Email classifier design document

Document explaining the design of the email classifier application.
Produced for the project in the course CS4125 at University of Limerick.

Christoffer Näs - 24271322

Selin Taskin - 24284378

Patrick Vorreiter - 24284335

2024-11-24

1 Table of contents

2 UML diagrams

2.1 Class diagram

2.1.1 Client and Cli

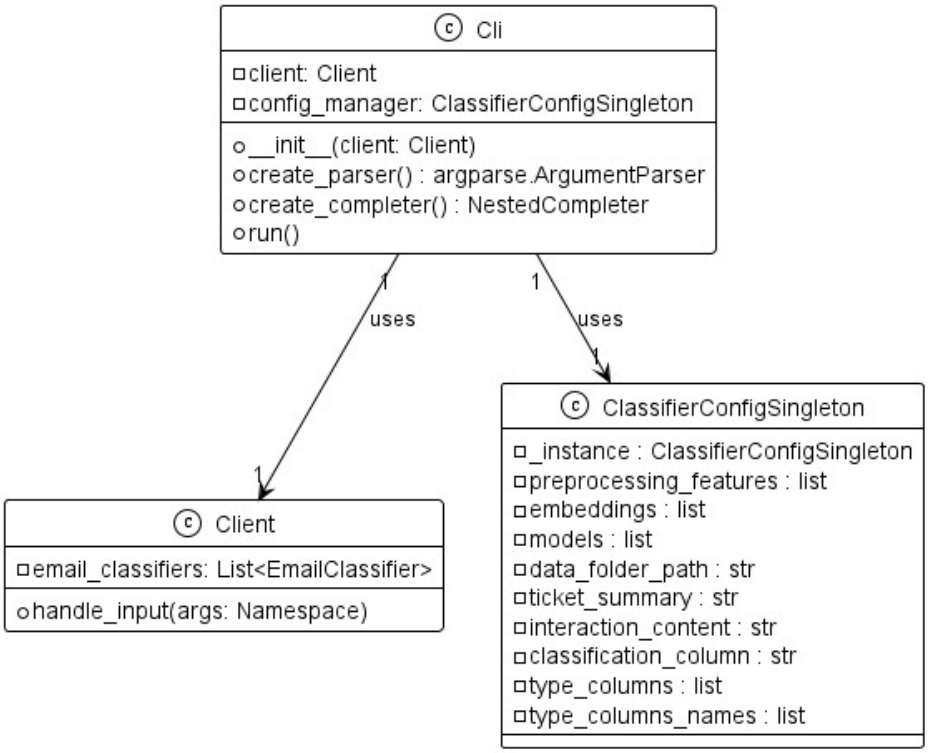

Figure 1. Diagram explaining the relationships between Cli and Client.

The *Cli* and the *Client* utilizes separation of concern where the *Cli* provides a way for the user to interact with the system and the *Client* contains the business logic.

2.1.2 ClassifierConfigSingleton (Singleton design pattern)

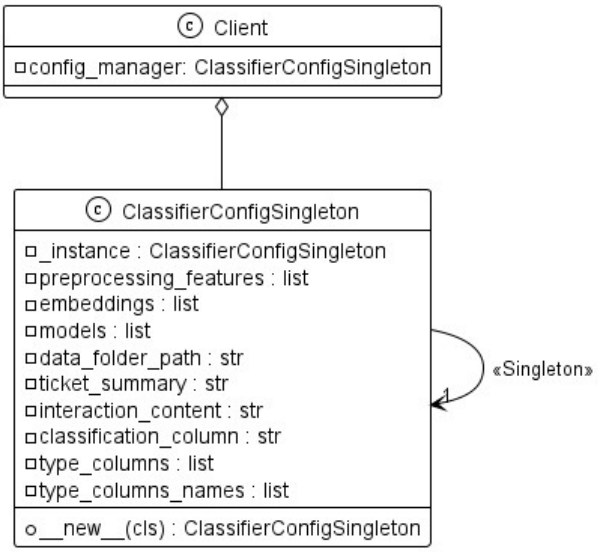

Figure 2.UML Class diagram for the ClassifierConfigSingleton.

The *ClassifierConfigSingleton* is used for global configuration in the email classifier and enables a single global access point. Figure 2 contains all of the attributes of the *ClassifierConfigSingleton*.

