ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Информационная безопасность – это сохранение и защита информации, а также ее важнейших элементов, в том числе системы и оборудование, предназначенные для использования, сбережения и передачи этой информации. Другими словами, это набор технологий, стандартов и методов управления, которые необходимы для защиты информационной безопасности.

Существует довольно много информационных атак нацеленные на разные цели, например кража информации, ее порча, или вывод из строя оборудования. Для достижения этих целей применяю разные атаки, например фишинг, брутфорс-атаки, DoS -атаки, Man-in-the-Middle и многие другие.

Сейчас все больше компаний стараются уделять внимание информационной безопасности, создают специальные отделы по защите и борьбе с информационными угрозами. Но не только компании подвержены информационным атакам, обычный пользователи тоже нередко страдают от них. Пользователи бывают разные, кто-то осведомлен больше и знает, как противостоять некоторым угрозам, а кто-то даже не задумывается об этом, не осознавая всей важности, использует везде один и тот же пароль. Так или иначе даже самый опытный пользователь может попасть на удочку злоумышленников.

Одна из таких удочек – спам письма. Спам (от англ. «spiced ham» - «ветчина со специями») — это массовая рассылка нежелательных писем. Чаще всего спам — это письмо электронной почты, которое отправляется сразу на большое количество адресов, но оно также может быть доставлено и через мгновенные сообщения, SMS и социальные сети. Спам может содержать в себе фишинг, целенаправленный фишинг (spearphishing), вишинг (vishing), смишинг (smishing), а также распространение вредоносных вложений или ссылок.

ГЛАВА 1. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

1.1 Способы борьбы со спамом

Существует немало способов противостоять спаму. Для лучшей защиты - способы реализуются вместе, а не выбирается какой-то один. К примеру, существует репутационный рейтинг отправителя письма. Репутация – это баллы от 0 до 100, где 100 – это идеальная репутация, а 0 – однозначно спам. Таким образом, если у отправителя рейтинг 70, то это означает, что в 30% процентах случаев, считается, что письмо относится к спаму. Репутация зарабатывается со временем и вычисляется на основе отзывов пользователей, если пользователь прочитал и отметил «это не спам», отправитель получает плюс к репутации, а если пользователь отметил, что «это спам», то, соответственно, отправитель получает минус. Часто если пользователь просто прочитал письмо и ничего не сделал это приравнивается к отметке «это не спам». Стоит понимать тот факт, что новые пользователи по началу не имеют репутации и, что спам не всегда имеет отрицательную направленность, поэтому стоит это учитывать при проведении массовых рассылок и увеличивать ее объем постепенно. На основе этой репутации строиться черный список IP-адресов и доменов. Такой список называется DNSBL (DNS blocklist). Если отправитель находится в блок-листе, то сообщение помечается как спам.

Также применяется фильтрация на основе формальных признаков. Происходит анализ формальных признаков сообщения, в котором обращается внимание на отсутствие отправителя, большое количество получателей, наличие и размер вложенных файлов. Письма, которые не прошли такую проверку, отправляются в спам.

Проверка служебных заголовков, информация из которых, позволяет спам фильтрам определить является письмо спамом или нет. Например, из заголовка X-Mailer можно узнать название программы отправителя. В письме,

отправленном с почтового сервиса Яндекс, поле X-Mailer содержит запись: Yamail [https://yandex.ru] 5.0.

Проверка протоколов аутентификации. Аутентификация — это способ проверить подлинность отправителя. Письмо, которое не прошло аутентификацию, классифицируется как спам или подозрительное. Существует три метода аутентификации: DKIM, SPF и DMARC. Подпись DKIM подтверждает, что сообщение отправил владелец домена. SPF блокирует отправку сообщений с ip-адресов, которых нет в записи. DMARC защищает домен от подмены электронной почты.

Фильтрация по образцу письма. На основе собранной базы спамерских писем создается их шаблон, с которыми сопоставляются следующие письма. Методы нечеткого сравнения способны распознавать даже видоизмененные спамерские сообщения. Фильтры учитывают десятки тысяч факторов, таких как регистр и цвета шрифта, знаки препинания, количество и порядок слов в предложениях.

Фильтрация по словам или, можно сказать по-другому, на основе анализа содержания текста письма. Зачастую спамерские письма очень хорошо видно, по таким популярным выражениям: без опыта, без риска, беспроцентный кредит, бесплатно, возврат денежных средств, выгодная сделка, выигрыш, выплата, дополнительный доход, дорогой друг, криптовалюта, лотерея, миллион, позвоните, поздравляем, розыгрыш, страховые накопления, скидка, специальное предложение, шанс, эксклюзив и так далее. О том, как можно проанализировать текст письма и о реализации данного фильтра пойдет речь дальше.

1.2 Спам-фильтр на основе анализа содержания текста письма

Обработкой естественного языка занимается направление NLP (Natural Language Processing). Это направление в машинном обучении, посвященное распознанию, генерации и обработке устной и письменной человеческой речи.

Направление находится на стыке дисциплин искусственного интеллекта и лингвистики.

Инженеры-программисты разрабатывают механизмы, позволяющие взаимодействовать компьютерам и людям посредствам естественного языка. Благодаря NLP компьютеры могут читать, интерпретировать, понимать человеческий язык, а также выдавать ответные результаты. Как правило, обработка основана на уровне интеллекта машины, расшифровывающего сообщения человека в значимую для нее информацию.

Приложения NPL окружают нас повсюду. Это поиск в Google или Яндексе, машинный перевод, чат-боты, виртуальные ассистенты вроде Siri, Алисы, Салюта от Сбера и пр. NLP применяется в digital-рекламе, сфере безопасности и многих других.

NLP решает множество задач обработки естественного языка, например: распознавание речи, обработка текста, извлечение информации, анализ информации, генерация текста и речи, автоматический пересказ, машинный перевод.

В данной курсовой работе реализовано решение задачи анализа текста для определения к какой категории (спам или не спам) отнести тот или иной текст (сообщение). В современном мире эта задача является весьма актуальной, например, по статистическим данным на 2019 год, предоставленные securelist.ru, свидетельствуют, что в мировом масштабе на почтовый трафик приходится более 50% спама (рисунок 1.1).

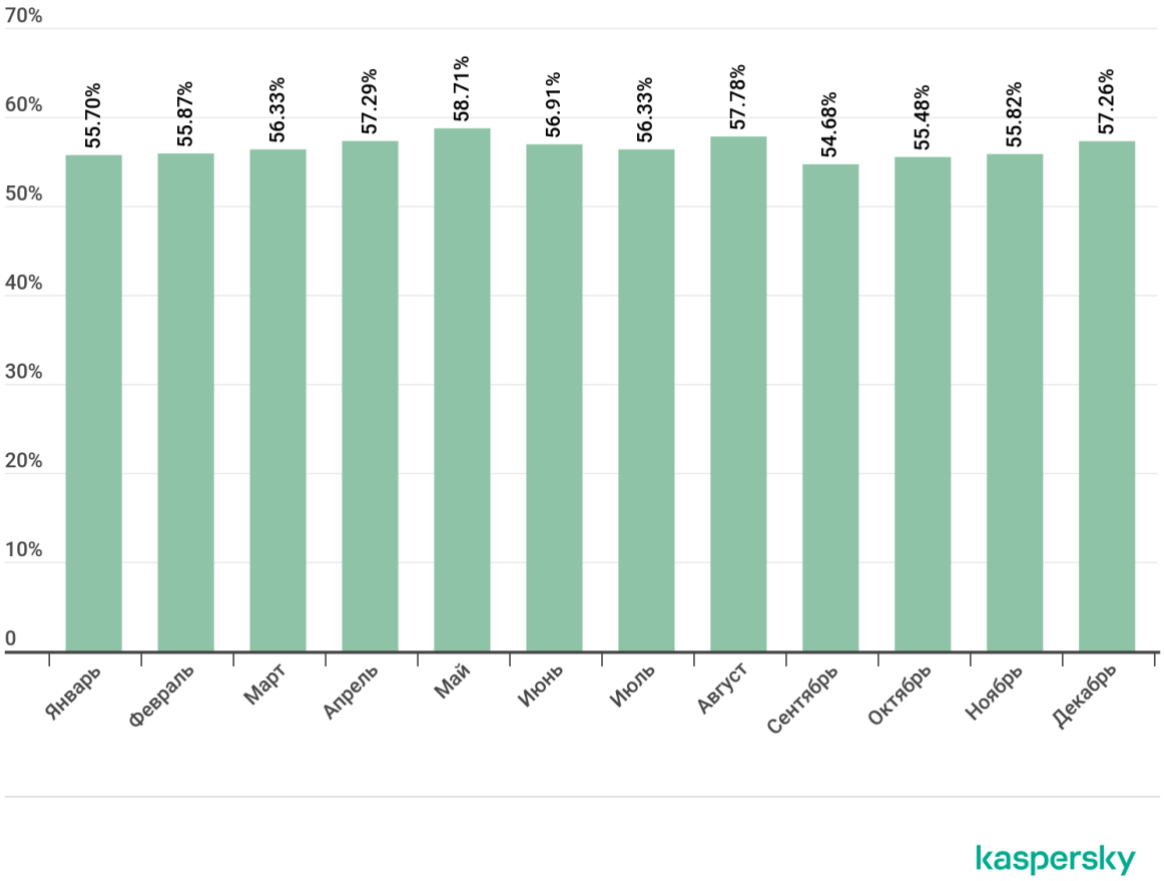

Рисунок 1.1 – Доля спама в мировом почтовом трафике, 2019 г.

Также нелишним будет упомянуть тот факт, что количество вредоносных программ, которые были найдены антивирусом за 2019 год, составляет 186 005 096 штук. Количество срабатываний почтового антивируса за 2019 год по месяцам отражен на рисунке 1.2.

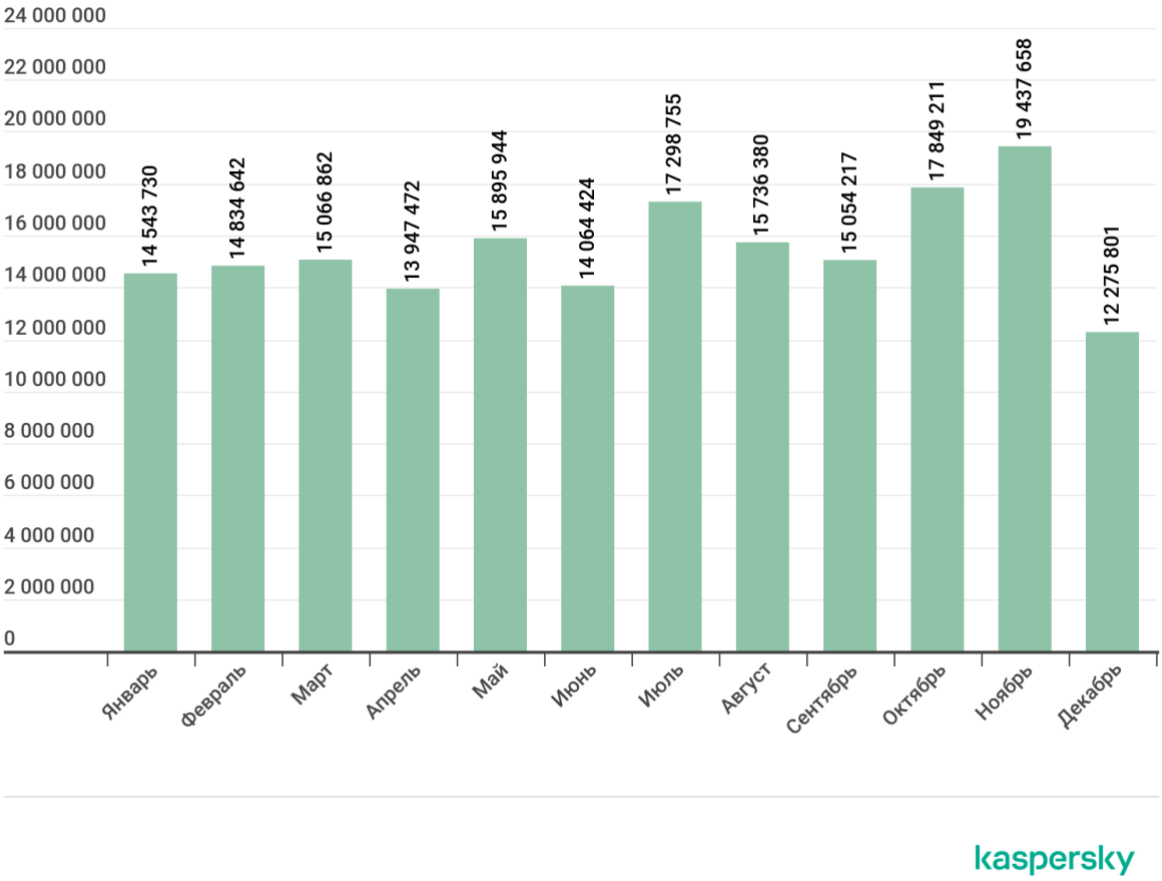

Рисунок 1.2 – Количество срабатываний почтового антивируса, 2019 г.

Цифра не маленькая и стоит понимать, что это только то, что удалось обнаружить. Это лишний раз показывает, насколько борьба со спамом является важной задачей.

Здесь также важно уточнить, что нельзя считать спамом любое сообщение, которое хоть как-то на него похоже, потому что это может быть сообщение, которое несет пользу для пользователя и он его ждет. Здесь нужно найти баланс.

1.3 Обработка текста

Если сообщение прошло предыдущие линии защиты, описанные в разделе 1.1, то для него наступает момент анализа содержимого. Нельзя просто взять текст и передать его алгоритму для определения является ли текст спамом. Данный текст нужно сначала «приготовить» или преобразовать в вид,

доступный для восприятия компьютером. Процесс готовки данных называется препроцессингом и включает в себя несколько этапов.

