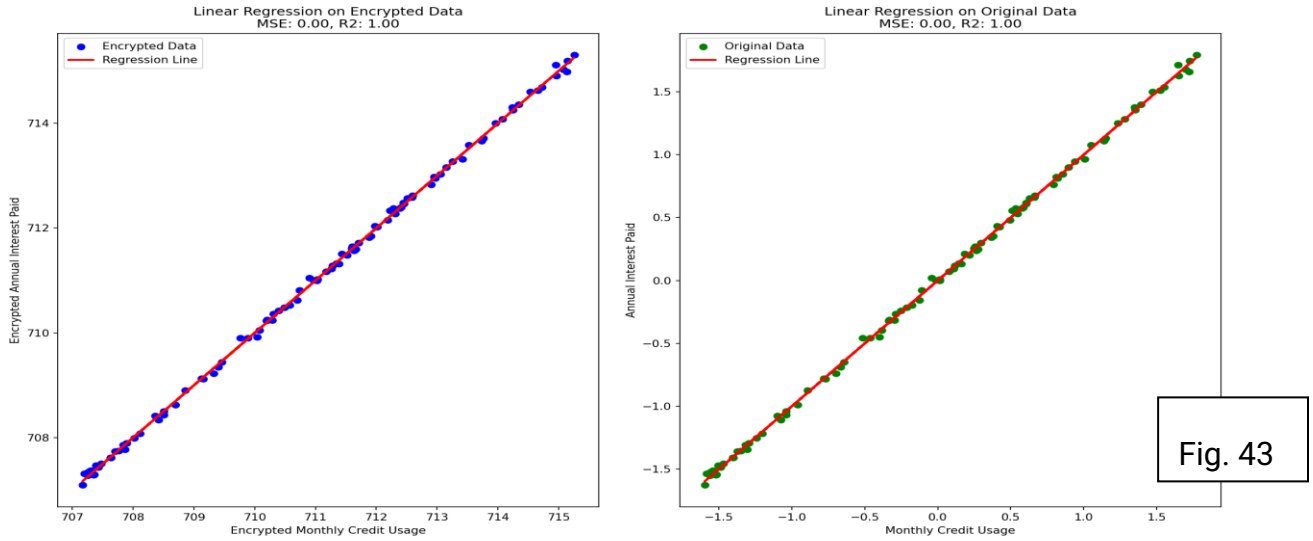


Fig. 43

```
[Model Evaluation Metrics - Encrypted Data]
  - MAE: 0.05
  - MAPE: 0.00%
  - RMSE: 0.06
  - MSE: 0.00
  - R2 Score: 1.00
```

```
[Model Evaluation Metrics - Original Data]
  - MAE: 0.02
  - MAPE: 0.67%
  - RMSE: 0.02
  - MSE: 0.00
  - R2 Score: 1.00
```

Both models achieve perfect results (MSE = 0.00, R^2 = 1.00), meaning the model perfectly predicts the outcomes for both encrypted and original data. This suggests that the encryption technique used does not introduce any loss of information or accuracy for this dataset (Fig. 43).