2.1.3 Client commands managing (Command Pattern)

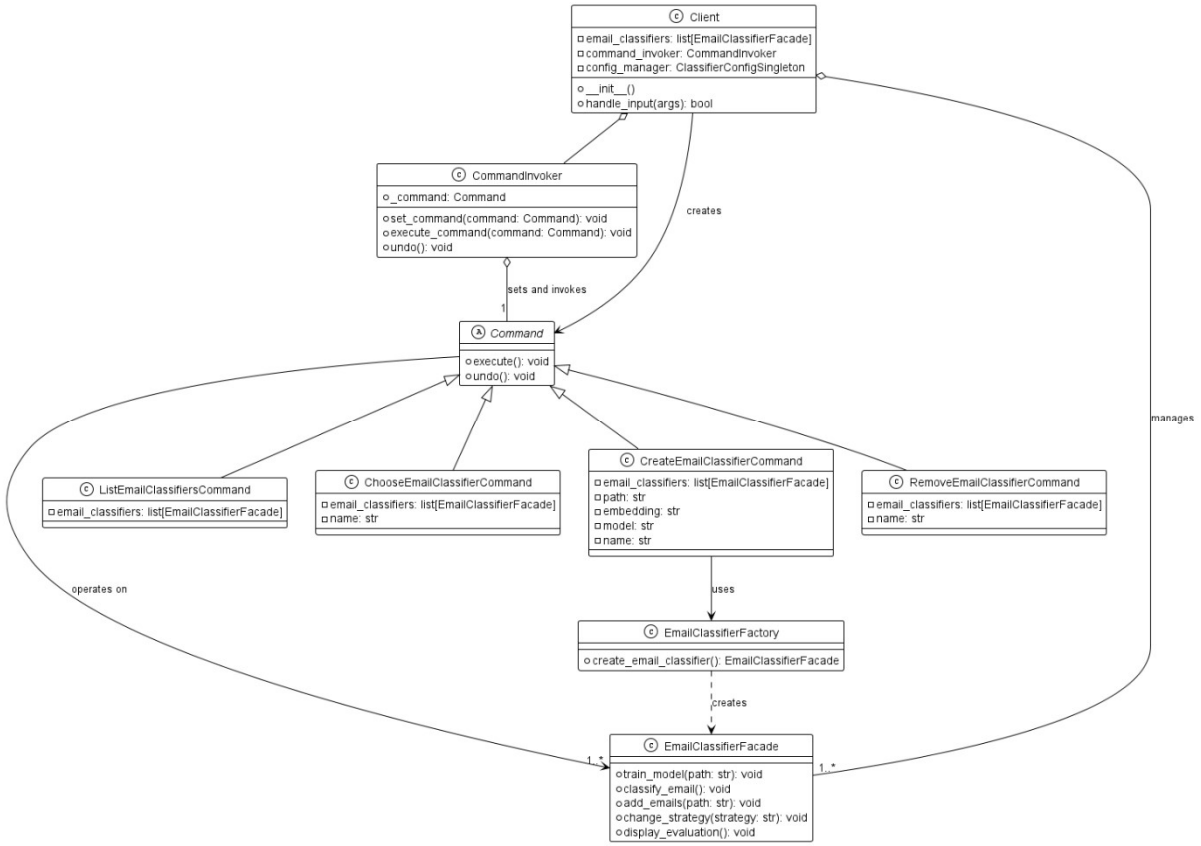

Figure 3.Explanation of commands operating on the list of email classifiers.

The *Client* handles inputs from the Cli and converts them into commands that operates on the list of *EmailClassifierFacade* objects. These commands are set and executed by the *CommandInvoker*.

2.1.4 Client commands operations (Command pattern)

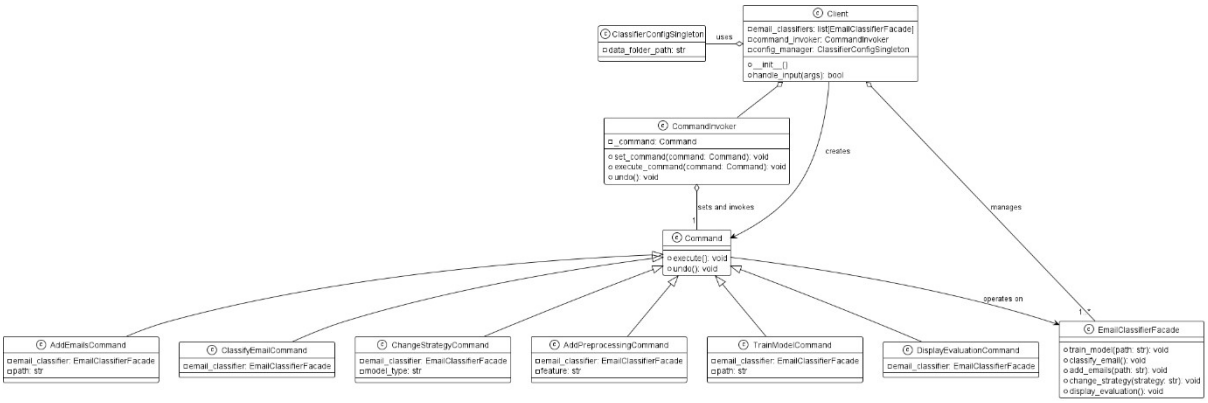

Figure 4. Explanation of commands operating on individual email classifiers.

The *Client* also handles inputs and create commands that operates on individual *EmailClassifierFacade* objects. The relationships of these commands can be seen in Figure 4.

2.1.5 EmailClassifierFacade (Facade Pattern)

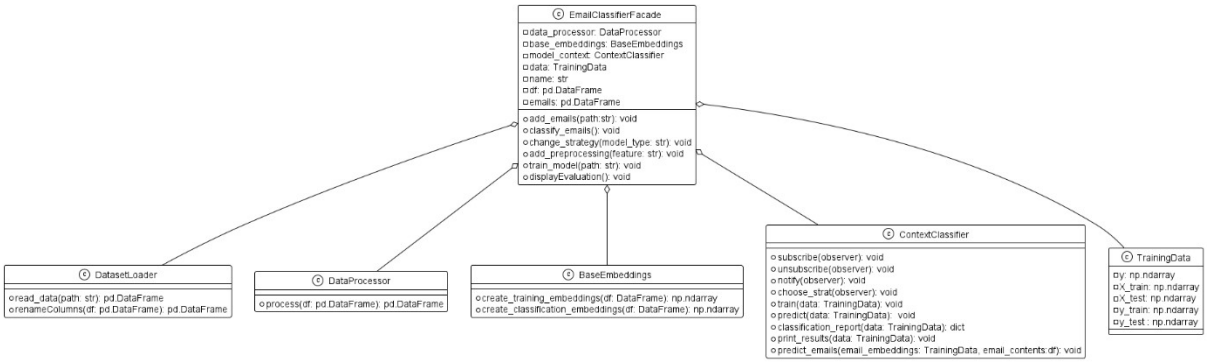

Figure 5.UML class diagram for the email classifier facade.