1.3.1 Токенизация по предложениям и по словам

Токенизация (иногда – сегментация) по предложениям – это процесс разделения письменного языка на предложения-компоненты. В английском, русском и некоторых других языках мы можем вычленять предложение каждый раз, когда находим определенный знак пунктуации – точку. Но точка используется не только для конца предложений, она также применяется при сокращении слов. В этом случае, чтобы предотвратить неправильную расстановку границ предложений, сильно поможет таблица сокращений.

Токенизация (иногда – сегментация) по словам – это процесс разделения предложений на слова-компоненты. В английском, русском и многих других языках, использующих ту или иную версию латинского алфавита, пробел – это неплохой разделитель слов. Но и здесь не все так просто, потому что могут попасться такие слова, которые являются одним словом, но при этом будут писаться через пробел.

1.3.2 Лемматизация и стемминг текста

Обычно тексты содержат разные грамматические формы одного и того же слова, а также могут встречаться однокоренные слова. Лемматизация и стемминг преследуют цель привести все встречающиеся словоформы к одной, нормальной словарной форме. Пример приведения разных словоформ к одной: dog, dogs, dog’s приводятся к словоформе dog.

Лемматизация и стемминг – это частные случаи нормализации.

Стемминг – это грубый эвристический процесс, который отрезает «лишнее» от корня слов, часто это приводит к потере словообразовательных суффиксов.

Лемматизация – это более тонкий процесс, который использует словарь и морфологический анализ, чтобы в итоге привести слово к его канонической форме – лемме.

Отличие этих операций в том, что стеммер действует без знания контекста и, соответственно, не понимает разницу между словами, которые имеют разный смысл в зависимости от части речи. Однако у стеммеров есть и свои преимущества: их проще внедрить, и они работают быстрее. Плюс, более низкая «аккуратность» может не иметь значения в некоторых случаях.

Примеры работы лемматизации и стемминга:

1. Слово good – это лемма для слова better. Стеммер не увидит эту связь, так как здесь нужно сверяться со словарем.
2. Слово play – это базовая форма слова playing. Тут справятся и стемминг, и лемматизация.

1.3.3 Стоп-слова

Стоп-слова – это слова, которые выкидываются из текста до/после обработки текста. При применении машинного обучения к текстам, такие слова могут добавлять много шума, поэтому необходимо избавляться от нерелевантных слов.

В качестве стоп-слова обычно используются артикли, междометия, союзы и т.д., в общем слова, которые не несут смысловой нагрузки. Не существует универсального списка стоп-слов, все зависит от конкретного случая.

1.3.4 Мешок слов

Алгоритмы машинного обучения не могут напрямую работать с сырым текстом, поэтому необходимо конвертировать тексты в наборы цифр (векторы). Это называется извлечением признаков.

Мешок слов – это популярная и простая техника извлечения признаков, используемая при работе с текстом. Она описывает вхождения каждого слова в текст.

Чтобы использовать модель, нужно:

– определить словарь известных слов (токенов);

– выбрать степень присутствия известных слов.

Любая информация о порядке или структуре слов игнорируется. Вот почему это называется мешком слов. Эта модель пытается понять, встречается ли знакомое слово в документе, но не знает, где именно оно встречается.

Интуиция подсказывает, что схожие документы имеют схожее содержимое. Также, благодаря содержимому, можно узнать кое-что о смысле документа.

1.4 Наивный байесовский алгоритм. Теорема Байеса

Наивный байесовский алгоритм — это вероятностный алгоритм машинного обучения, основанный на применении теоремы Байеса и используемый в самых разных задачах классификации.

Теорема Байеса — это простая математическая формула, используемая для вычисления условных вероятностей.

Условная вероятность — это вероятность наступления одного события при условии, что другое событие (по предположению, допущению, подтверждённому или неподтверждённому доказательством утверждению) уже произошло. Определить условную вероятности можно по формуле (1.1):

$$P(A|B) = \frac{P(B|A) * P(A)}{P(B)} \tag{1.1}$$

Формула (1.1) показывает, как часто происходит событие A при наступлении события B, обозначается как P(A|B) и имеет второе название «апостериорная вероятность». При этом необходимо знать:

1. как часто происходит событие B при наступлении события A, что обозначается в формуле как P(B|A);
2. какова вероятность того, что A не зависит от других событий, обозначаемая в формуле как P(A);
3. какова вероятность того, что B не зависит от других событий. В формуле она обозначается как P(B).

Можно сказать, что теорема Байеса — это способ определения вероятности исходя из знания других вероятностей.

Основным допущением наивного байесовского алгоритма является то, что каждая характеристика вносит независимый и равный вклад в конечный результат. Но допущения наивного байесовского алгоритма, как правило, некорректны в реальных задачах. Допущение о независимости всегда некорректно, но часто хорошо работает на практике. Поэтому алгоритм и называется наивным.

Для задачи классификации спама теорема Байеса примет следующий вид: определение условной вероятности для спам писем (формула (1.2)) и определение условной вероятности для не спам писем (формула (1.3)).

$$P(SPAM|WORDS) = \frac{P(WORDS|SPAM) * P(SPAM)}{P(WORDS)} \tag{1.2}$$

$$P(HAM|WORDS) = \frac{P(WORDS|HAM) * P(HAM)}{P(WORDS)} \tag{1.3}$$

В расширенном виде формулы (1.2) и (1.3) примут следующий вид (формула (1.4)) и (формула (1.5)).

$$P(SPAM|Word_1, \dots, Word_n) = \frac{P(Word_1|SPAM) * \dots * P(Word_n|SPAM) * P(SPAM)}{P(Word_1) * \dots * P(Word_n)} \tag{1.4}$$

$$P(HAM|Word_1, \dots, Word_n) = \frac{P(Word_1|HAM) * \dots * P(Word_n|HAM) * P(HAM)}{P(Word_1) * \dots * P(Word_n)} \tag{1.5}$$

Для определения принадлежности письма к определенной категории, воспользовавшись формулами (1.4) и (1.5), нужно сравнить апостериорные вероятности. Если получили, что апостериорная вероятность

P(SPAM|WORDS) больше, чем апостериорная вероятность P(HAM|WORDS), то письмо соответственно является спамом (формула (1.6)), а если получили противоположную ситуацию, то – не спам (формула (1.7)). Если P(SPAM|WORDS) равняется P(HAM|WORDS), то однозначного ответа дать не получится (формула (1.8)).

$$P(SPAM|Word_1, \dots, Word_n) > P(HAM|Word_1, \dots, Word_n) \tag{1.6}$$

$$P(SPAM|Word_1, \dots, Word_n) < P(HAM|Word_1, \dots, Word_n) \tag{1.7}$$

$$P(SPAM|Word_1, \dots, Word_n) = P(HAM|Word_1, \dots, Word_n) \tag{1.8}$$

Стоит обратить внимание на вот какой факт. В формулах (1.4) и (1.5) знаменатель (вероятность слова) является константой и никак не может повлиять на ранжирование классов, поэтому можно его отбросить и использовать пропорциональные формулы (1.9) и (1.10):

$$P(SPAM|Word_1, \dots, Word_n) \propto \prod_{i=1}^{n} P(Word_i|SPAM) * P(SPAM) \tag{1.9}$$

$$P(HAM|Word_1, \dots, Word_n) \propto \prod_{i=1}^{n} P(Word_i|HAM) * P(HAM) \tag{1.10}$$

1.5 Проблема арифметического переполнения

При достаточно большой длине документа придется перемножать большое количество очень маленьких чисел. Для того чтобы при этом избежать арифметического переполнения снизу зачастую пользуются свойством логарифма произведения, которое показано в формуле (1.11):

$$\log ab = \log a + \log b \tag{1.11}$$

Так как логарифм функция монотонная, ее применение к обеим частям выражения изменит только его численное значение, но не параметры, при которых достигается максимум. При этом логарифм от числа близкого к нулю будет числом отрицательным, но в абсолютном значении существенно большим чем исходное число, что делает логарифмические значения вероятностей более удобными для анализа. С учетом данного свойства формулы (1.9) и (1.10) примут вид, показанный в формулах (1.12) и (1.13):

$$P(SPAM|Word_1, \dots, Word_n) \propto \sum_{i=1}^{n} \log P(Word_i|SPAM) + \log P(SPAM) \tag{1.12}$$

$$P(HAM|Word_1, \dots, Word_n) \propto \sum_{i=1}^{n} \log P(Word_i|HAM) + \log P(HAM) \tag{1.13}$$

Основание логарифма в данном случае не имеет значения, поэтому можно использовать как натуральный, так и любой другой логарифм.

1.6 Вычисление компонент формулы теоремы Байеса

Для анализа принадлежности сообщения к спаму или не спаму будет использоваться мультиномиальный наивный Байесовский классификатор.

Откуда берутся P(SPAM) и P(WORDS|SPAM)?

Оценка вероятностей P(SPAM) и P(WORDS|SPAM) осуществляется на обучающей выборке. Вероятность класса (для данного примера класса спам) определяется по формуле (1.14):

$$P(SPAM) = \frac{N_{SPAM}}{N} \tag{1.14}$$

где N_{SPAM} – количество спам писем; N – количество всех писем в обучающей выборке.

Оценка вероятности слова в классе может вычисляться несколькими путями, но так как используется мультиномиальный Байесовский классификатор, то вычисления нахождения вероятности слова в классе спам будет проходить по формуле (1.15):

$$P(Word_i|SPAM) = \frac{N_{Word_i \in SPAM}}{N_{WORDS \in SPAM}} \tag{1.15}$$

где $N_{Word_i \in SPAM}$ – количество вхождений слова $Word_i$ в классе SPAM; $N_{WORDS \in SPAM}$ – количество всех слов в классе SPAM.

Другими словами, числитель описывает сколько раз слово (включая повторы) встречается в сообщениях определённого класса, а знаменатель – это суммарное количество слов во всех сообщениях этого определённого класса.

Аналогичным образом вычисляются P(HAM) и P(WORDS|HAM).

1.7 Проблема неизвестных слов. Сглаживание Лапласа

С формулой (1.15) есть одна небольшая проблема. Если на этапе классификации встретится слово, которого не было на этапе обучения, то значения (рассмотрим на примере класса спам) $N_{Word_i \in SPAM}$, а следственно и $P(Word_i|SPAM)$ будут равны нулю. Это приведет к тому, что сообщение с этим словом нельзя будет классифицировать, так как оно будет иметь нулевую вероятность по всем классам. Избавиться от этой проблемы путем анализа большего количества документов не получится, потому что составить обучающую выборку, содержащую все возможные слова, невозможно. Типичным решением проблемы неизвестных слов является аддитивное сглаживание (сглаживание Лапласа). Идея заключается в том, что мы считаем, будто каждое слово встречается на один раз больше, то есть прибавляем единицу к частоте каждого слова. Таким образом вычисление нахождения вероятности слова в классе спам с применением сглаживания Лапласа производится по формуле (1.16):

$$P(Word_i|SPAM) = \frac{N_{Word_i \in SPAM} + 1}{N_{WORDS \in SPAM} + 1 * |WORDS|} \quad (1.16)$$

где $N_{Word_i \in SPAM}$ – количество вхождений слова $Word_i$ в классе SPAM; $N_{WORDS \in SPAM}$ – количество всех слов в классе SPAM; 1 – это принятый нами коэффициент сглаживания, этот коэффициент можно заменить на α: $0<\alpha\leq 1$ (если $\alpha = 1$, то это сглаживание Лапласа); $|WORDS|$ – количество уникальных слов в обучающей выборке.

Логически данный подход смещает оценку вероятностей в сторону менее вероятных исходов. Таким образом слова, которые не попадались на этапе обучения модели, получат пусть маленькую, но все же не нулевую вероятность.

На практике это выглядит так. Допустим на этапе обучения было выделено три слова указанное количество раз (таблица 1.1). А на этапе классификации появилось новое слово Автобус, которого не было на этапе обучения. Тогда оригинальная и смещённая по Лапласу оценка вероятностей будет выглядеть следующим образом (рисунок 1.3).

Таблица 1.1 – Слово - Частота

Слово	Частота
Машина	3
Мопед	2
Грузовик	1

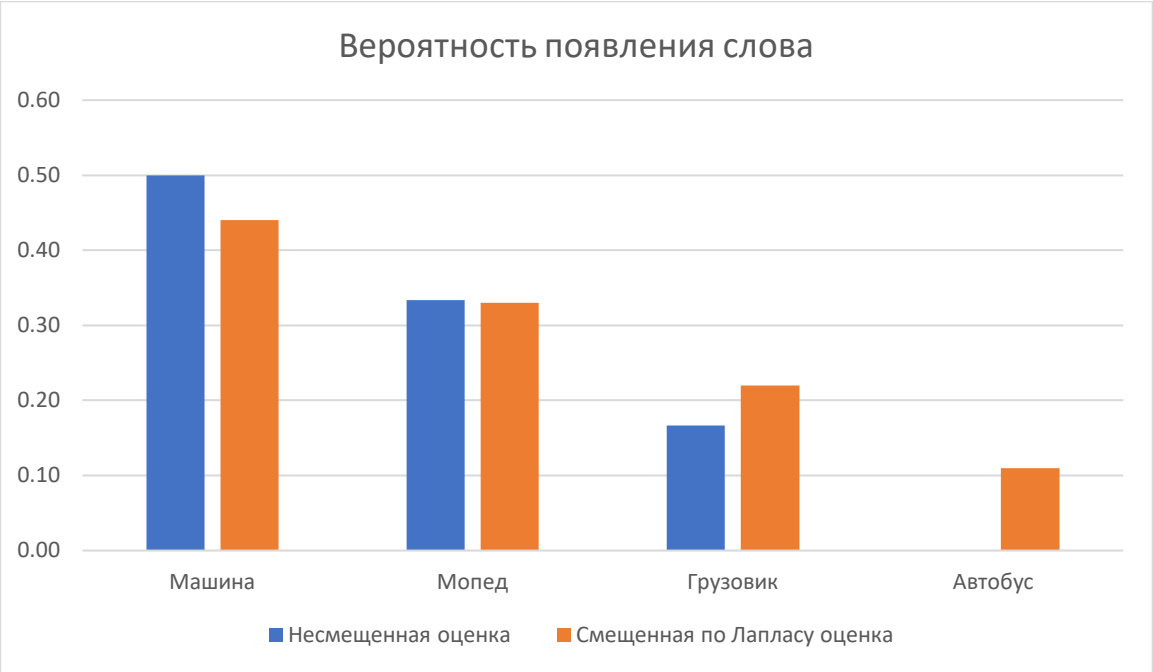

Рисунок 1.3 – Оценка вероятностей (оригинальная и смещенная по Лапласу)

Из рисунка 1.3 видно, что смещённая оценка никогда не бывает нулевой, что защищает от проблемы неизвестных слов.