3. Retail Data:

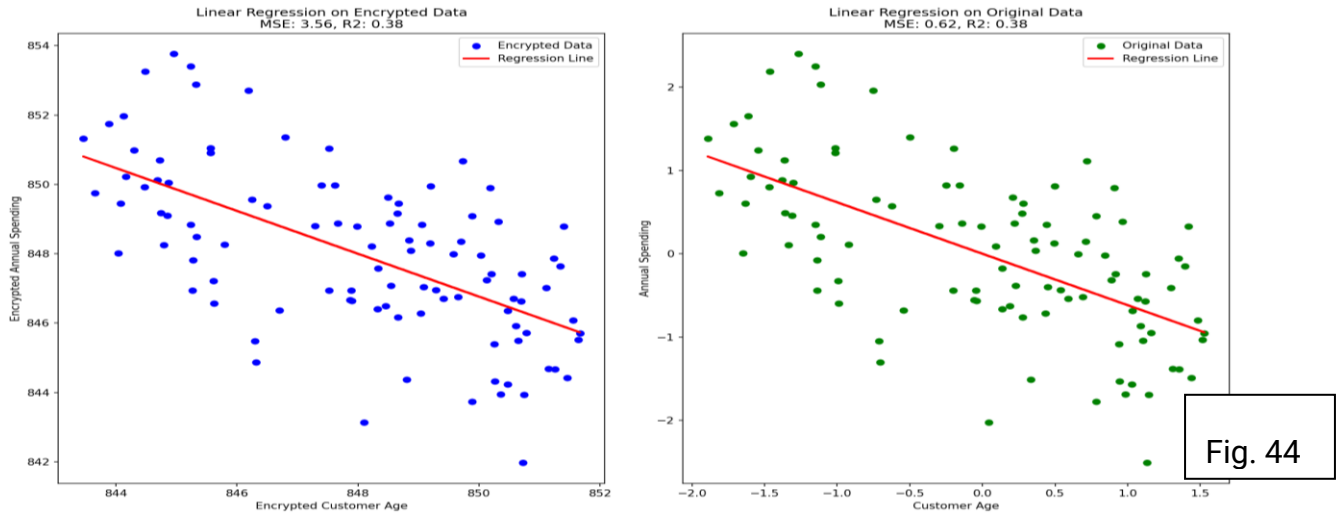


Fig. 44

```
[Model Evaluation Metrics - Encrypted Data]
  - MAE: 1.52
  - MAPE: 0.00%
  - RMSE: 1.89
  - MSE: 3.56
  - R2 Score: 0.38
```

```
[Model Evaluation Metrics - Original Data]
  - MAE: 0.63
  - MAPE: 5.10%
  - RMSE: 0.79
  - MSE: 0.62
  - R2 Score: 0.38
```

The encrypted data performs significantly worse than the original data in terms of MSE (3.56 vs. 0.62). This is due to the scaling of the data in the OPE. However, both models have the same R^2 score of 0.38, indicating that the proportion of variance explained by the model is the same, but the predictive accuracy is noticeably lower with the encrypted data (Fig. 44).

4. Housing Data:

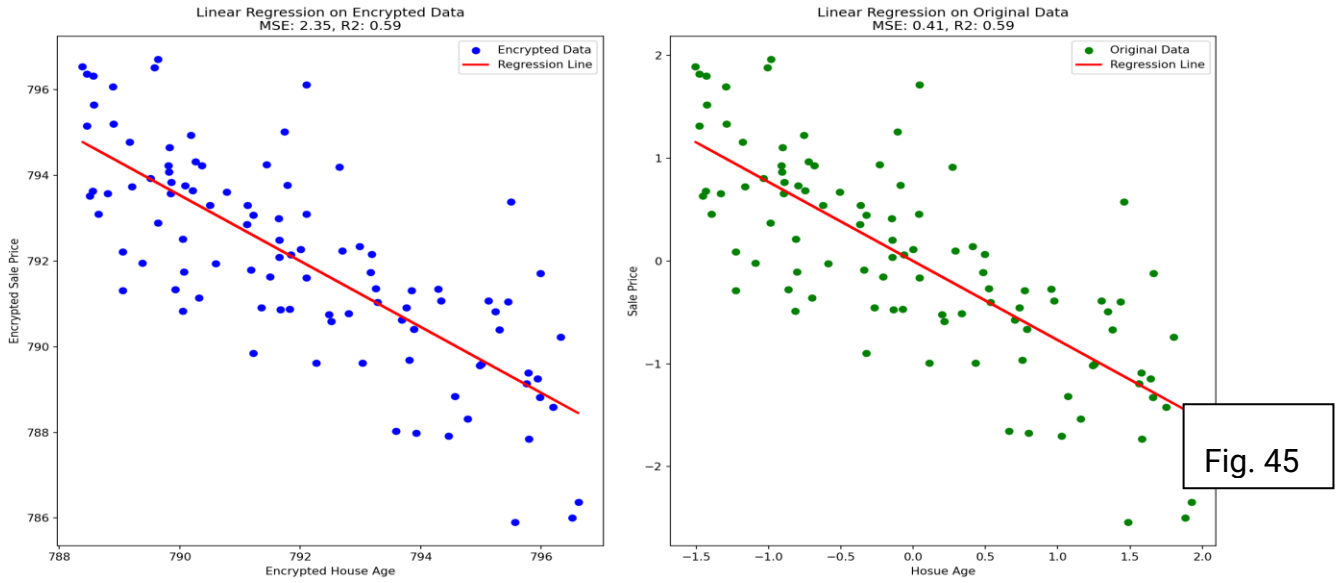


Fig. 45

```
[Model Evaluation Metrics - Encrypted Data]
  - MAE: 1.21
  - MAPE: 0.00%
  - RMSE: 1.53
  - MSE: 2.35
  - R2 Score: 0.59
```

```
[Model Evaluation Metrics - Original Data]
  - MAE: 0.50
  - MAPE: 1.81%
  - RMSE: 0.64
  - MSE: 0.41
  - R2 Score: 0.59
```