The *EmailClassifierFacade* makes use of the Facade pattern and hides complex underlaying logic. It contains all the necessary components required to preprocess data, create embeddings, train model, and classify emails.

2.1.6 Data Processor(Decorator pattern)

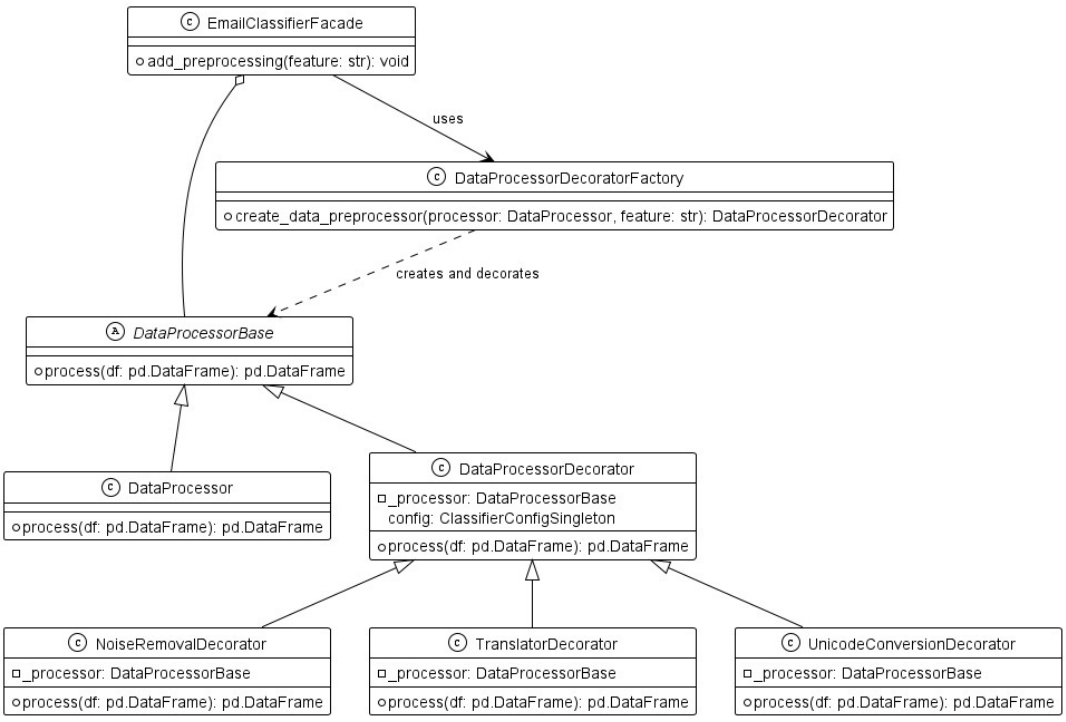

Figure 6. Diagram explaining the relationships connected to the Data Processor.

The *EmailClassifierFacade* contains a *DataProcessorBase* used for preprocessing data. Preprocessing can be extended with the help of a *DataProcessorDecorator*.