1.8 Реализация классификатора

Для реализации Байесовского классификатора необходим датасет, в котором проставлены соответствия между сообщениями и их классами. Затем из этого датасета необходимо собрать следующую статистику, которая будет использоваться на этапе классификации:

– относительные частоты классов сообщений, то есть как часто встречаются сообщения того или иного класса;

– суммарное количество слов в документах каждого класса;

– относительные частоты слов в пределах каждого класса;

– размер словаря выборки (количество уникальных слов в выборке).

Совокупность этой информации будет являться моделью классификатора.

Затем на этапе классификации необходимо для каждого класса рассчитать следующие значения по формулам: формула (1.17) определение условной вероятности для спам писем (полная формула со всеми доработками)

и формула (1.18) определение условной вероятности для не спам писем (полная формула со всеми доработками). Затем выбирается класс с максимальным значением условной вероятности.

$$P(SPAM|Word_1, \dots, Word_n) = \log\frac{N_{SPAM}}{N} + \sum_{i=1}^{n} \log\frac{N_{Word_i \in SPAM} + 1}{N_{WORDS \in SPAM} + 1 * |WORDS|} \quad (1.17)$$

$$P(HAM|Word_1, \dots, Word_n) = \log\frac{N_{HAM}}{N} + \sum_{i=1}^{n} \log\frac{N_{Word_i \in HAM} + 1}{N_{WORDS \in HAM} + 1 * |WORDS|} \quad (1.18)$$

Еще раз уточним, что значит каждая буква в формулах (1.17) и (1.18):

– N – количество всех писем;

– N_{SPAM} – количество спам писем;

– N_{HAM} – количество не спам писем;

– $N_{Word_i \in SPAM}$ - количество вхождений слова $Word_i$ в классе SPAM;

– $N_{Word_i \in HAM}$ - количество вхождений слова $Word_i$ в классе HAM;

– $N_{WORDS \in SPAM}$ – количество всех слов в классе SPAM;

– $N_{WORDS \in HAM}$ – количество всех слов в классе HAM;

– $|WORDS|$ – количество уникальных слов в обучающей выборке;

– 1 – это принятый нами коэффициент сглаживания, этот коэффициент можно заменить на α: $0<\alpha\leq1$ (если $\alpha = 1$, то это сглаживание Лапласа).

1.9 Пример работы классификатора

Допустим, у нас есть пять сообщений для которых известны их классы:

1. SPAM: предоставляю услуги бухгалтера бесплатно
2. SPAM: бесплатно подарю остров
3. SPAM: спешите купить автомобиль
4. HAM: надо купить макароны

5. HAM: сегодня будет собрание

Модель классификатора будет выглядеть следующим образом (таблица 1.2 и 1.3).

Таблица 1.2 – Модель классификатора

	SPAM	HAM
Частоты классов	3	2
Суммарное кол-во слов	10	6

Таблица 1.3 – Модель классификатора

	SPAM	HAM
предоставляю	1	0
услуги	1	0
бухгалтера	1	0
бесплатно	2	0
подарю	1	0
остров	1	0
спешите	1	0
купить	1	1
автомобиль	1	0
надо	0	1
макароны	0	1
сегодня	0	1
будет	0	1
собрание	0	1

$|WORDS|$=14

Теперь классифицируем сообщение «надо купить билеты».

Рассчитаем значение выражения для класса SPAM:

$$\ln\frac{3}{5}+\left(\ln\frac{0+1}{10+1*14}+\ln\frac{1+1}{10+1*14}+\ln\frac{0+1}{10+1*14}\right)=-9{,}35$$

Рассчитаем значение выражения для класса HAM:

$$\ln\frac{2}{5}+\left(\ln\frac{1+1}{6+1*14}+\ln\frac{1+1}{6+1*14}+\ln\frac{0+1}{6+1*14}\right)=-8{,}52$$

Получаем, что оценка класса HAM больше, чем класса SPAM. Это значит, что данное сообщение будет отнесено к классу HAM.

Данный пример отражает работу классификатора, пример не отражает предварительную обработку текста, о которой говорилось в пункте 1.3.

1.10 Выбор языка программирования

С поставленной задачей реализовать спам фильтр (Байесовский классификатор) лучше всего справится язык Python. Python – это бесспорный лидер среди языков программирования ИИ. Для это языка существует большое количество различных готовых к использованию библиотек, которые облегчают и ускоряют написание кода.

1.11 Выбор библиотек

Для работы потребуются следующие библиотеки: NLTK, NumPy, Pandas, Multiprocessing, WordCloud.

Библиотека NLTK является одной их лучших библиотек Python для решения задачи обработки естественного языка. Она предоставляет множество полезных функций для обработки текстов, включая токенизацию, выделение корней, синтаксический анализ и многие другие функции, необходимые для создания моделей машинного обучения.

Библиотека NumPy добавляет поддержку больших многомерных массивов и матриц, вместе с большой библиотекой высокоуровневых математических функций для операций с этими массивами.

Библиотека Pandas добавляет удобный функционал для работы с данными, представленными в виде таблиц. Pandas обеспечивает широкий спектр функционала, включая слияние и объединение данных, фильтрацию, группировку, агрегацию, обработку пропущенных значений, визуализацию данных и многое другое.

Библиотека Multiprocessing позволяет распараллеливать выполнение задач, что ускоряет обработку данных и увеличивает производительность программы при работе с большими объемами данных.

Библиотека WordCloud предоставляет удобный интерфейс для построения графика облаков слов.

1.12 Данные для обучения модели

Точность работы модели напрямую зависит от качества данных для обучения. Лучшие команды мира, тратят достаточно много времени на улучшение своих тренировочных данных. Нередко компании предоставляют код своих моделей машинного обучения в открытый доступ, но данные, на которых модели обучались, часто остаются в тайне. В интернете есть довольно много различных датасетов, но не всегда они подходят на все 100%.

В работе используются несколько датасетов, который были взяты с сайта www.kaggle.com. Датасет «Spam Email Classification Dataset»[1] содержит 83446 email сообщений, где 52,6% относятся к спам сообщениям, а 47,4% - совершенно легальных сообщения. Датасет «Spam Mails Dataset»[2] содержит 5171 email сообщений, где 29% относятся к спам сообщениям, а 71% - совершенно легальных сообщения. Датасет «Spam email Dataset»[3] содержит

[1] Spam Email Classification Dataset: сайт. – URL: https://www.kaggle.com/datasets/purusinghvi/email-spam-classification-dataset (дата обращения: 07.10.2023)

[2] Spam Mails Dataset: сайт. – URL: https://www.kaggle.com/datasets/venky73/spam-mails-dataset/data (дата обращения: 07.10.2023)

[3] Spam email Dataset: сайт. – URL: https://www.kaggle.com/datasets/jackksoncsie/spam-email-dataset (дата обращения: 07.10.2023)

5728 email сообщений, где 24% относятся к спам сообщениям, а 76% - совершенно легальных сообщения.

ГЛАВА 2. ПРАКТИЧЕСКАЯ РЕАЛИЗАЦИЯ.

2.1 Структура проекта

В проекте реализовано три класса: TextProcessor, TrainMultinomialNB и SpamDetector.

2.2 Класс TextProcessor

Класс TextProcessor используется для обработки и токенизации текста. В данном классе есть только один публичный метод, представляющий собой пайплайн обработки текста. Этот пайплайн включает в себя следующие шаги: сначала из переданного текста удаляются все не ASCII символы, затем текст приводится к нижнему регистру и удаляются короткие слова, после происходит удаление стоп-слов, после этого применяется стемминг и наконец происходит токенизация.

При инициализации объекта можно повлиять на пайплайн. Можно задать минимальную длину слова, которое будет считаться коротким (по умолчанию это 2; можно установить любое значение большее или равное 0). Также к списку стоп-слов, которые берутся из библиотек NLTK и WordCloud, можно добавить пользовательские слова, тем самым увеличив множество стоп-слов, которые будут удаляться из текста.

Класс TextProcessor обладает высокой универсальностью, его можно использовать везде, где необходим реализованный в нем пайплайн. Непосредственно для данной задачи он используется в двух других классах: TrainMultinomialNB и SpamDetector.

2.3 Класс TrainMultinomialNB

Перед использованием классификатора его нужно сначала натренировать, для этого и предназначен класс TrainMultinomialNB. Результат тренировки это два csv файла. Первый такой файл хранит данные об отношении спам и не спам писем к их общему числу, то есть там хранятся

вероятности, что полученное письмо окажется спамом или не спамом. Пример такого csv файла показан на рисунке 2.1.

```
1   ,ham,spam
2   ratios-to-total-emails,0.4738040456332087,0.5261959543667913
3
```

Рисунок 2.1 – Данные о вероятности получения не спам и спам письма

Второй csv файл содержит частоты слов, встречающиеся в письмах. Структура этого csv следующая: «,frequency,frequency_ham,frequency_spam». В качестве индексов выступают слова. Колонки frequency – это частота появления слова в любом письме, frequency_ham – это частота появления слова в не спам письме, а frequency_spam — это частота появления слова в спам письме. Пример csv файла с частотами слов показан на рисунке 2.2.

```
1   ,frequency,frequency_ham,frequency_spam
2   escapenumb,1149940,809995,339945
3   escapelong,227296,39137,188159
4   use,46972,34745,12227
5   org,43603,42569,1034
6   list,40709,35320,5389
7   help,40603,35172,5431
8   pleas,40603,27227,13376
9   mail,39850,30499,9351
10  new,37142,25739,11403
```

Рисунок 2.2 – Данные о частотах слов писем в обучающем датасете

Для повышения точности классификатора нужно искать большие датасеты электронных писем. Большой датасет поспособствует увеличению числа уникальных слов – это позволить классифицировать письма различных тематик, а также из большего датасета удастся выявить больше данных о частотах появления слов в той или иной категории писем – это позволит с большей уверенностью классифицировать письма.

При инициализации данного класса в него должен передаваться путь до подготовленного датасета с письмами. В данном случае, подготовленный

датасет – это датасет, в котором удалены лишние столбцы и присутствуют только столбцы «,text,label», где text – это текст письма, а label это его категория. Категория 1 – это спам, категория 0 – это не спам. Пример подготовленного датасета показан на рисунке 2.3.

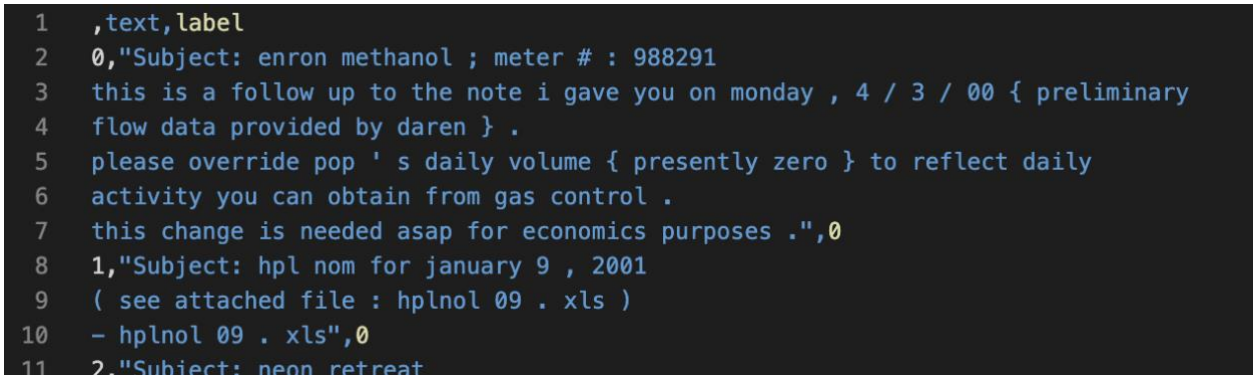

Рисунок 2.3 – Пример датасета подготовленного для обучения

Повлиять на процесс обучения можно, через методы класса. Перед началом обучения с помощью метода set_min_word_frequency можно установить минимальную частоту слова. По умолчанию это значение 15. Это значит, что если конкретное слово в процессе обучения встретилось меньше 15, то оно не будут добавлено в csv файл с частотами слов. С помощью метода set_processor можно повлиять на инициализацию объекта класса TextProcessor, передав желаемые параметры обработчика. Это делается также перед началом обучения.