The MSE for encrypted data is higher than for the original data (2.35 vs. 0.41), but the R^2 scores are identical (0.59). This suggests that, while the encryption impacts the prediction error, the overall fit to the data remains similar (Fig. 45).

For datasets like Healthcare and Banking, encryption does not significantly degrade the model's performance. In some cases, such as the Banking data, the encryption introduces no loss of accuracy (MSE = 0, $R^2 = 1$). However, for datasets like Retail and Housing, encryption leads to a noticeable increase in MSE, although the model's ability to explain the variance in the data (R^2) is largely unaffected.

While encryption can slightly reduce prediction accuracy, it does not drastically impact the overall fit of the model, as measured by R^2. This suggests that the encryption process can be implemented without severely compromising the model's explanatory power, though attention to prediction error (MSE) is important when working with encrypted data. Below shows MSE and R^2 comparisons for each dataset (Fig. 46).

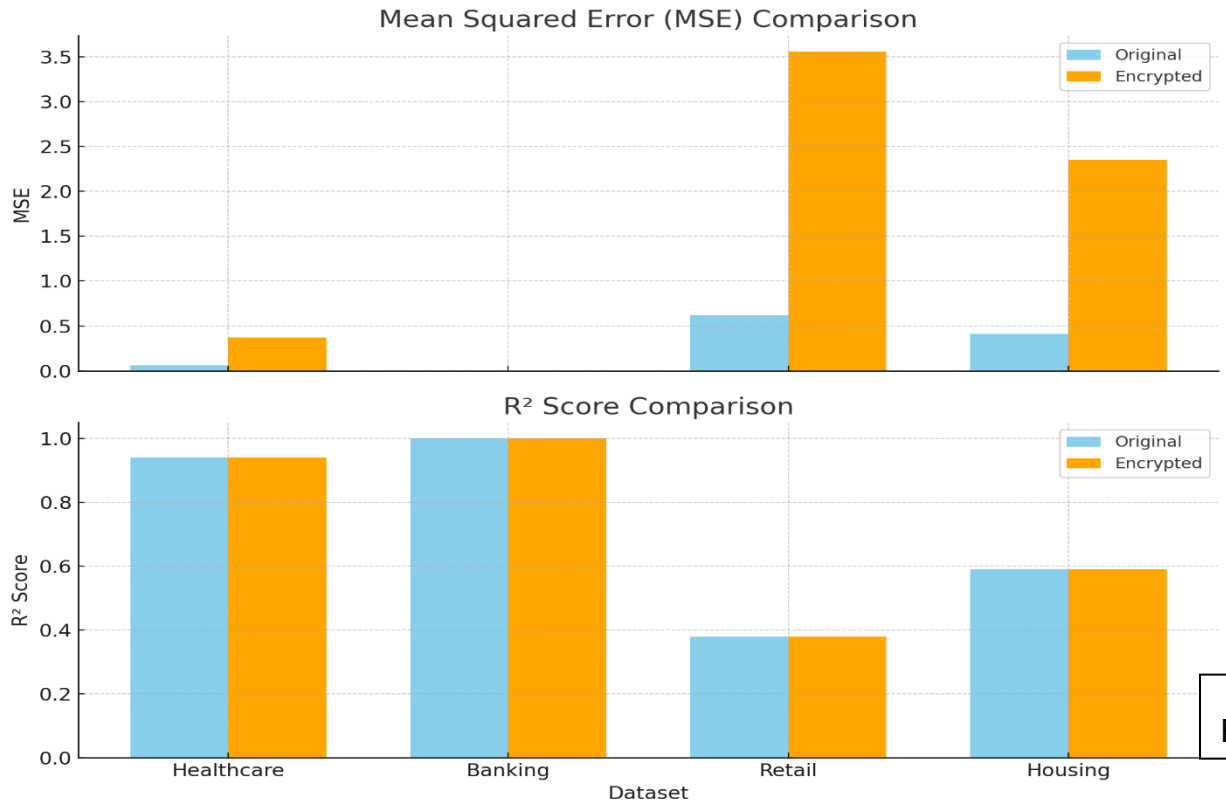


Fig. 46

Looking back to my hypothesis, my aim was to develop a ZKP that is able to prove and encrypt data, and a simple LR model would be able to train on that encrypted data, with no disclosure or access to the original data. I believe I have achieved the brief of what I have set myself to, and I believe that this project can still go a lot further.