2.1.7 EmbeddingsFactory (Factory Pattern)

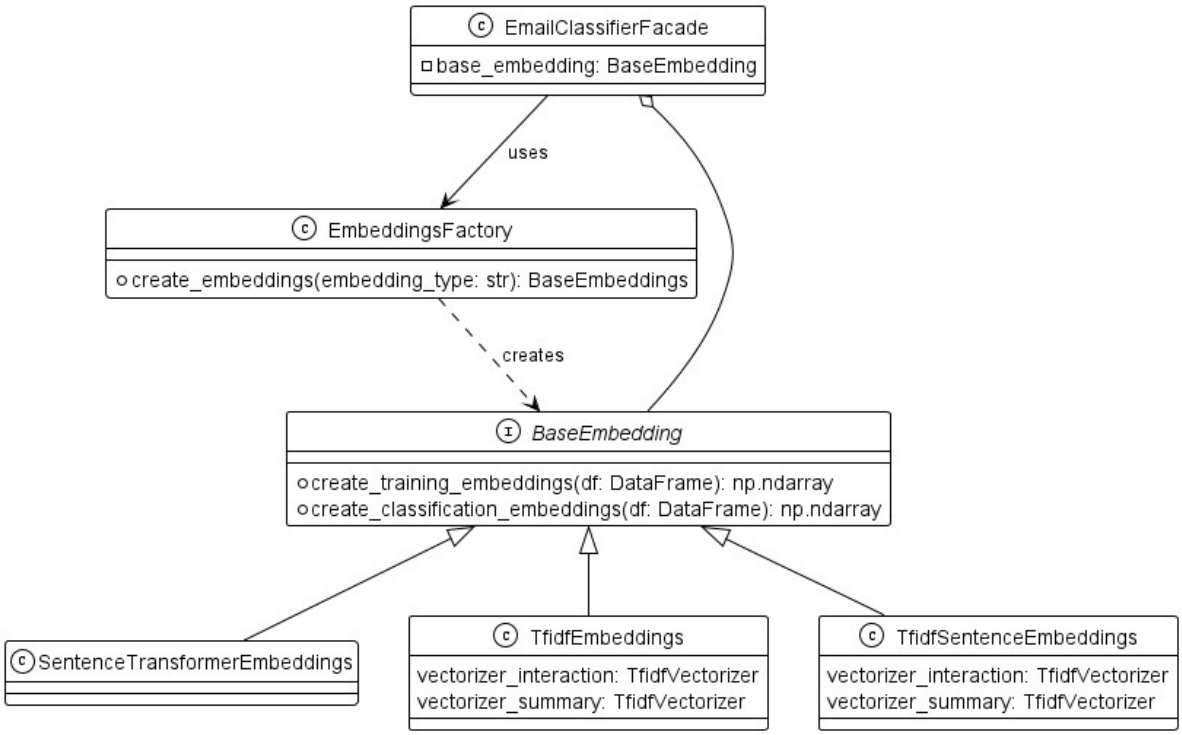

Figure 7. Diagram explaining the relationships to BaseEmbedding.

The *BaseEmbedding* is used to transform data in the *EmailClassifierFacade* into embeddings that can be used in the *ContextClassifier*.

2.1.8 ContextClassifier

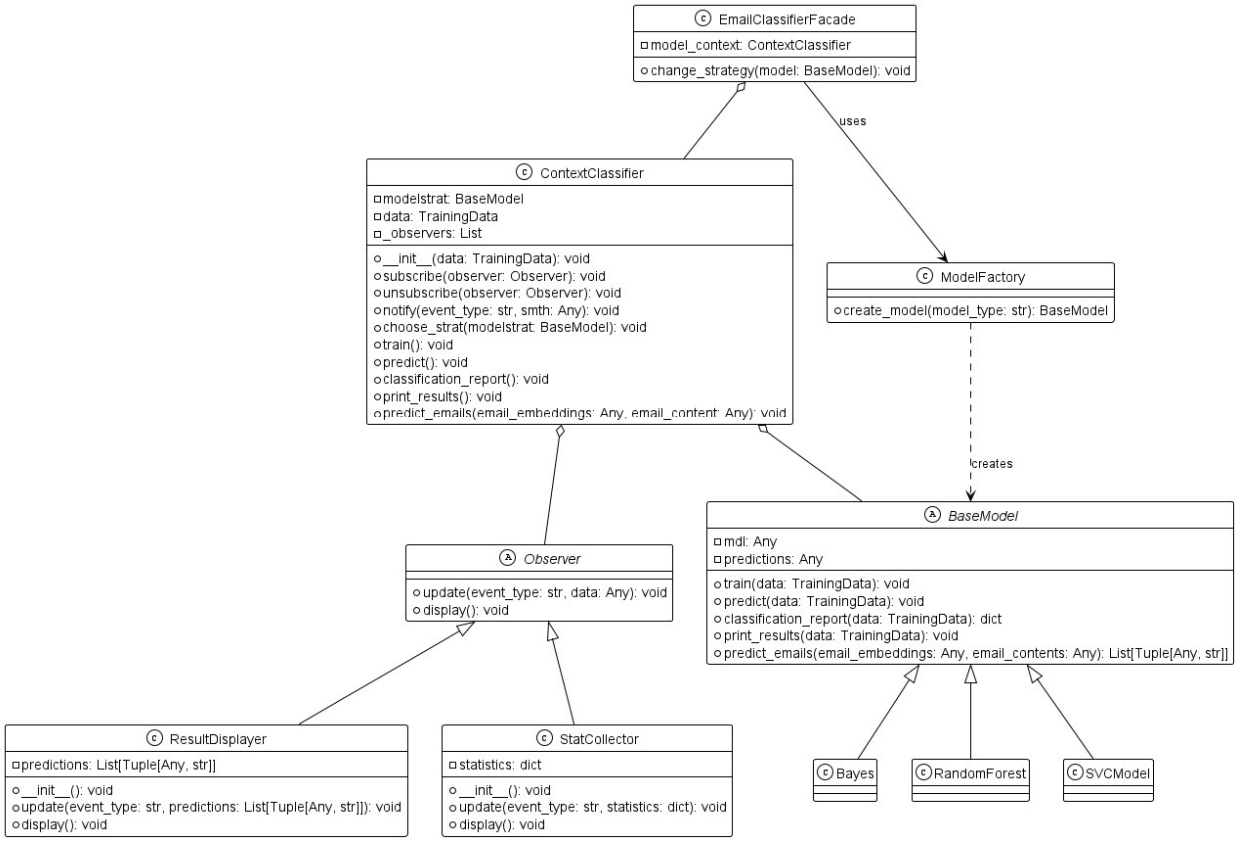

Figure 8. Diagram explaining the all the relationships to the ContextClassifier.

The *ContextClassifier* contains all the necessary logic for training the machine learning model and using the model to classify emails. It uses the *TrainingData* to train models, which contains the transformed data created by the *BaseEmbedding*, split for training.