Обучение модели начинается с помощью вызова метода train, в который передается путь до папки, где будет сохранены два результирующих csv файла. Этот параметр является опциональным, и если его не передать, то результат работы будет сохранен рядом с файлом датасета с письмами.

Стоит отметить приватный метод __prepare_text, который реализован с использованием технологии распараллеливания. Распараллеливание производится на все ядра процессора. Использование данной технологии существенно сокращает время обучения модели.

2.4 Класс SpamDetector

Класс SpamDetector используется для классификации писем. Для инициализации объекта данного класса нужно передать пути до двух csv файлов, полученных после обучения модели.

Повлиять на классификатор можно с помощью методов класса. С помощью метода set_smoothing_factor, может поменять значение сглаживающего фактора на любое другое, лежащие в диапазоне от 0 до 1 включительно. По умолчанию значение сглаживающего фактора установлено в 1, что советует сглаживанию Лапласа. Также с помощью метода set_processor можно установить параметры для инициализации объекта класса TextProcessor.

Для определения принадлежности письма к какому-либо классу используется метод detecting_spam, реализующий мультиномильный наивный Байесовский классификатор. Данный метод принимает в качестве аргумента какое-то сообщение, которое необходимо классифицировать. Результат работы метода – это категория класса, к которому относится сообщение: 1 – спам, 0 – не спам.

2.5 Анализ датасета Spam Mails Dataset

Важными данными для решения задачи классификации с применением мультиномиального наивного Байесовского классификатора является вероятность получить не спам или спам письмо, и частоты появления слов в той или иной категории письма.

Человеку проще воспринимать графическую информацию, поэтому с помощью библиотек Python визуализируем ключевую информацию.

На рисунке 2.4 показана круговая диаграмма, демонстрирующая отношение не спам и спам писем к общему числу сообщений в датасете.

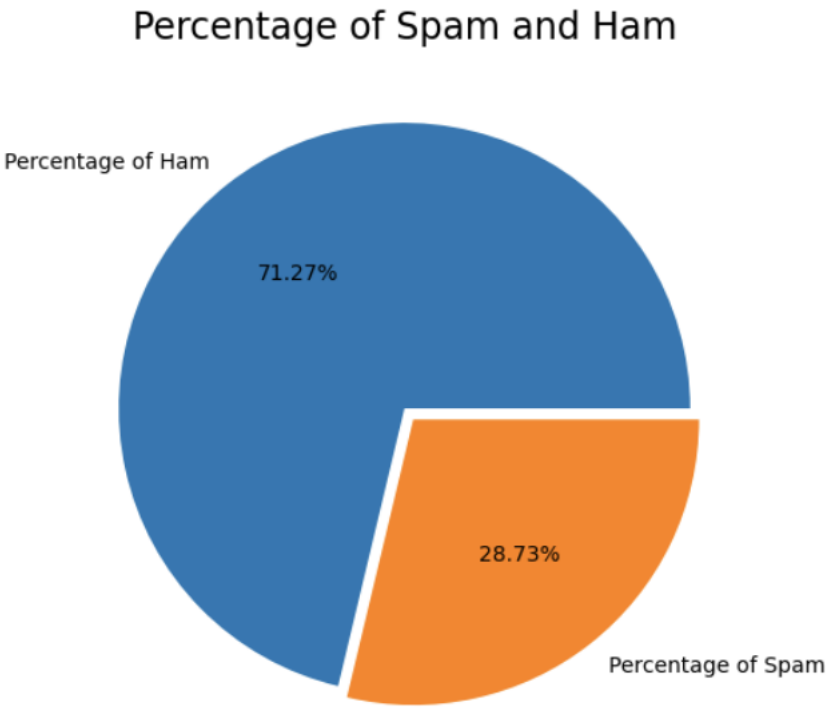

Рисунок 2.4 – Диаграмма отношения не спама и спама к общему числу писем

С помощью библиотеки WordCloud удобно визуализировать текстовую информацию. Например, на рисунке 2.5 показано множество стоп-слов, которые исключаются из письма.

Рисунок 2.5 – WordCloud для стоп-слов

Одна из важных особенностей WordCloud – размер слов на графике. Чем слово больше, там чаще оно встречается. Такой тип визуализации позволяет быстро выделить ключевые термины или темы, которые часто встречаются в тексте. Если для wordcloud стоп-слов, это так не актуально, потому что там слова встречаются по одну разу, то для текстовых сообщений – это очень удобная особенность, наглядно показывающая частоту появления слов. На рисунках 2.6 и 2.7 показаны облака слов для спам тем и спам писем, а на рисунках 2.8 и 2.9 – для не спам тем и не спам писем.

