Работа с текстом

В этом домашнем задании вам предстоит поработать с текстовыми данными и научиться находить спам сообщения!

In [1]:
```
import inspect
import math
import random
import re
from collections import Counter, defaultdict
from string import punctuation

import numpy as np
from nltk import SnowballStemmer, download
from nltk.corpus import stopwords
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, accuracy_score
from sklearn.model_selection import KFold, train_test_split
from scipy.special import logsumexp
```

In [2]:
```
download("stopwords")
```

```
[nltk_data] Downloading package stopwords to /home/sashka/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

Out[2]: True

In [3]:
```
def set_seed(seed=42):
    np.random.seed(seed)
    random.seed(seed)


# Этой функцией будут помечены все места, которые необходимо дозаполнить
# Это могут быть как целые функции, так и отдельные части внутри них
# Всегда можно воспользоваться интроспекцией и найти места использования этой функции :)
def todo():
    stack = inspect.stack()
    caller_frame = stack[1]
    function_name = caller_frame.function
    line_number = caller_frame.lineno
    raise NotImplementedError(f"TODO at {function_name}, line {line_number}")


SEED = 0xC0FFEE
set_seed(SEED)
```

In [4]:
```
def read_dataset(filename):
    x, y = [], []
    with open(filename, encoding="utf-8") as file:
        for line in file:
            cl, sms = re.split(r"^(ham|spam)[\t\s]+(.*)$", line)[1:3]
            x.append(sms)
            y.append(cl)
    return x, y
```

In [5]:
```
X, y = read_dataset("src/homeworks/homework8/spam.txt")
```

In [6]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.9, random_state=SEED, stratify=y)
```

In [7]:
```
for x_, y_ in zip(X_train[:5], y_train[:5]):
    print(f"{y_}: {x_}")
```

```
ham: Two fundamentals of cool life: "Walk, like you are the KING"...! OR "Walk like you Dont care,whoever is the KIN
G"!... Gud nyt
ham: Haha... Where got so fast lose weight, thk muz go 4 a month den got effect... Gee,later we go aust put bk e wei
ght.
ham: I wish things were different. I wonder when i will be able to show you how much i value you. Pls continue the b
risk walks no drugs without askin me please and find things to laugh about. I love you dearly.
ham: Tmr then ü brin lar... Aiya later i come n c lar... Mayb ü neva set properly ü got da help sheet wif ü...
ham: For many things its an antibiotic and it can be used for chest abdomen and gynae infections even bone infection
s.
```

In [8]:
```
Counter(y_train)
```

Out[8]: Counter({'ham': 4344, 'spam': 672})

Bag of Words (2 балла)

Реализуйте простой подсчет слов в тексте, в качестве токенизатора делите по пробелу, убрав перед этим все знаки пунктуации и приведя к нижнему регистру.

После этого обучите простую логистическую модель, измерьте ее качество и сделайте выводы.

In [9]:
```python
class BagOfWords:
    def __init__(self, vocabulary_size: int = 1000):
        """Init Bag-of-Words instance

        Args:
            vocabulary_size: maximum number of tokens in vocabulary
        """

        self._vocabulary_size = vocabulary_size
        self._vocabulary: dict[str, int] = None

    def _tokenize(self, sentence: str) -> list[str]:
        sentence = sentence.lower()
        translator = str.maketrans('', '', punctuation)
        sentence = sentence.translate(translator)
        tokens = sentence.split()

        return [token for token in tokens if token]

    def fit(self, sentences: list[str]):
        """Fit Bag-of-Words based on list of sentences"""

        all_tokens = []

        for sentence in sentences:
            all_tokens.extend(self._tokenize(sentence))

        token_counts = Counter(all_tokens)
        most_common_tokens = token_counts.most_common(self._vocabulary_size)

        self._vocabulary = {token: i for i, (token, _) in enumerate(most_common_tokens)}

    def transform(self, sentences: list[str]) -> np.ndarray:
        """Vectorize texts using built vocabulary

        Args:
            sentences: list of sentences to vectorize

        Return:
            transformed texts, matrix of (n_sentences, vocab_size)
        """

        if self._vocabulary is None:
            raise RuntimeError("Fit before transforming!")

        num_features = len(self._vocabulary)

        bow_matrix = np.zeros((len(sentences), num_features), dtype=int)

        for i, sentence in enumerate(sentences):
            tokens = self._tokenize(sentence)
            for token in tokens:
                if token in self._vocabulary:
                    token_idx = self._vocabulary[token]
                    bow_matrix[i, token_idx] += 1

        return bow_matrix

    def fit_transform(self, sentences: list[str]) -> np.ndarray:
        self.fit(sentences)
        return self.transform(sentences)
```