2.1.9 ContextClassifier(strategy pattern)

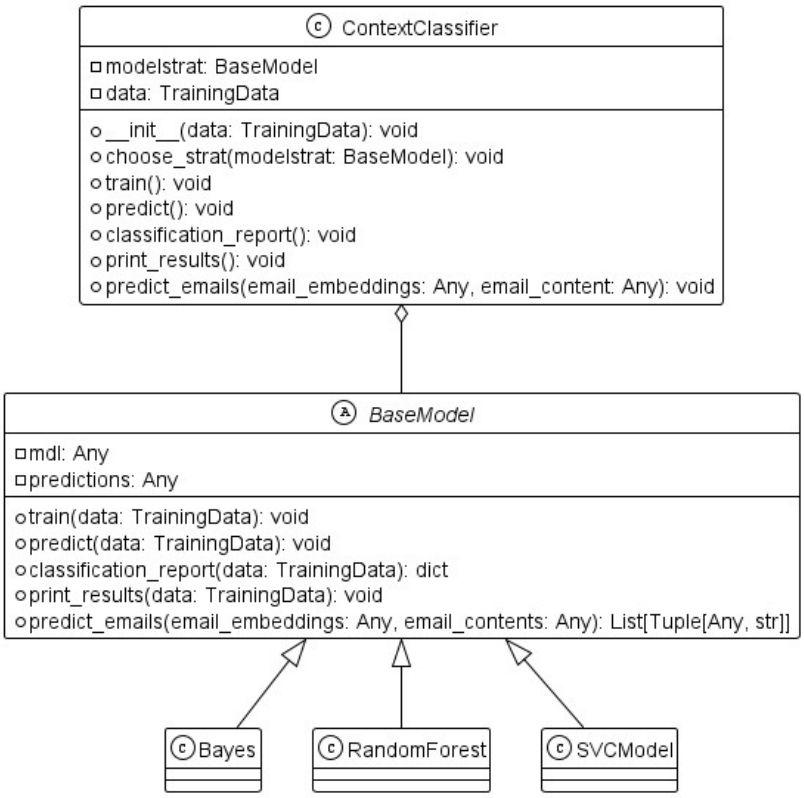

Figure 9. Diagram explaining the strategy pattern in detail.

2.1.10 Observer (Observer Pattern)

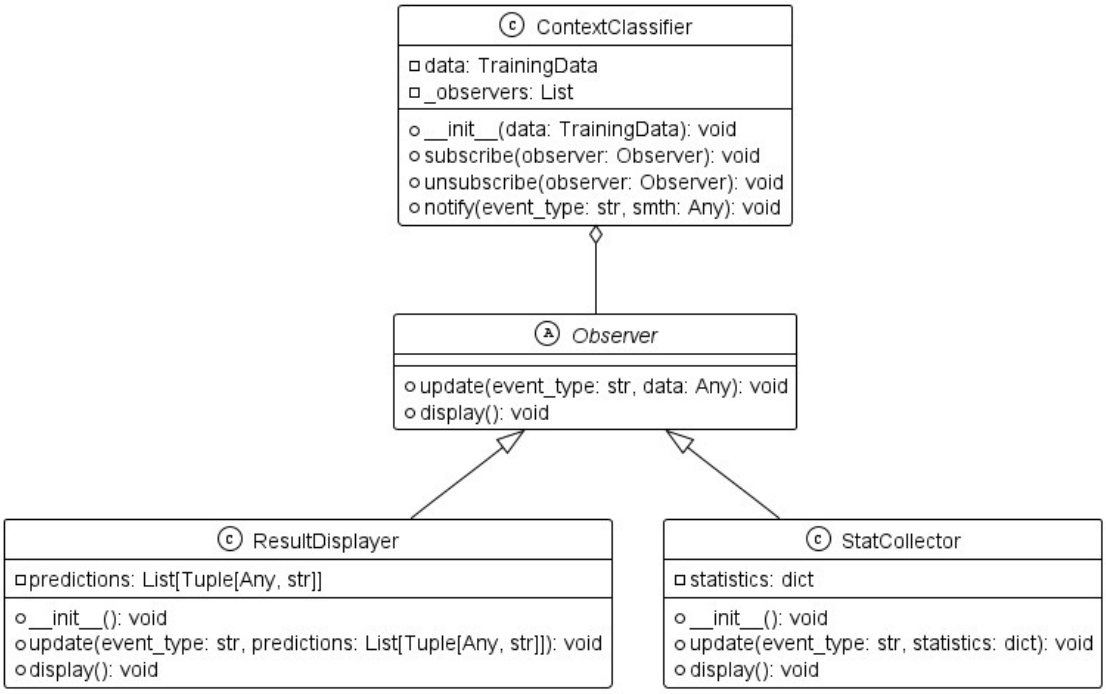

Figure 10.Diagram explaining the observer pattern in detail.