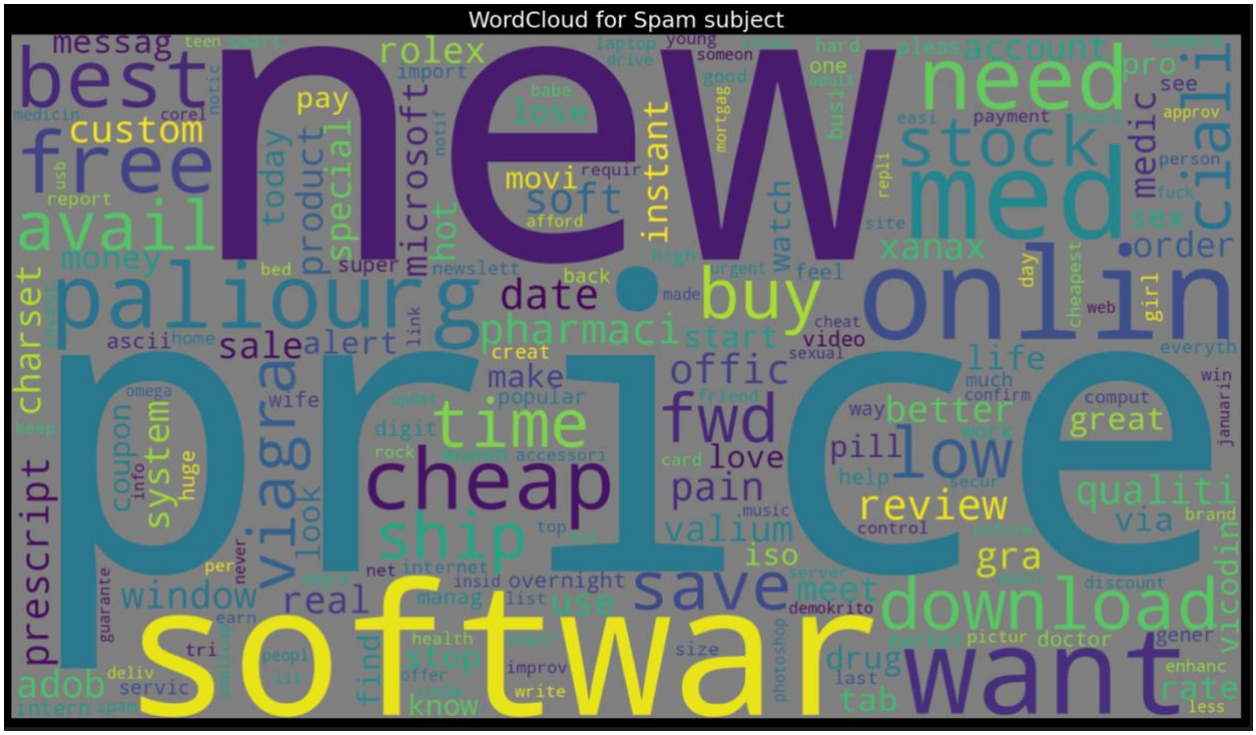

Рисунок 2.6 – WordCloud для Spam subject

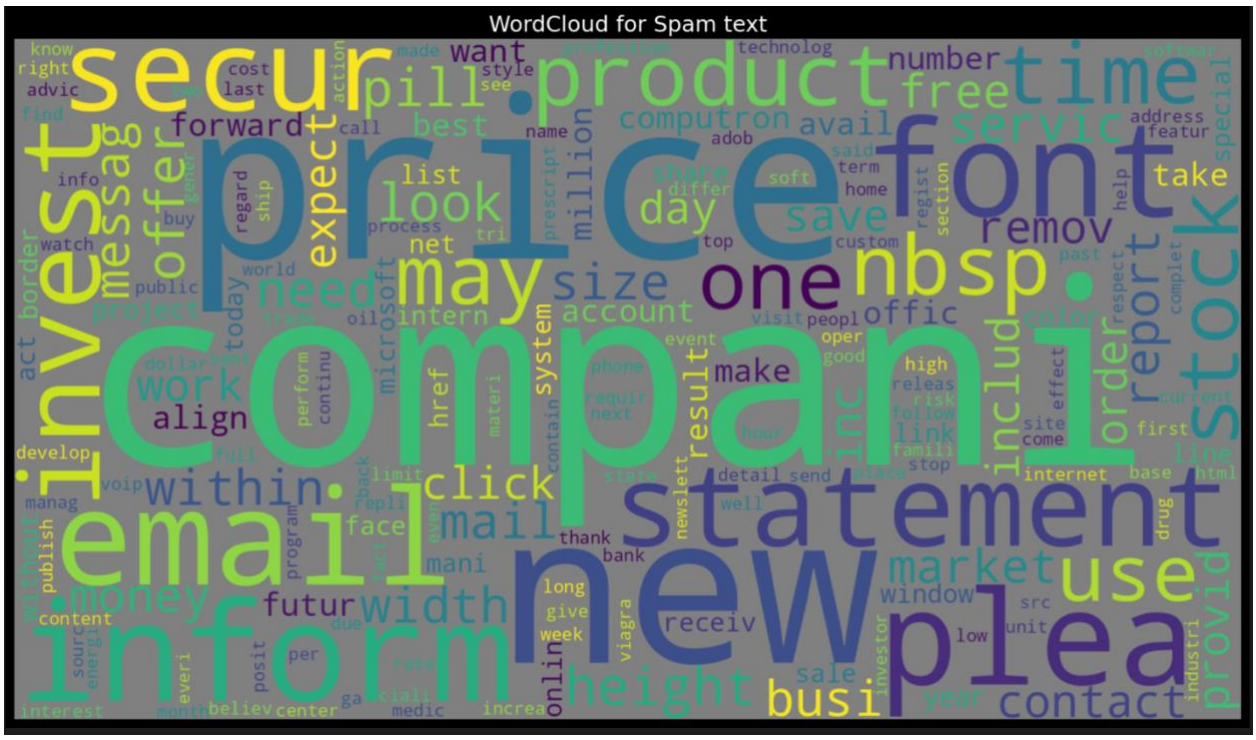

Рисунок 2.7 – WordCloud для Spam text

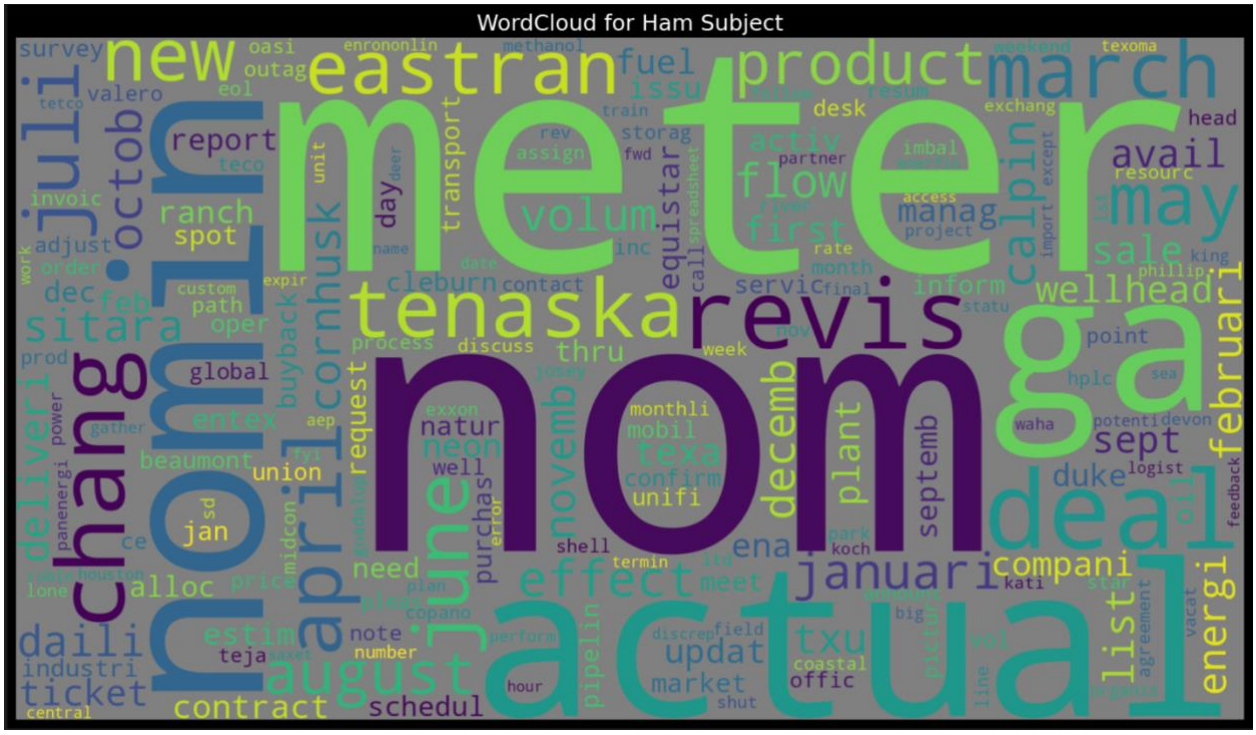

Рисунок 2.8 – WordCloud для Ham subject

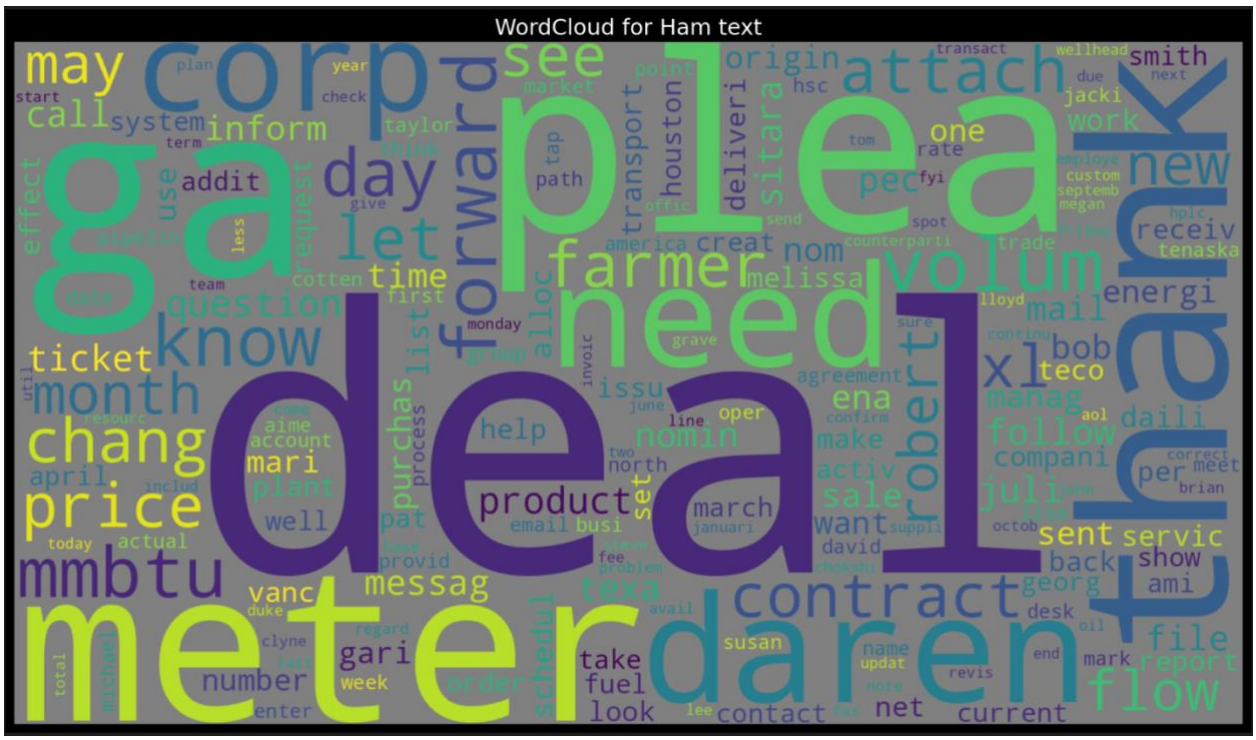

Рисунок 2.9 – WordCloud для Ham text

Также можно строить другой график частот слов, показанный на рисунке 2.10. Здесь он представлен только для первых 50 наиболее встречающихся слов, но его можно построить и для большего количества слова. Такой график, можно использовать при очистке данных от шумовых слов, а также, как и в случае с облаками слов, для выявления ключевых терминов или тем.

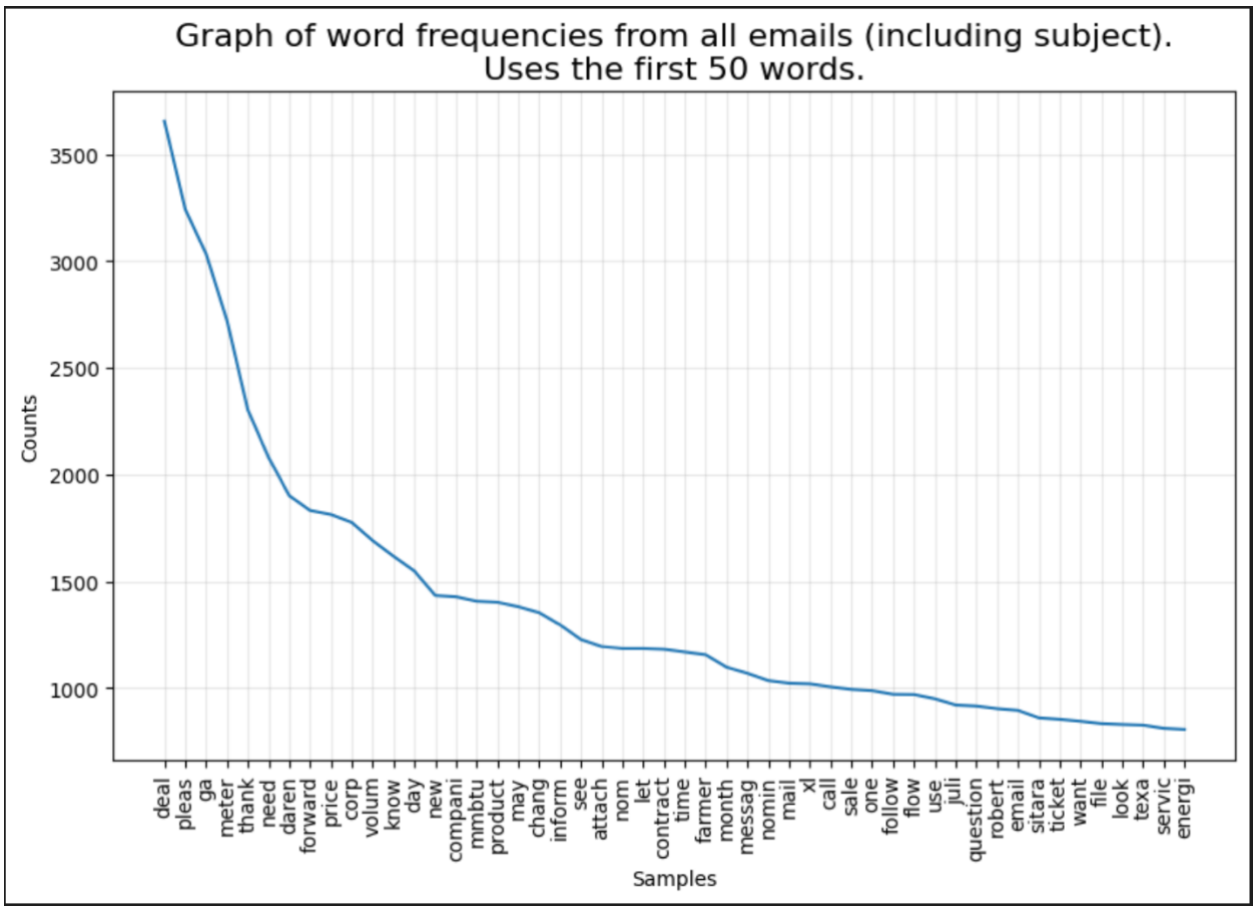

Рисунок 2.10 – График частоты 50 наиболее встречающихся слов в датасете

2.6 Обучение модели

Для обучения модели будет использовать самый большой датасет писем «Spam Email Classification Dataset», это позволит добиться большей точности классификации. Результат запуска обучения модели показан на рисунке 2.11.

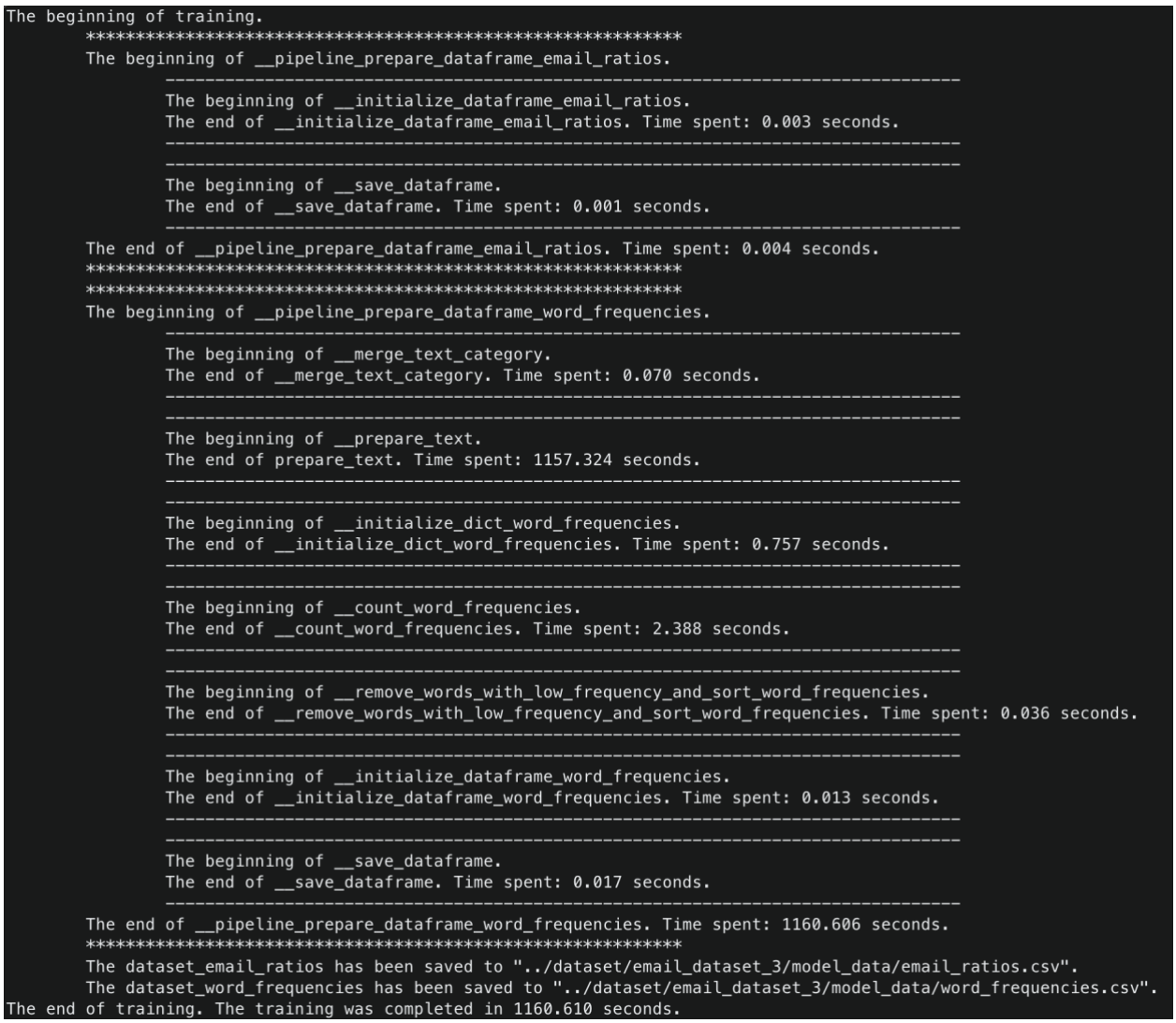

Рисунок 2.11 – Результат обучения модели

2.7 Тестирование модели

Тестирование будет проводиться на двух датасетах «Spam Mails Dataset» и «Spam email Dataset». Эти датасеты не использовались в обучении. Для тестирования модели будут использоваться три классификатора, у которых один и тот же набор данных частот слов, полученный при обучении модели на датасете «Spam Email Classification Dataset», но разные наборы данных вероятностей получения не спам или спам письма. Визуализация этих вероятностей показана на рисунке 2.12.

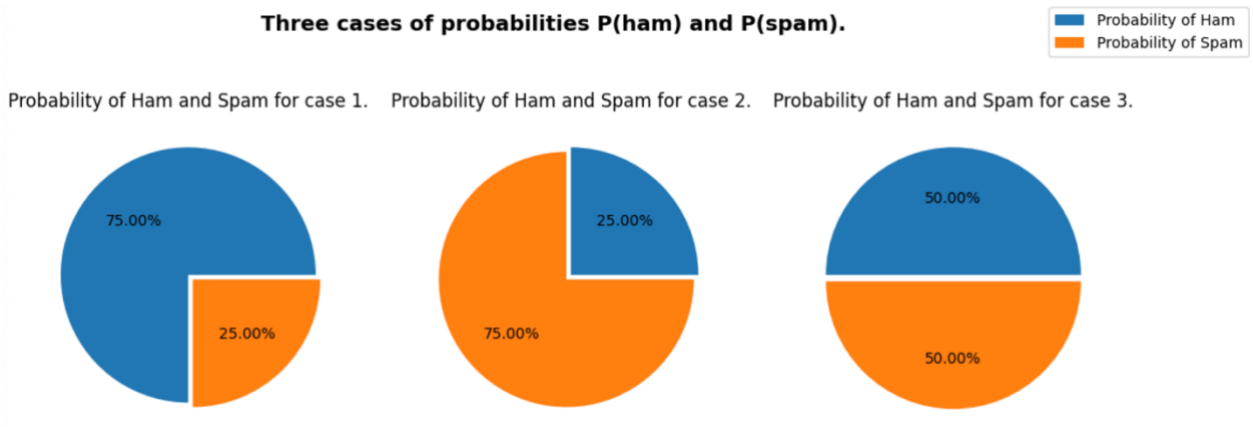

Рисунок 2.12 – Три случая вероятностей получения не спам или спам письма

Матрица ошибок, полученная при тестировании модели на датасете «Spam Mails Dataset», показана на рисунке 2.13. Метрики качества модели показаны на рисунках 2.14–2.16.