In [29]:
```python
def get_accuracy(bow_model, size: int, param: dict, model) -> int:
    bow = bow_model(vocabulary_size=size, **param)

    X_train_search, X_val_search, y_train_search, y_val_search = train_test_split(X_train, y_train, test_size=0.1,

    X_train_bow = bow.fit_transform(X_train_search)
    X_val_bow = bow.transform(X_val_search)

    model = model()
    model.fit(X_train_bow, y_train_search)

    y_pred = model.predict(X_val_bow)
    return accuracy_score(y_val_search, y_pred)
```

In [30]:
```python
def get_best_param(bow_model, params: list[dict] = None, model=LogisticRegression) -> dict:
    sizes = range(1, 3500, 100)
    best_accuracy = -1
    best_params = {}

    if params is None:
        params = [{}]
```

```
    for size in sizes:
        for param in params:
            cur_accuracy = get_accuracy(bow_model, size, param, model)

            if cur_accuracy > best_accuracy:
                best_accuracy = cur_accuracy
                best_params = {"vocabulary_size": size, **param}

    return best_params
```

In [12]:
```
bow_best_param = get_best_param(BagOfWords)

bow = BagOfWords(**bow_best_param)
X_train_bow = bow.fit_transform(X_train)
X_test_bow = bow.transform(X_test)

X_train_bow.shape, X_test_bow.shape
```

Out[12]:
```
((5016, 701), (558, 701))
```

In [13]:
```
bow_best_param
```

Out[13]:
```
{'vocabulary_size': 701}
```

In [14]:
```
model = LogisticRegression()
model.fit(X_train_bow, y_train)

y_pred = model.predict(X_test_bow)
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

         ham       0.99      1.00      0.99       483
        spam       0.99      0.93      0.96        75

    accuracy                           0.99       558
   macro avg       0.99      0.97      0.98       558
weighted avg       0.99      0.99      0.99       558
```

Оптимальный размер словаря: 701

Метрики при нем достаточно высокие: ham определяется почти идеально. Для spam относительно низкое значение recall: модель пропускает spam.

Обработка текста (1 балл)

Добавьте на этапе токенизатора удаление стоп-слов и стемминг, для этого можно воспользоваться `SnowballStemmer` из библиотеки `nltk`.

⚠️ `nltk` уже довольно устаревшая библиотека и скорее не рекомендуется ее использовать, однако в учебных целях более чем достаточно.

Обучите логистическую регрессию, попробуйте по-разному комбинировать стемминг и удаление стоп-слов, сделайте выводы.

In [15]:
```
class BagOfWordsStem(BagOfWords):
    def __init__(
        self,
        vocabulary_size: int,
        language: str = "english",
        ignore_stopwords: bool = True,
        remove_stopwords: bool = True,
    ):
        super().__init__(vocabulary_size)

        if remove_stopwords and not ignore_stopwords:
            raise ValueError("To remove stop-words they should be ignored by stemmer")

        self._stemmer = SnowballStemmer(language)
        self._stopwords = set(stopwords.words(language))

        self._remove_stopwords = remove_stopwords
        self._ignore_stopwords = ignore_stopwords

    def _tokenize(self, sentence: str) -> list[str]:
        tokens = super()._tokenize(sentence)

        processed_tokens = []

        for token in tokens:
            is_stopword = token in self._stopwords

            if is_stopword and self._remove_stopwords:
```

```
                continue

            if is_stopword and self._ignore_stopwords:
                processed_tokens.append(token)
            else:
                processed_tokens.append(self._stemmer.stem(token))

        return processed_tokens
```