2.2 Sequence diagram

2.2.1 Add email process

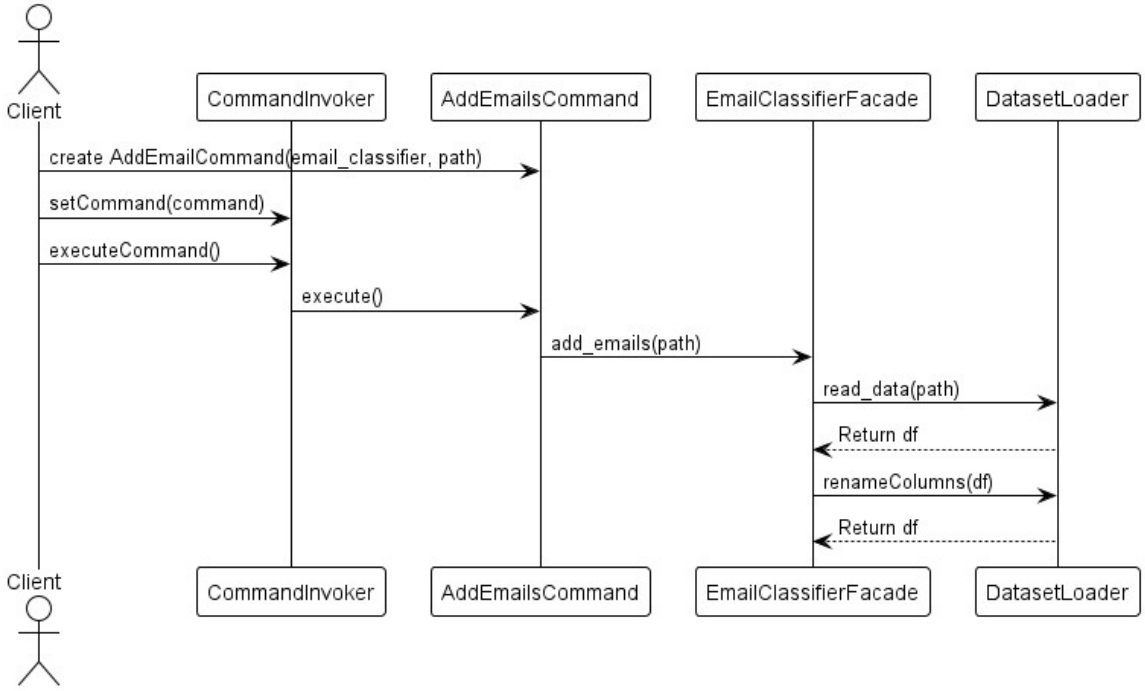

Figure 11. Diagram explaining the interactions when executing the add email command.

The diagram shows the interaction between the systems objects over time for the process of executing the add email command. The diagram shows the correct command creation and execution following the command pattern principles. This process is simplified in the other diagrams to save space.

2.2.2 Classify email process

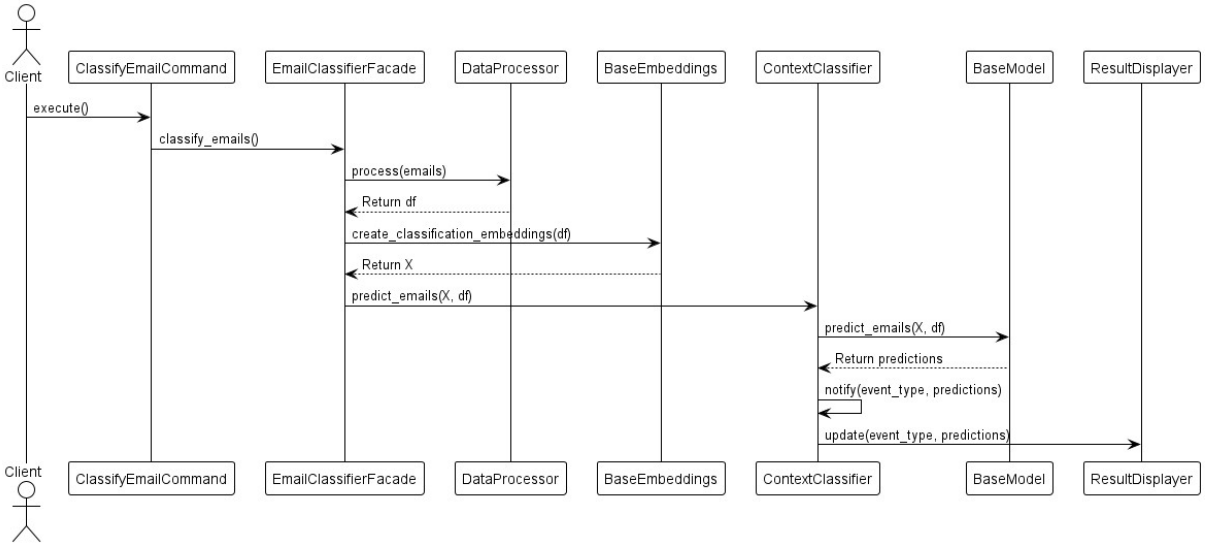

Figure 12.Diagram explaining the interactions when executing the classify email command.

The diagram shows the interaction between the systems objects over time for the process of executing the classify email command. The command execution is simplified in this diagram to save space.

2.2.3 Create email classifier process

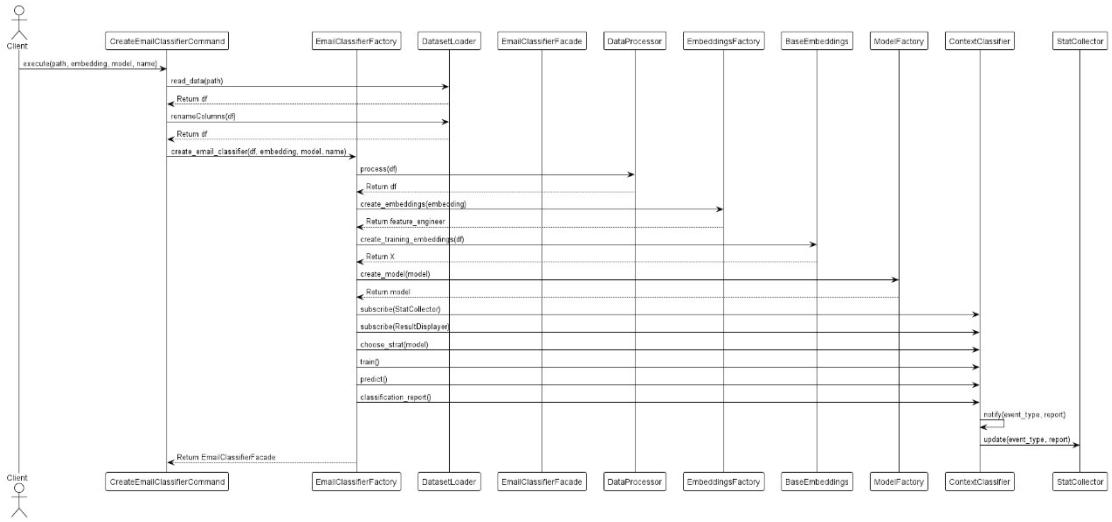

Figure 13. Diagram explaining the interactions when executing the create email classifier command.