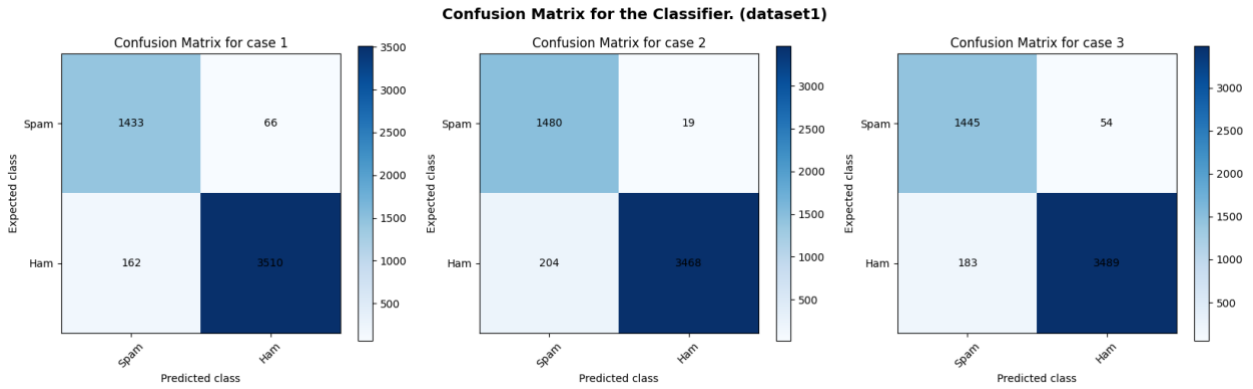

Рисунок 2.13 – Матрица ошибок для трех классификаторов (тест на датасете «Spam Mails Dataset»)

```
Accuracy for MultinomialNB_case1 = 0.9559079481725005
Precision for MultinomialNB_case1 = 0.9559706470980653
Recall for MultinomialNB_case1 = 0.8984326018808777
F1 score for MultinomialNB_case1 = 0.9263089851325145
```

Рисунок 2.14 – Метрики качества модели для классификатора 1 (тест на датасете «Spam Mails Dataset»)

```
Accuracy for MultinomialNB_case2 = 0.9568748791336299
Precision for MultinomialNB_case2 = 0.9873248832555037
Recall for MultinomialNB_case2 = 0.8788598574821853
F1 score for MultinomialNB_case2 = 0.9299403078856425
```

Рисунок 2.15 – Метрики качества модели для классификатора 2 (тест на датасете «Spam Mails Dataset»)

```
Accuracy for MultinomialNB_case3 = 0.9541674724424676
Precision for MultinomialNB_case3 = 0.9639759839893263
Recall for MultinomialNB_case3 = 0.8875921375921376
F1 score for MultinomialNB_case3 = 0.9242085065558043
```

Рисунок 2.16 – Метрики качества модели для классификатора 3 (тест на датасете «Spam Mails Dataset»)

Матрица ошибок, полученная при тестировании модели на датасете «Spam email Dataset», показана на рисунке 2.17. Метрики качества модели показаны на рисунках 2.18–2.20.

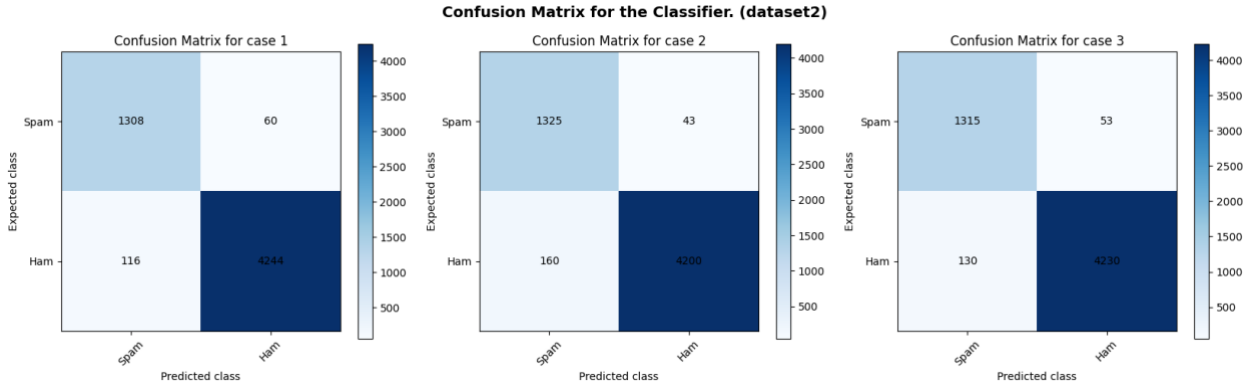

Рисунок 2.17 – Матрица ошибок для трех классификаторов (тест на датасете «Spam email Dataset»)

```
Accuracy for MultinomialNB_case1 = 0.9692737430167597
Precision for MultinomialNB_case1 = 0.956140350877193
Recall for MultinomialNB_case1 = 0.9185393258426966
F1 score for MultinomialNB_case1 = 0.9369627507163324
```

Рисунок 2.18 – Метрики качества модели для классификатора 1 (тест на датасете «Spam email Dataset»)

```
Accuracy for MultinomialNB_case2 = 0.9645600558659218
Precision for MultinomialNB_case2 = 0.9685672514619883
Recall for MultinomialNB_case2 = 0.8922558922558923
F1 score for MultinomialNB_case2 = 0.9288468279004557
```

Рисунок 2.19 – Метрики качества модели для классификатора 2 (тест на датасете «Spam email Dataset»)

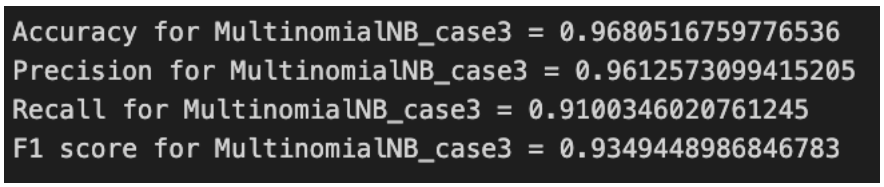

Рисунок 2.20 – Метрики качества модели для классификатора 3 (тест на датасете «Spam email Dataset»)

По данным метрикам видно, что использовать вероятность получения не спама или спама в случае 2 уменьшает точность работы модели, а при использовании вероятностей из случаев 1 и 3 точность примерно равна.

В целом метрика F1 score находятся на уровне 92%–93%, что является достаточно хорошей точностью работы модели. Также из датасета «Spam Mails Dataset» видно, что можно поискать решения по улучшению метрики Recall, что дает некоторый потенциал для роста точности работы модели.

ЗАКЛЮЧЕНИЕ

На сегодняшний день проблема спама не потеряла своей актуальности. Мультиномиальный наивный Байесовский классификатор достаточно прост в реализации и обладает высокой точностью работы, что является хорошим решением для данной проблемы. Результаты тестирования подтверждают ее высокую точность работы, а также свидетельствуют о наличии потенциала для роста точности.

Рост точности можно добиться путем улучшения метрики Recall.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Современные спам фильтры и End-to-End шифрование [электронный ресурс] – URL https://habr.com/ru/articles/237745/ (05.10.2023)

2. Фатхутдинова Д. Спам-фильтр [электронный ресурс] – URL https://www.unisender.com/ru/glossary/chto-takoe-spam-filtry/#anchor-1 (05.10.2023)

3. Вергелис, Щербакова, Сидорина, Куликова Спам и фишинг в 2019 году [электронный ресурс] – URL https://securelist.ru/spam-report-2019/95727/ (05.10.2023)

4. Наивный байесовский классификатор [электронный ресурс] – URL https://www.bazhenov.me/blog/2012/06/11/naive-bayes.html (07.10.2023)

5. Метрики в задачах машинного обучения [электронный ресурс] – URL https://habr.com/ru/companies/ods/articles/328372/ (08.10.2023)

Приложение

SpamDetector.py

```
import math

import pandas as pd

import TextProcessor


class SpamDetector:
    """
    A class for detecting spam in email messages using a Multinomial
Naive Bayes classifier.

    Class Attributes:
        __processor (TextProcessor): An instance of TextProcessor is used
for text processing and tokenization.
            Default minimum word length is 2. Should be greater than or
equal to 0.
            Default includes stop words from the nltk and wordcloud
libraries.
        __smoothing_factor (float): The smoothing factor.
            When equal to 1, corresponds to Laplace smoothing.
            Default is 1.0. Should be in the range 0 < __smoothing_factor
<= 1.

    Attributes:
        __word_frequencies (pd.DataFrame): The DataFrame of word
frequencies.
        __count_unique_words (int): The count of unique words; sourced
from DataFrame word frequencies.
        __count_words_ham (int): The count of words in ham emails;
sourced from DataFrame word frequencies.
        __count_words_spam (int): The count of words in spam emails;
sourced from DataFrame word frequencies.
        __ham_probability (float): The probability of ham emails; sourced
from DataFrame email_ratios.
        __spam_probability (float): The probability of spam emails;
sourced from DataFrame email_ratios.

    Note:
        Smoothing is used to handle the issue of zero probabilities in
the Multinomial Naive Bayes classifier.
    """

    __processor = TextProcessor.TextProcessor()
    __smoothing_factor = 1.0

    def __init__(self, path_to_dataset_word_frequencies: str,
path_to_dataset_email_ratios: str) -> None:
        """
        Initializes a SpamDetector object.

        Parameters:
            path_to_dataset_word_frequencies (str): The file path to the
DataFrame file of word frequencies.
                The DataFrame must have the following structure:
                    Columns should be labeled as follows: frequency,
frequency_ham, frequency_spam.
                    Indexes should be made up of words.
```

```
                Example:
                    ,frequency,frequency_ham,frequency_spam
                    deal,3655,3549,106
                    pleas,3243,2737,506
                    ga,3034,2861,173
                    meter,2721,2718,3
                    thank,2304,2125,179

            path_to_dataset_email_ratios (str): The file path to the
DataFrame file containing the ratios of ham and spam emails to total
emails.
                The DataFrame must have the following structure:
                    Columns should be labeled as follows: ham, spam.
                    Index should be only one and named: ratios-to-total-
emails.
                Example:
                    ,ham,spam
                    ratios-to-total-
emails,0.7127329192546584,0.28726708074534163

        Raises:
            FileNotFoundError: If the specified DataFrame file of word
frequencies or email relationships is not found.
            AttributeError: If the DataFrame file of word frequencies or
email ratios has an incorrect structure.

        Returns:
            None
        """

        try:
            self.__word_frequencies =
pd.read_csv(path_to_dataset_word_frequencies, index_col=0)

            self.__count_unique_words =
len(self.__word_frequencies.frequency) # N(Unique Words)
            self.__count_words_ham =
sum(self.__word_frequencies.frequency_ham) # N(Words ∈ ham)
            self.__count_words_spam =
sum(self.__word_frequencies.frequency_spam) # N(Words ∈ spam)

        except FileNotFoundError:
            raise FileNotFoundError(
                f'\n\tError: The DataFrame file of word frequencies is
not found at "{path_to_dataset_word_frequencies}".\n'
                f"\tThe DataFrame must have the following structure:\n"
                f"\t\tColumns should be labeled as follows: frequency,
frequency_ham, frequency_spam.\n"
                f"\t\tIndexes should be made up of words.\n\n"
                f"\tExample:\n"
                f"\t\t,frequency,frequency_ham,frequency_spam\n"
                f"\t\tdeal,3655,3549,106\n"
                f"\t\tpleas,3243,2737,506\n"
                f"\t\tga,3034,2861,173\n"
                f"\t\tmeter,2721,2718,3\n"
                f"\t\tthank,2304,2125,179\n"
            )
        except AttributeError:
            raise AttributeError(
                f"\n\tError: The DataFrame file of word frequencies has
an incorrect structure.\n"
                f"\tThe DataFrame must have the following structure:\n"
```

```
                f"\t\tColumns should be labeled as follows: frequency,
frequency_ham, frequency_spam.\n"
                f"\t\tIndexes should be made up of words.\n\n"
                f"\tExample:\n"
                f"\t\t,frequency,frequency_ham,frequency_spam\n"
                f"\t\tdeal,3655,3549,106\n"
                f"\t\tpleas,3243,2737,506\n"
                f"\t\tga,3034,2861,173\n"
                f"\t\tmeter,2721,2718,3\n"
                f"\t\tthank,2304,2125,179\n"
            )

        try:
            email_ratios = pd.read_csv(path_to_dataset_email_ratios,
index_col=0)

            self.__ham_probability = email_ratios.ham['ratios-to-total-
emails'] # P(ham)
            self.__spam_probability = email_ratios.spam['ratios-to-total-
emails'] # P(spam)

        except FileNotFoundError:
            raise FileNotFoundError(
                f'\n\tError: The DataFrame file containing the ratios of
ham and spam emails to total emails is not found at
"{path_to_dataset_email_ratios}".\n'
                f"\tThe DataFrame must have the following structure:\n"
                f"\t\tColumns should be labeled as follows: ham, spam.\n"
                f"\t\tIndex should be only one and named: ratios-to-
total-emails.\n\n"
                f"\tExample:\n"
                f"\t\t,ham,spam\n"
                f"\t\tratios-to-total-
emails,0.7127329192546584,0.28726708074534163\n"
            )
        except (AttributeError, KeyError):
            raise AttributeError(
                f"\n\tError: The DataFrame file containing the ratios of
ham and spam emails to total emails has an incorrect structure.\n"
                f"\tThe DataFrame must have the following structure:\n"
                f"\t\tColumns should be labeled as follows: ham, spam.\n"
                f"\t\tIndex should be only one and named: ratios-to-
total-emails.\n\n"
                f"\tExample:\n"
                f"\t\t,ham,spam\n"
                f"\t\tratios-to-total-
emails,0.7127329192546584,0.28726708074534163\n"
            )

    def detecting_spam(self, message: str) -> int:
        """
        Detects spam in a message.

        Parameters:
            message (str): The string containing the message text.

        Returns:
            int: Returns 1 if the message is considered spam, and 0 if it
is ham.

        Formula:
```