Conclusions

There are several reasons as to why ZKPs can be extremely useful in the world of AI and why they should really be considered:

1. Privacy Preservation: ZKPs allow for computations to be performed on encrypted data without revealing the data itself. This is crucial for protecting sensitive information, such as personal data in healthcare or financial records.
2. Data Security: By using ZKPs, AI systems can access data securely without the risk of data breaches or leaks. Encrypted data can be used in computations without exposing it to potential attackers.
3. Compliance with Regulations: Many regulations, such as GDPR in the EU or HIPAA in the US, require strict data privacy measures. ZKPs can help AI systems comply with these regulations by ensuring that sensitive data remains confidential.
4. Enhanced Collaboration: ZKPs enable parties to collaborate and share data without actually sharing the underlying data itself. This can foster collaboration in research, healthcare, and other fields where data sharing is critical, but privacy concerns are paramount.
5. Increased Trust: With ZKPs, individuals and organizations can trust that their data is being used appropriately and securely. This can lead to greater acceptance and adoption of AI technologies that rely on sensitive data.
6. Improved AI Model Training: AI models often require large amounts of data for training. ZKPs can enable secure access to diverse datasets without centralizing or exposing the data, leading to more robust and accurate AI models.

These advantages highlight how ZKPs can play a pivotal role in unlocking the full potential of AI while addressing privacy and security concerns in data-driven applications.

Although, there can be some disadvantages that come along with it:

1. Complexity and Resource Intensive: Implementing ZKPs can be complex and resource-intensive, requiring specialized cryptographic knowledge and computational power. This can increase development time and costs.
2. Performance Overhead: ZKPs often introduce performance overhead due to the computational burden of generating and verifying proofs. This can impact the speed and efficiency of AI algorithms, especially in real-time applications.
3. Trust in Implementation: The security of ZKPs relies heavily on the correct implementation of cryptographic protocols. Any vulnerabilities or flaws in the implementation can compromise the entire system's security and privacy guarantees.
4. Potential for Misuse: ZKPs can also be used in malicious ways, such as masking illegal activities or facilitating illicit transactions. Balancing privacy with the need for accountability and transparency is a delicate issue.
5. Regulatory and Compliance Challenges: Compliance with data protection regulations and industry standards can be more complex in ZKP-based systems. Ensuring that ZKPs align with legal requirements without compromising privacy is an ongoing concern.
6. Compatibility Issues: Integrating ZKP technologies with existing systems or platforms may face compatibility challenges. Ensuring seamless interoperability with legacy systems or third-party services can be non-trivial.

While ZKPs offer compelling privacy and security benefits, careful consideration of these disadvantages is essential for their successful adoption and deployment in AI and other applications.

Looking at what I was able to discover from my results, shows that even though the data is manipulated significantly when encrypted, the relationship between each data point remains the same. This proves that using various encryption methods and zero-knowledge can change the way AI trains on sensitive data.

My Improvements

There are several issues that, mainly given time and research, would improve both the ZKP and AI aspects of my project.

My ZKP algorithm is not developed to the highest security standard and is only used on a very basic level. I believe that with further research and a lot more time, that I could develop a proper ZKP system that would completely prevent the disclosure of data to AI models. I was able to improve this by integrating hash commitments to prove knowledge of the data before it undergoes training in the finial iteration development.

With using the Flask GUI, which means the interface is run as a local host on my computer. This is not necessarily the best option as I cannot easily get people to beta test my project unless they run the whole program on their device, which is not efficient. Beta testing is really important when creating any GUI or front end.

To show that I can manipulate the encrypted data, I created a very simple linear regression model. Obviously, with more time I would be able to develop the LR model to be a lot more accurate. Additionally, I would love to see ZKPs being used in other areas of machine learning, particularly it would be amazing in the area of deep learning and GANs so that the model would learn and improve over time.