The diagram shows the interaction between the systems objects over time for the process of executing the create email classifier command. The command execution is simplified in this diagram to save space.

2.3 Use case diagram

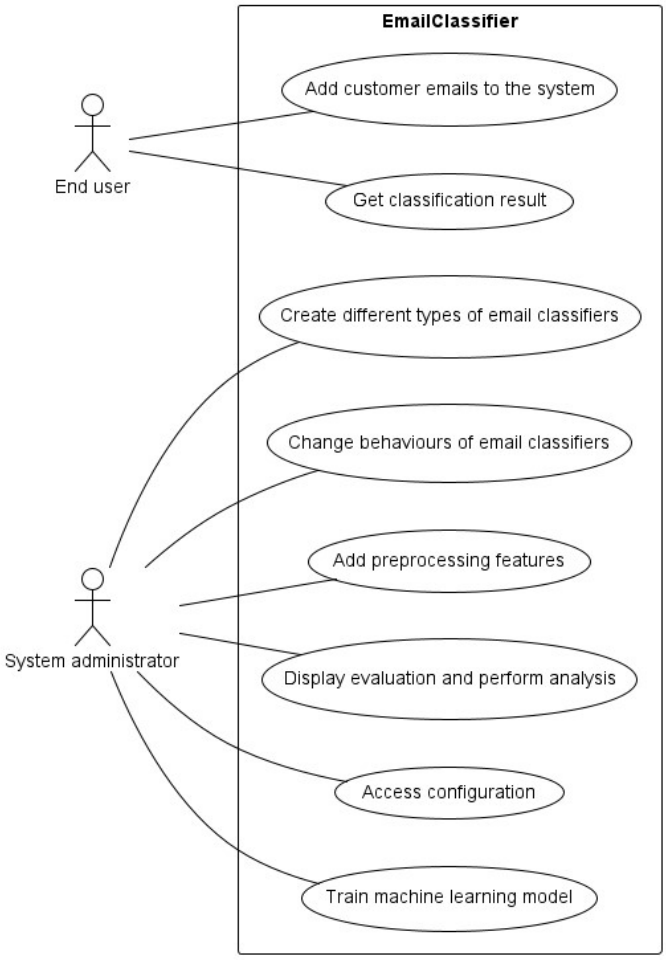

Figure 14.Use case diagram for the email classifier.

2.3.1 Create different types of email classifiers

Actor: System administrator
Description: The system administrator wishes to create different types of email classifiers using different inputs and configurations. This flexibility allows the creation of classifiers optimized for specific scenarios.
Precondition: Valid inputs and configurations exist.
Postcondition: An email classifier exists.
Main Flow:

1. The system administrator requests to create an email classifier.
2. The system administrator chooses input values and configurations.
3. The system creates an email classifier based on the chosen input values and configurations.

2.3.2 Change behaviours of email classifiers

Actor: System administrator
Description: The system administrator wishes to change the behaviours, such as algorithm types, of an email classifier to select the best-performing one for given data.
Precondition: An email classifier exists.
Postcondition: The email classifier is updated with the modified behaviour.
Main Flow:

1. The system administrator requests to change an email classifier.
2. The system administrator chooses a behaviour to change.
3. The system administrator chooses how to change chosen behaviour.
4. The system changes the behaviour of the email classifier.

2.3.3 Add preprocessing features

Actor: System administrator
Description: The system administrator wishes to add a new preprocessing feature to an email classifier to enhance classifiers accuracy for specific email data.
Precondition: An email classifier and an extra preprocessing feature exist.
Postcondition: The email classifier now includes the selected preprocessing feature.
Main Flow:

1. The system administrator chooses an email classifier.
2. The system administrator chooses a preprocessing feature to add.
3. The system extends the behaviours of the email classifier with the chosen preprocessing feature.

2.3.4 Display evaluation and perform analysis

Actor: System administrator
Description: The system administrator wishes to see the performance and evaluation results of the classification models. This information is used to evaluate the performance of the classification.
Precondition: A trained and tested model exists.
Postcondition: Performance evaluation metrics are available for the system administrator to review.
Main Flow:

1. The system administrator requests to see the evaluation results.
2. The system displays evaluation results and performance metrics including accuracy score to the system administrator.

2.3.5 Access configuration

Actor: System administrator
Description: The system administrator wishes to access the configurations of the email classifier. The configuration information is used to evaluate previous classification processes.
Precondition: Configurations for an email classifier exist and are accessible.
Postcondition: The system administrator views the current configuration details of the email classifier.

Main Flow:

1. The system administrator requests to access the configuration.
2. The system displays the current configuration details to the system administrator.