$$\text{(№1)}\quad P(\text{ham} \mid \text{Word}_i) = \log(P(\text{ham})) + \sum_{i=1}^{n} \log(P(\text{Word}_i \mid \text{ham}))$$

```
                                                                n
            (№2) P(spam | Word_i) = log(P(spam)) + ∑ log(P(Word_i |
spam))
                                                               i=1

        Where:
            log(P(ham)): The logarithm of the probability of receiving
ham email.
                                Calculated using the method
"__log_probability_ham_email".

            log(P(spam)): The logarithm of the probability of receiving
spam email.
                                 Calculated using the method
"__log_probability_spam_email".

             n
             ∑ log(P(Word_i | ham)): Sum of log probabilities of words in
the ham category from the given message.
            i=1                          Calculated using the method
"__sum_log_prob_words_ham".

             n
             ∑ log(P(Word_i | spam)): Sum of log probabilities of words
in the spam category from the given message.
            i=1                           Calculated using the method
"__sum_log_prob_words_spam".
        """

        prepared_message = self.__processor.pipeline(message)

        ham_prob_given_word_i =
self.__log_probability_ham_email(self.__ham_probability) +
self.__sum_log_prob_words_ham(prepared_message)
        spam_prob_given_word_i =
self.__log_probability_spam_email(self.__spam_probability) +
self.__sum_log_prob_words_spam(prepared_message)

        return int(spam_prob_given_word_i > ham_prob_given_word_i)

    def __log_probability_ham_email(self, ham_email_probability: float) -
> float:
        """
        Calculate the log probability of receiving ham email.

        Parameters:
            ham_email_probability (float): The probability of receiving
ham email.

        Returns:
            float: The logarithm of ham email probability.

        Formula:
                                  N ∈ ham
            log(P(ham)) = log(---------) = log(ham_email_probability)
                                    N

        Where:
            N ∈ ham: Count of ham emails.
            N: Count of all emails.

        Note:
```

```
            The ham_email_probability is used; the formula simply
underscores information on how this variable was obtained.
        """

        return math.log(ham_email_probability)

    def __log_probability_spam_email(self, spam_email_probability: float)
-> float:
        """
        Calculate the log probability of receiving spam email.

        Parameters:
            spam_email_probability (float): The probability of receiving
spam email.

        Returns:
            float: The logarithm of spam email probability.

        Formula:
                                  N ∈ spam
            log(P(spam)) = log(----------) = log(spam_email_probability)
                                    N

        Where:
            N ∈ spam: Count of spam emails.
            N: Count of all emails.

        Note:
            The spam_email_probability is used; the formula simply
underscores information on how this variable was obtained.
        """

        return math.log(spam_email_probability)

    def __sum_log_prob_words_ham(self, prepared_message: list[str]) ->
float:
        """
        Calculate the sum of log probabilities of words from a given
message, given that they are in the ham category.

        Parameters:
            prepared_message (list[str]): The list of processed words in
the message.

        Returns:
            float: The sum of log probabilities of words in the ham
category from the given message.

        Formula:
            n                          n              N(Word_i ∈ ham) +
smoothing_factor
            ∑ log(P(Word_i | ham)) = ∑ log(------------------------------
--------------------)
          i=1                        i=1     N(Words ∈ ham) +
smoothing_factor*N(Unique Words)

        Where:
            N(Word_i ∈ ham): Count of occurrences of the Word_i in ham
emails.
            N(Words ∈ ham): Total count of all Words that appeared in ham
emails.
            N(Unique Words): Count of all unique words.
            smoothing_factor: Smoothing factor to avoid zero.
```

```
        """

        sum_log_prob_word_i_ham = 0
        numerator = self.__count_words_ham +
self.__smoothing_factor*self.__count_unique_words

        sum_log_prob_word_i_ham =
sum([math.log((self.__word_frequencies.frequency_ham.get(word_i,
0)+self.__smoothing_factor) / numerator) for word_i in prepared_message])

        return sum_log_prob_word_i_ham

    def __sum_log_prob_words_spam(self, prepared_message: list[str]) ->
float:
        """
        Calculate the sum of log probabilities of words from a given
message, given that they are in the spam category.

        Parameters:
            prepared_message (list[str]): The list of processed words in
the message.

        Returns:
            float: The sum of log probabilities of words in the spam
category from the given message.

        Formula:
            n                              n              N(Word_i ∈ spam) +
smoothing_factor
            ∑ log(P(Word_i | spam)) = ∑ log(-----------------------------
----------------------)
           i=1                            i=1     N(Words ∈ spam) +
smoothing_factor*N(Unique Words)

        Where:
            N(Word_i ∈ spam): Count of occurrences of the Word_i in spam
emails.
            N(Words ∈ spam): Total count of all Words that appeared in
spam emails.
            N(Unique Words): Count of all unique words.
            smoothing_factor: Smoothing factor to avoid zero.
        """

        sum_log_prob_word_i_spam = 0
        numerator = self.__count_words_spam +
self.__smoothing_factor*self.__count_unique_words

        sum_log_prob_word_i_spam =
sum([math.log((self.__word_frequencies.frequency_spam.get(word_i,
0)+self.__smoothing_factor) / numerator) for word_i in prepared_message])

        return sum_log_prob_word_i_spam

    @classmethod
    def set_processor(cls, min_length: int = None, user_stop_words:
list[str] | tuple[str] | set[str] = None) -> None:
        """
        Set parameters for an instance of TextProcessor to be used for
text processing and tokenization.

        Parameters:
            min_length (int, optional): Minimum word length.
                Default is 2. Should be greater than or equal to 0.
```

```
            user_stop_words (list[str] | tuple[str] | set[str],
optional): The user-defined list of stop words. This list is added to the
stop words defined by default.
                Default includes stop words from the nltk and wordcloud
libraries.
                Example:
                    user_stop_words = ['ect', 'enron', 'hou', 'hpl',
'subject']

        Returns:
            None
        """

        cls.__processor = TextProcessor.TextProcessor(min_length,
user_stop_words)

    @classmethod
    def set_smoothing_factor(cls, smoothing_factor: float) -> None:
        """
        Set the smoothing factor for the class.

        Parameters:
            smoothing_factor (float): The smoothing factor to be set.
Should be in the range 0 < smoothing_factor <= 1.

        Raises:
            ValueError: If the smoothing_factor is not in the valid
range.

        Returns:
            None
        """

        if 0 < smoothing_factor and smoothing_factor <= 1:
            cls.__smoothing_factor = smoothing_factor
        else:
            raise ValueError(f"Error: Smoothing factor should be in the
range 0 < smoothing_factor <= 1. Got: {smoothing_factor}")


if __name__ == "__main__":

    path_to_dataset_word_frequencies =
'../dataset/email_dataset_1/model_data/word_frequencies.csv'
    path_to_dataset_email_ratios =
'../dataset/email_dataset_1/model_data/email_ratios.csv'

    detector = SpamDetector(path_to_dataset_word_frequencies,
path_to_dataset_email_ratios)
    detector.set_processor(user_stop_words=['ect', 'enron', 'hou', 'hpl',
'subject'])

    # Some examples
    emails =
pd.read_csv('../dataset/email_dataset_1/spam_ham_dataset.csv')
    for i in range(1, 4):
        print(f"Email №{i}\n")
        print(f"{emails.text[i*10]}")
        print(f"\nThe algorithm determined that the email...
{detector.detecting_spam(emails.text[i*10])}")
        print(f"It was expected that the email...
{emails.label_num[i*10]}")
```

```
print(f"**********************************************************************
**************\n")
```

TextProcessor.py

```
import string

import nltk
import wordcloud
from nltk.stem.porter import PorterStemmer # We have various stemmers;
let's utilize them.


class TextProcessor:
    """
    Text processing pipeline class.

    Attributes:
        __len_min_ch (int): Minimum word length.
            Default is 2. Should be greater than or equal to 0.
        __stop_words (set[str]): Set of stop words.
            Default includes stop words from the nltk and wordcloud
libraries.
    """

    def __init__(self, min_length: int = None, user_stop_words: list[str]
| tuple[str] | set[str] = None) -> None:
        """
        Initializes the TextProcessor object.

        Parameters:
            min_length (int, optional): Minimum word length.
                Default is 2. Should be greater than or equal to 0.
            user_stop_words (list[str] | tuple[str] | set[str],
optional): The user-defined list of stop words. This list is added to the
stop words defined by default.
                Default includes stop words from the nltk and wordcloud
libraries.
                Example:
                    user_stop_words = ['ect', 'enron', 'hou', 'hpl',
'subject']

        Raises:
            ValueError: If the provided min_length is less than 0.
            ValueError: If the provided user_stop_words is not a list,
tuple, or set.
            ValueError: If the provided user_stop_words contains anything
other than strings.

        Returns:
            None
        """

        if min_length is None:
            self.__len_min_ch = 2
        else:
            if min_length >= 0:
                self.__len_min_ch = min_length
            else:
                raise ValueError(f"Error: min_length should be greater
than or equal to 0. Got: {min_length}")
```

```
        stop_words_n1 = set(nltk.corpus.stopwords.words('english'))
        stop_words_n2 = set(wordcloud.STOPWORDS)

        if user_stop_words is None:
            self.__stop_words = set.union(stop_words_n1, stop_words_n2)
        else:
            if isinstance(user_stop_words, (list, tuple, set)):
                if all(isinstance(word, str) for word in
user_stop_words):
                    user_stop_words = set(user_stop_words)
                    self.__stop_words = set.union(stop_words_n1,
stop_words_n2, user_stop_words)
                else:
                    raise ValueError("Error: user_stop_words should
contain only strings.")
            else:
                raise ValueError(f"Error: user_stop_words should be a
list, tuple, or set. Got {type(user_stop_words)}")

    def pipeline(self, raw_text: str) -> list[str]:
        """
        Processes the text through predefined processing stages.

        Parameters:
            raw_text (str): The raw input text.

        Returns:
            list[str]: The list of processed words.
        """

        prepared_text = self.__remove_NonASCII(raw_text)
        prepared_text =
self.__lowercase_and_remove_short_words(prepared_text)
        prepared_text = self.__remove_stopwords(prepared_text)
        prepared_text = self.__stemming(prepared_text)
        prepared_text = self.__tokenize(prepared_text)

        return prepared_text

    def __remove_NonASCII(self, text: str) -> str:
        """
        Removes all non-ASCII characters from the text.

        Parameters:
            text (str): The input text.

        Returns:
            str: The processed text.
        """

        for ch in text:
            if not ch in string.ascii_letters:
                text = text.replace(ch, ' ')

        return text

    def __lowercase_and_remove_short_words(self, text: str) -> str:
        """
        Converts the text to lowercase and removes short words.

        Parameters:
            text (str): The input text.

        Returns:
```

```
            str: The processed text.
        """

        return ' '.join([word for word in text.split() if len(word) >
self.__len_min_ch]).lower()

    def __remove_stopwords(self, text: str) -> str:
        """
        Removes stop words from the text.

        Parameters:
            text (str): The input text.

        Returns:
            str: The processed text.
        """

        return ' '.join([word for word in text.split() if word not in
self.__stop_words])

    def __stemming(self, text: str) -> str:
        """
        Performs stemming on words in the text.

        Parameters:
            text (str): The input text.

        Returns:
            str: The processed text.
        """

        stemmer = PorterStemmer()
        return ' '.join([stemmer.stem(word) for word in text.split()])

    def __tokenize(self, text: str) -> list[str]:
        """
        Tokenizes the text.

        Parameters:
            text (str): The input text.

        Returns:
            list[str]: The list of tokens.
        """

        return text.split()


if __name__ == "__main__":

    processor = TextProcessor(min_length=1, user_stop_words=('ect',
'enron', 'hou', 'hpl', 'subject'))

    raw_text = "This is a sample text for processing."
    processed_text = processor.pipeline(raw_text)

    print(processed_text)
```