In [16]:
```
params = [{"remove_stopwords": True, "ignore_stopwords": True},
          {"remove_stopwords": False, "ignore_stopwords": True},
          {"remove_stopwords": False, "ignore_stopwords": False}]

bows_best_param = get_best_param(BagOfWordsStem, params)

bow = BagOfWordsStem(**bows_best_param)
X_train_bow = bow.fit_transform(X_train)
X_test_bow = bow.transform(X_test)

X_train_bow.shape, X_test_bow.shape
```

Out[16]:
```
((5016, 401), (558, 401))
```

In [17]:
```
bows_best_param
```

Out[17]:
```
{'vocabulary_size': 401, 'remove_stopwords': False, 'ignore_stopwords': True}
```

In [18]:
```
model = LogisticRegression()
model.fit(X_train_bow, y_train)

y_pred = model.predict(X_test_bow)
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

         ham       0.99      1.00      0.99       483
        spam       0.99      0.93      0.96        75

    accuracy                           0.99       558
   macro avg       0.99      0.97      0.98       558
weighted avg       0.99      0.99      0.99       558
```

Оптимальный размер словаря: 401, remove_stopwords: False, ignore_stopwords: True

Лучшее значение accuracy достигается при неудалении стоп-слов и их игнорировании стеммером. То есть стоп-слова сохраняются в их изначальной форме. Оптимальный размер словаря ожидаемо меньше, чем при методе без стемминга.

Метрики не изменились, значит достигли такой же точности с меньшим словарем.

TF-IDF (2 балла)

Доработайте предыдущий класс до полноценного Tf-Idf, затем, аналогично, проведите эксперименты с логистической регрессией.

In [19]:
```
class Tokenizer:
    def __init__(
        self,
        vocabulary_size: int,
        language: str = "english",
        ignore_stopwords: bool = True,
        remove_stopwords: bool = True,
    ):
        self._vocabulary_size = vocabulary_size
        self._vocabulary: dict[str, int] = None

        if remove_stopwords and not ignore_stopwords:
            raise ValueError("To remove stop-words they should be ignored by stemmer")

        self._stemmer = SnowballStemmer(language)
        self._stopwords = set(stopwords.words(language))

        self._remove_stopwords = remove_stopwords
        self._ignore_stopwords = ignore_stopwords

    def tokenize(self, sentence: str) -> list[str]:
        sentence = sentence.lower()
        translator = str.maketrans('', '', punctuation)
        sentence = sentence.translate(translator)
        tokens = sentence.split()

        tokens = [token for token in tokens if token]
        processed_tokens = []

        for token in tokens:
```

```
            is_stopword = token in self._stopwords

            if is_stopword and self._remove_stopwords:
                continue

            if is_stopword and self._ignore_stopwords:
                processed_tokens.append(token)
            else:
                processed_tokens.append(self._stemmer.stem(token))

        return processed_tokens
```