2.3.6 Train machine learning model

Actor: System administrator
Description: The system administrator wishes to train and retrain a machine learning model with different configurations, such as embeddings and algorithms, to be able to adapt and improve the model.
Precondition: Existing model configurations available.
Postcondition: A machine learning model is trained.
Main Flow:

1. The system administrator chooses a certain configuration to train the machine learning model.
2. The system trains the machine learning model.

2.3.7 Add customer emails to the system

Actor: End user
Description: The end user wants to add emails to the classifier to get a classification result from the model.
Precondition: Customer emails are available and correctly formatted for input.
Postcondition: Customer emails are stored in the system.
Main Flow:

1. End user adds the email that they want to classify to the system.
2. The system stores the emails in the system for future classification.

2.3.8 Get classification result

Actor: End User
Description: The end user wants to receive classification results for emails added to the system to more effectively and efficiently answer customer emails.
Precondition: Emails have been added to the system.
Postcondition: The classification results for the emails are available to the end user.
Main Flow:

1. The end user selects that he wishes to get classification results for added emails.
2. The system classifies emails and displays the classifications for the end user.

3 Design pattern usages

3.1 Singleton Pattern

The Singleton pattern ensures that only one instance of a specific class exists throughout the application. In this project, this pattern is used for managing the configuration settings. The *ClassifierConfigSingleton* contains configuration settings for the *Cli* and for other components that must stay the same. By using the singleton patterns, the system avoids issues with inconsistent configurations. Additionally, if there are any changes in the configuration, then these changes are reflected across the system. Hence, this pattern provides a centralised and globally accessible configuration object. The singleton design pattern also ensures the reliability of the system by ensuring that the configuration is consistent throughout the whole system. Additionally, it improves maintainability by offering a single point of access for configuration, simplifying updates and troubleshooting.

3.2 Observer Pattern

The Observer pattern is essential in communication between the subject and the observers. The observers are notified of the changes in the state. In this project, there are two different observers. These observers are *StatCollector* and *ResultDisplayer.* These observers subscribe to the *ContextClassifier* object and get updated and notified of the changes in this *ContextClassifie*r object.

The *StatCollector* observer observes the training of the machine learning model. After the model is trained and tested on the split data, the metrics are updated by the *ContextClassifie*r and displayed by the *StatCollector.* The *ResultDisplayer* observes the classification of the new input data. This new input data are the test emails that the employee wants to classify. The results of the classification, such as which category an email belongs to, are displayed by *ResultDisplayer.*

This observer pattern design ensures the modularity of our system. This way, each of the observers is identified continuously, even if there is any change happening in the ContextClassifier. Additionally, it also creates flexibility in the system as it is possible to add more subscribers to the system.

3.3 Strategy Pattern

The Strategy pattern is implemented to dynamically switch between different classification models, such as using Random Forest, support vector machines, and Bayes. The *ContextClassifier* is responsible for choosing the strategy. As it is responsible for changing between the models, also classification tasks are done based on the chosen strategy by the *ContextClassifier* object. Furthermore, the prediction of the class of an email and the calculation of the accuracy and other metrics are done with the context of the *ContextClassifier* object.

With this pattern, the system becomes more flexible as individual strategies can be changed separately. Additionally, these patterns make the overall system more maintainable by allowing strategies to be separate.

3.4 Factory Pattern

The Factory pattern is used to encapsulate the creation of classifier and embedding objects. The factory pattern is used in different parts of the system. There is an *EmbeddingsFactory* that is responsible for creating different types of embedding objects, such as TF-IDF or sentence transformer. *ModelFactory* class is responsible for creating the objects for different types of machine learning models. Additionally, *EmailClassifierFactory* class is responsible for creating email classification objects.

This pattern increases the system's scalability by making it easy to create new types of models, embeddings, and classifier objects. Also, this pattern decouples the instantiation logic from the rest of the system and makes it easier to create classifiers in the future without modifying existing code.

3.5 Facade Pattern

The Facade pattern simplifies interactions with complex subsystems. With the `Facade` pattern, the interactions between the preprocessing of the data, training of the machine learning model, and evaluation are unified. *EmailClassifierFacade* class utilizes this design pattern and provides one interface for different components to execute without needing to interact directly with individual subsystems.

This pattern supports modularity by reducing the complexity of the interactions between different systems. In addition, this design pattern improves the usability and reduces coupling in the system.

3.6 Decorator Pattern

The Decorator pattern allows to add additional functionality to an object at runtime without modifying its structure. In this project, it is applied to the data processing pipeline, enabling the dynamic addition of preprocessing steps to modify the data before model training or prediction. The *DataProcessor* class serves as the base, while decorators like *NoiseRemovalDecorator*, *TranslatorDecorator*, and *UnicodeConversionDecorator* extend its functionality. The pattern ensures that preprocessing steps can be used in various combinations without altering the core logic. For instance, deduplication can be combined with noise removal or translation, all without changing the existing processing logic. This

flexibility makes it easy to customize the pipeline, while modularity and maintainability are ensured by isolating each transformation in its own decorator.

3.7 Command pattern

The Command pattern is used in the *Client.* It is used to encapsulate requests for operations on the email classifiers in the *Client* class as an object. This ensures clean and modular command handling. Each command executes a specific task, preserves previous states, and is managed by the *CommandInvoker*. This enables the ability to undo a command. The command pattern used decouples logic, simplifies the addition of new commands, enhances reusability, and ensures robust state management. This contributes to making the system more flexible, reliable, and scalable.