TrainMultinomialNB.py

```
import multiprocessing
import time
import os
```

```
from functools import wraps

import pandas as pd

import TextProcessor


class TrainMultinomialNB:
    """
    A class for training a Multinomial Naive Bayes classifier for email
classification.

    Class Attributes:
        __processor (TextProcessor): An instance of TextProcessor is used
for text processing and tokenization.
            Default minimum word length is 2. Should be greater than or
equal to 0.
            Default includes stop words from the nltk and wordcloud
libraries.
        __min_word_frequency (int): The minimum word frequency threshold.
            Default is 15. Should be greater than or equal to 0.

    Attributes:
        __email_dataset (pd.DataFrame): The DataFrame containing email
data for training.
        __path_to_email_dataset (str): The file path to the DataFrame
file of emails and labels.
        __word_frequencies (dict): The dictionary containing word
frequencies; derived from the DataFrame with email data for training.
        __dataframe_email_ratios (pd.DataFrame): The DataFrame containing
ham and spam email ratios to total emails; derived from the DataFrame
with email data for training.
        __dataframe_word_frequencies (pd.DataFrame): The DataFrame
containing word frequencies; derived from the DataFrame with email data
for training.
    """

    __processor = TextProcessor.TextProcessor()
    __min_word_frequency = 15

    def __init__(self, path_to_email_dataset: str) -> None:
        """
        Initializes a TrainMultinomialNB object.

        Parameters:
            path_to_email_dataset (str): The file path to the DataFrame
file of emails and labels.
                The DataFrame must have the following structure:
                    Columns should be labeled as follows: text, label.
                        Note: label = 1 is spam, while label = 0 is ham.
                    Example:
                        ,text,label
                        0,Some text of some letter...,1
                        1,Some text of some letter...,1
                        2,Some text of some letter...,0
                        3,Some text of some letter...,1

        Raises:
            FileNotFoundError: If the specified DataFrame file of emails
and labels is not found.
            AttributeError: If the DataFrame file of emails and labels
has an incorrect structure.

        Returns:
```

```
            None
        """

        try:
            self.__email_dataset = pd.read_csv(path_to_email_dataset,
index_col=0)

        except FileNotFoundError:
            raise FileNotFoundError(
                f'\n\tError: The DataFrame file of emails and labels is
not found at "{path_to_email_dataset}".\n'
                f"\tThe DataFrame must have the following structure:\n"
                f"\t\tColumns should be labeled as follows: text,
label.\n"
                f"\tNote:\n"
                f"\t\tlabel = 1 is spam, while label = 0 is ham.\n\n"
                f"\tExample:\n"
                f"\t\t,text,label\n"
                f"\t\t0,Some text of some letter...,1\n"
                f"\t\t1,Some text of some letter...,1\n"
                f"\t\t2,Some text of some letter...,0\n"
                f"\t\t3,Some text of some letter...,1\n"
            )

        if not(['text', 'label'] == list(self.__email_dataset.columns) or
['label', 'text'] == list(self.__email_dataset.columns)):
            raise AttributeError(
                f'\n\tError: The DataFrame file of emails and labels has
an incorrect structure.\n'
                f"\tThe DataFrame must have the following structure:\n"
                f"\t\tColumns should be labeled as follows: text,
label.\n"
                f"\tNote:\n"
                f"\t\tlabel = 1 is spam, while label = 0 is ham.\n\n"
                f"\tExample:\n"
                f"\t\t,text,label\n"
                f"\t\t0,Some text of some letter...,1\n"
                f"\t\t1,Some text of some letter...,1\n"
                f"\t\t2,Some text of some letter...,0\n"
                f"\t\t3,Some text of some letter...,1\n"
            )

        self.__path_to_email_dataset = path_to_email_dataset
        self.__word_frequencies = {}
        self.__dataframe_email_ratios = pd.DataFrame
        self.__dataframe_word_frequencies = pd.DataFrame

    def __calculate_execution_time(method: callable) -> callable:
        """
        Decorator to calculate and print the execution time of the
decorated method.

        Parameters:
            method (callable): The method to be decorated.

        Returns:
            callable: The decorated method.
        """
        @wraps(method)
        def wrapper(*args, **kwargs) -> None:
            """
            Wrapper function that prints the beginning and end of the
decorated method's execution, along with the time spent on execution.
```

```
            Parameters:
                *args: Variable-length argument list.
                **kwargs: Arbitrary keyword arguments.

            Returns:
                None
            """

            if method.__name__ == 'train':
                print(f"The beginning of training.")
                start_time = time.time()
                method(*args, **kwargs)
                print(f"The end of training. The training was completed
in {'%.3f' % (time.time() - start_time)} seconds.")
            elif method.__name__ in
['__pipeline_prepare_dataframe_email_ratios',
'__pipeline_prepare_dataframe_word_frequencies']:
                print(f"\t{'*' * 60}")
                print(f'\tThe beginning of {method.__name__}.')
                start_time = time.time()
                method(*args)
                print(f"\tThe end of {method.__name__}. Time spent:
{'%.3f' % (time.time() - start_time)} seconds.")
                print(f"\t{'*' * 60}")
            else:
                print(f"\t\t{'-' * 80}")
                print(f"\t\tThe beginning of {method.__name__}.")
                start_time = time.time()
                method(*args)
                print(f"\t\tThe end of {method.__name__}. Time spent:
{'%.3f' % (time.time() - start_time)} seconds.")
                print(f"\t\t{'-' * 80}")

        return wrapper

    @__calculate_execution_time
    def train(self, output_folder_path: str = None) -> None:
        """
        Trains the Multinomial Naive Bayes classifier based on the
provided email dataset during object initialization.

        Parameters:
            output_folder_path (str, optional): The folder path where the
resulting DataFrames will be saved.
                If not provided or the folder does not exist, the
DataFrames will be saved next to the DataFrame file containing emails and
labels.

        Returns:
            None

        Note:
            The results of the trained Multinomial Naive Bayes classifier
will be stored in two DataFrames:
                ham and spam email ratios to total emails, and word
frequencies.
        """

        if output_folder_path is None or not
os.path.exists(output_folder_path):
            output_folder_path =
os.path.dirname(self.__path_to_email_dataset)
```

```
        path_to_save_dataset_word_frequencies =
os.path.join(output_folder_path, 'word_frequencies.csv')
        path_to_save_dataset_email_ratios =
os.path.join(output_folder_path, 'email_ratios.csv')


self.__pipeline_prepare_dataframe_email_ratios(path_to_save_dataset_email
_ratios)

self.__pipeline_prepare_dataframe_word_frequencies(path_to_save_dataset_w
ord_frequencies)

        print(f'\tThe dataset_email_ratios has been saved to
"{path_to_save_dataset_email_ratios}".')
        print(f'\tThe dataset_word_frequencies has been saved to
"{path_to_save_dataset_word_frequencies}".')

    @__calculate_execution_time
    def __pipeline_prepare_dataframe_email_ratios(self,
path_to_save_dataset_email_ratios: str) -> None:
        """
        Implements a pipeline to prepare and save the DataFrame with ham
and spam email ratios to total emails.
            This includes initializing and preparing the DataFrame with
ham and spam email ratios to total emails
            and saving the resulting DataFrame to the specified path.

        Parameters:
            path_to_save_dataset_email_ratios (str): The file path to
save the DataFrame file of email ratios.

        Returns:
            None
        """

        self.__initialize_dataframe_email_ratios()
        self.__save_dataframe(self.__dataframe_email_ratios,
path_to_save_dataset_email_ratios)

    @__calculate_execution_time
    def __pipeline_prepare_dataframe_word_frequencies(self,
path_to_save_dataset_word_frequencies: str) -> None:
        """
        Implements a pipeline to prepare and save the DataFrame with word
frequencies.
            This includes merging text categories, processing text,
initializing word frequencies, counting word frequencies,
            removing low-frequency words, sorting words by frequency, and
saving the resulting DataFrame to the specified path.

        Parameters:
            path_to_save_dataset_word_frequencies (str): The file path to
save the DataFrame file of word frequencies.

        Returns:
            None
        """

        self.__merge_text_category()

        print(f"\t\t{'-' * 80}")
        print(f"\t\tThe beginning of __prepare_text.")
        start_time = time.time()
```

```
        self.__email_dataset['text'] =
self.__email_dataset['text'].apply(self.__prepare_text)
        print(f"\t\tThe end of prepare_text. Time spent: {'%.3f' %
(time.time() - start_time)} seconds.")
        print(f"\t\t{'-' * 80}")

        self.__initialize_dict_word_frequencies()
        self.__count_word_frequencies()

self.__remove_words_with_low_frequency_and_sort_word_frequencies()
        self.__initialize_dataframe_word_frequencies()
        self.__save_dataframe(self.__dataframe_word_frequencies,
path_to_save_dataset_word_frequencies)

    @__calculate_execution_time
    def __merge_text_category(self) -> None:
        """
        Merges text categories in the email dataset based on labels.

        Parameters:
            None

        Returns:
            None
        """

        self.__email_dataset =
self.__email_dataset.groupby('label')['text'].apply(lambda x: '
'.join(x)).reset_index()
        self.__email_dataset = self.__email_dataset[['text', 'label']]

    def __prepare_text(self, text: str) -> list[str]:
        """
        Processes and tokenizes the input text.

        Parameters:
            text (str): The input text to be processed and tokenized.

        Returns:
            list[str]: The list of processed and tokenized words.

        Note:
            This method utilizes multiprocessing to expedite the text
processing.
            Parallelization is performed across all available CPU cores.
        """

        words = text.split()
        text = []

        step = int(len(words) / (multiprocessing.cpu_count()))

        for i in range(multiprocessing.cpu_count()-1):
            text.append(' '.join(words[step*i:step*(i+1)]))
        else:
            text.append(' '.join(words[step*(multiprocessing.cpu_count()-
1):len(words)]))

        with multiprocessing.Pool(multiprocessing.cpu_count()) as p:
            processed_text = p.map(self.__processor.pipeline, text)

        return [word for sublist in processed_text for word in sublist]

    @__calculate_execution_time
```

```
    def __initialize_dict_word_frequencies(self) -> None:
        """
        Initializes the dictionary for word frequencies.

        Parameters:
            None

        Returns:
            None

        Note:
            The dictionary self.__word_frequencies is initialized such
that each unique word in the texts of email messages
                is assigned a list [0, 0, 0], where the first element is
the overall frequency of the word,
                the second is the frequency in ham emails, and the third
is the frequency in spam emails.
        """

        unique_words = set.union(set(self.__email_dataset['text'][0]),
set(self.__email_dataset['text'][1]))
        self.__word_frequencies = {key: [0, 0, 0] for key in
unique_words}

    @__calculate_execution_time
    def __count_word_frequencies(self) -> None:
        """
        Counts word frequencies in the email dataset.

        Parameters:
            None

        Returns:
            None
        """

        for word in self.__email_dataset['text'][0]:
            self.__word_frequencies[word][0] += 1
            self.__word_frequencies[word][1] += 1

        for word in self.__email_dataset['text'][1]:
            self.__word_frequencies[word][0] += 1
            self.__word_frequencies[word][2] += 1

    @__calculate_execution_time
    def __remove_words_with_low_frequency_and_sort_word_frequencies(self)
-> None:
        """
        Removes low-frequency words and sorts the word frequencies
dictionary in descending order.

        Parameters:
            None

        Returns:
            None

        Note:
            The threshold for low-frequency words is determined by the
value set in the class attribute self.__min_word_frequency.
        """

        self.__word_frequencies = dict(sorted(filter(lambda item:
item[1][0] >= self.__min_word_frequency,
```

```
self.__word_frequencies.items()), key=lambda item: item[1][0],
reverse=True))

    @__calculate_execution_time
    def __initialize_dataframe_word_frequencies(self) -> None:
        """
        Initializes the DataFrame for word frequencies.

        Parameters:
            None

        Returns:
            None

        Note:
            The DataFrame self.__dataframe_word_frequencies is
initialized using the dictionary of word frequencies (i.e.,
self.__word_frequencies).
        """

        self.__dataframe_word_frequencies =
pd.DataFrame.from_dict(self.__word_frequencies, orient='index',
columns=['frequency', 'frequency_ham', 'frequency_spam'])

    @__calculate_execution_time
    def __save_dataframe(self, dataframe: pd.DataFrame, path_to_save:
str) -> None:
        """
        Saves the given DataFrame to the specified file path.

        Parameters:
            dataframe (pd.DataFrame): The DataFrame to be saved.
            path_to_save (str): The file path to save the DataFrame.

        Returns:
            None
        """

        dataframe.to_csv(path_to_save)

    @__calculate_execution_time
    def __initialize_dataframe_email_ratios(self) -> None:
        """
        Initializes and prepares the DataFrame with ham and spam email
ratios to total emails.

        Parameters:
            None

        Returns:
            None

        Note:
            Calculates ham and spam email ratios to total emails in the
training DataFrame with email data.
            The resulting DataFrame (self.__dataframe_email_ratios) has
'ham' and 'spam' as columns and 'ratios-to-total-emails' as the index.
        """

        count_ham =
len(self.__email_dataset[self.__email_dataset['label'] == 0])
        count_spam =
len(self.__email_dataset[self.__email_dataset['label'] == 1])
```

```
        self.__dataframe_email_ratios = pd.DataFrame({'ham':
[count_ham/(count_ham + count_spam)], 'spam': [count_spam/(count_ham +
count_spam)]}, index=['ratios-to-total-emails'])

    @classmethod
    def set_processor(cls, min_length: int = None, user_stop_words:
list[str] | tuple[str] | set[str] = None) -> None:
        """
        Set parameters for an instance of TextProcessor to be used for
text processing and tokenization.

        Parameters:
            min_length (int, optional): Minimum word length.
                Default is 2. Should be greater than or equal to 0.
            user_stop_words (list[str] | tuple[str] | set[str],
optional): The user-defined list of stop words. This list is added to the
stop words defined by default.
                Default includes stop words from the nltk and wordcloud
libraries.
                Example:
                    user_stop_words = ['ect', 'enron', 'hou', 'hpl',
'subject']

        Returns:
            None
        """

        cls.__processor = TextProcessor.TextProcessor(min_length,
user_stop_words)

    @classmethod
    def set_min_word_frequency(cls, min_word_frequency: int) -> None:
        """
        Set the minimum word frequency threshold for filtering words in
the dataset.

        Parameters:
            min_word_frequency (int): The minimum word frequency
threshold. Should be greater than or equal to 0.

        Raises:
            ValueError: If the provided min_word_frequency is less than
0.

        Returns:
            None
        """

        if min_word_frequency >= 0:
            cls.__min_word_frequency = min_word_frequency
        else:
            raise ValueError(f"Error: min_word_frequency should be
greater than or equal to 0. Got: {min_word_frequency}")


if __name__ == "__main__":

    path_to_email_dataset =
'../dataset/email_dataset_1/model_data/spam_ham_dataset_ready_to_train.cs
v'

    output_folder_path = '../dataset/email_dataset_1/model_data/'

    training = TrainMultinomialNB(path_to_email_dataset)
```

```
    training.set_processor(user_stop_words=['ect', 'enron', 'hou', 'hpl',
'subject'])

    training.train(output_folder_path)
```