I was able to make a significant improvement already between the time I had to develop my project after the regional SciFest. I was able to improve my algorithm and collect a wide variety of results using different datasets and improve the overall accuracy of the model. With a lot more time and development, my project could still improve significantly. I truly believe that using ZKPs to help with sensitive data in AI models is a true possibility in the future, and it has the ability to completely revolutionize the world of AI.

Acknowledgments

I would like to acknowledge the help I have received from:

My parents for always encouraging me in my exploration of Computer Science. They have always supported me and make sure I have everything I need.

My mentoring teacher, Zita Murphy, for organising everything with SciFest and for the help and support she has given me towards my projects, and with preparation for the National Final.

My school supporting me with my journey through SciFest and the National Final 2024. I hope I get to represent the school many more times in the future.

The many YouTube content creators for posting free tutorials on how ZKPs work and the mathematical aspects to them, and tutorials on linear regression models.

Appendices - My Entire Script

Here you will find all my code for my ZKP and AI model. I decided that this section is the best place to include the code as it would become quite tedious and repetitive to comment on every line of code in my early sections. All of the final code and code from early iterations can be viewed on my GitHub Repository, as typing the code here would take up too many pages. The entire project directory and code can be viewed here with the QR code.

References

Abhishek, Allena Vankata Sai. 2023. *Kaggle.* Accessed April 29, 2024. https://www.kaggle.com/datasets/abhishek14398/salary-dataset-simple-linear-regression?rvi=1.

Alexandra, Boldyreva, Chenette Nathan, and O'Neill Adam. 2011. *Order-Preserving Encryption Revisited.* Advances in Cryptology - CRYPTO 2011.

Bhanot, Rajdeep, and Hans Rahul. 2015. " "A review and comparative analysis of various encryption algorithms."." *International Journal of Security and Its Applications 9* no. 4 289-306.

Bi, Qifang, Goodman Katherine E, Kaminsky Joshua, and Lessler Justin. 2019. " What is machine learning? A primer for the epidemiologist." *American journal of epidemiology 188, no. 12* 2222-2239.

Chi, Lianhua, and Zhu Xingquan. 2017. """Hashing techniques: A survey and taxonomy."." *ACM Computing Surveys (Csur) 50* (ACM Computing Surveys (Csur) 50) no. 1 : 1-36.

Corona, Erika, and Filippo Eros Pani. 2013. """A review of lean-kanban approaches in the software development."." *WSEAS transactions on information science and applications 10,.* 1-13.

dcbuilder.eth, and Worldcoin Team. 2023. "Intro to ZKML." *World.* 23 February. Accessed November 27, 2024. https://world.org/blog/engineering/intro-to-zkml.

Fowler, Martin, and Jim Highsmith. 2001. """The agile manifesto."." *Software development 9* 8: 28-35.

Gurjot. 2024. *GeeksForGeeks.* 16 April. Accessed April 29, 2024. https://www.geeksforgeeks.org/linear-regression-formula/.

Hasan, Jahid. 2019. “Overview and applications of zero knowledge proof (ZKP). .” *International Journal of Computer Science and Network 8, no. 5* 2277-5420.

Krawczyk, Hugo, Mihir Bellare, and Canetti Ran. 1997. “HMAC: Keyed-hashing for message authentication.” No. rfc2104.

Loeber, Patrick. 2021. *Python Flask Beginner Tutorial - Todo App - Crash Course*. https://youtu.be/yKHJsLUENl0?si=1QD3M1uN5Mg9-T7v.

Tranmer, Mark, and Mark Elliot. 2008. “ "Multiple linear regression."." *The Cathie Marsh Centre for Census and Survey Research (CCSR) 5, no. 5 (2008): 1-5*.

2024. “What is HMAC (Hash Based Message Authentication Code)).” *GeeksForGeeks*. 01 July. Accessed November 17, 2024. https://www.geeksforgeeks.org/what-is-hmachash-based-message-authentication-c.

Xing, Zhibo, Zhang Zijian, Liu Jiamou, Zhang Ziang, Li Meng, Zhu Liehuang, and Russello. Giovanni. 2023. “Zero-knowledge proof meets machine learning in verifiability: A survey.” *arXiv preprint arXiv:2310.14848*.