In [20]:

```
class TFIDFVectorizer:
    def __init__(
        self,
        vocabulary_size: int,
        language: str = "english",
        ignore_stopwords: bool = True,
        remove_stopwords: bool = True,
        use_idf: bool = False,
    ):
        self._vocabulary_size = vocabulary_size
        self._vocabulary = None
        self._idf = None
        self._use_idf = use_idf

        self._tokenizer = Tokenizer(vocabulary_size, language, ignore_stopwords, remove_stopwords)

    def _tokenize(self, sentence: str) -> list[str]:
        return self._tokenizer.tokenize(sentence)

    def fit(self, sentences: list[str]):
        """Build vocabulary and compute IDF"""

        all_tokens = []
        document_frequency = defaultdict(int)

        for sentence in sentences:
            tokens = self._tokenize(sentence)
            all_tokens.extend(tokens)

            for token in set(tokens):
                document_frequency[token] += 1

        token_counts = Counter(all_tokens)
        most_common_tokens = token_counts.most_common(self._vocabulary_size)

        self._vocabulary = {token: i for i, (token, _) in enumerate(most_common_tokens)}

        n_sentences = len(sentences)

        if self._use_idf:
            self._idf = np.zeros(len(self._vocabulary))
            for token, i in self._vocabulary.items():
                n_w = document_frequency.get(token, 0)
                self._idf[i] = np.log(n_sentences / (n_w + 1.0)) + 1.0
        else:
            self._idf = np.ones(len(self._vocabulary))

    def transform(self, sentences: list[str]) -> np.ndarray:
        """Transform sentences to TF-IDF vectors"""

        n_sentences = len(sentences)
        n_features = len(self._vocabulary)
        tfidf_matrix = np.zeros((n_sentences, n_features), dtype=float)

        for i, sentence in enumerate(sentences):
            tokens = self._tokenize(sentence)
            if not tokens:
                continue

            sentence_len = len(tokens)
            token_counts_in_sentence = Counter(tokens)

            for token, count in token_counts_in_sentence.items():
                if token in self._vocabulary:
                    token_idx = self._vocabulary[token]

                    if self._use_idf:
                        tfidf_matrix[i, token_idx] = count / sentence_len * self._idf[token_idx]
                    else:
                        tfidf_matrix[i, token_idx] = count

        return tfidf_matrix

    def fit_transform(self, sentences: list[str]) -> np.ndarray:
```

```
        self.fit(sentences)
        return self.transform(sentences)
```

In [21]:
```
params = [
    {"remove_stopwords": True, "ignore_stopwords": True, "use_idf": True},
    {"remove_stopwords": False, "ignore_stopwords": True, "use_idf": True},
    {"remove_stopwords": False, "ignore_stopwords": False, "use_idf": True}
]
tfidf_best_param = get_best_param(TFIDFVectorizer, params)

tfidf = TFIDFVectorizer(**tfidf_best_param)
X_train_tfidf = tfidf.fit_transform(X_train)
X_test_tfidf = tfidf.transform(X_test)

X_train_tfidf.shape, X_test_tfidf.shape
```

Out[21]:
```
((5016, 701), (558, 701))
```

In [22]:
```
tfidf_best_param
```

Out[22]:
```
{'vocabulary_size': 701,
 'remove_stopwords': False,
 'ignore_stopwords': True,
 'use_idf': True}
```

In [23]:
```
tfidf = TFIDFVectorizer(**tfidf_best_param)
X_train_tfidf = tfidf.fit_transform(X_train)
X_test_tfidf = tfidf.transform(X_test)
```

In [24]:
```
model = LogisticRegression()
model.fit(X_train_tfidf, y_train)

y_pred = model.predict(X_test_tfidf)
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

         ham       0.97      1.00      0.98       483
        spam       0.97      0.83      0.89        75

    accuracy                           0.97       558
   macro avg       0.97      0.91      0.94       558
weighted avg       0.97      0.97      0.97       558
```

При использовании idf:

Оптимальный размер словаря: 701, remove_stopwords: False, ignore_stopwords: True

Лучшее значение accuracy так же как и для BagOfWordsStem достигается при неудалении стоп-слов и их игнорировании стеммером.

Метрики в целом снизились. Скорее всего это связано с тем, что слова, которые часто встречаются в спаме (низкий IDF), являются сильными идентификаторами спама, а TF-IDF снижает их занчимость.

NaiveBayes (5 баллов)

Наивный байесовский классификатор — это простой и эффективный алгоритм машинного обучения, основанный на теореме Байеса с наивным предположением независимости признаков.

Формула Байеса

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$$

В контексте классификации текста это значит: $P(\text{класс}|\text{документ}) \propto P(\text{класс}) \cdot P(\text{документ}|\text{класс})$

Почему "наивность"? Потому что предпологаем, что все слова независимы:

$$P(w_1, w_2, \ldots | class) = P(w_1|class) \cdot P(w_2|class) \cdot \ldots$$

Классификация текста

Таким образом, для классификации текста необходимо:

1. Вычислить априорную вероятность класса: $P(class)$, доля документов с таким классом
2. Вычислить правдоподобие: $P(text|class) = \prod_{i=1}^{n} P(w_i|class)$

Примечание: $P(w_i|class)$ — это частота слова в данном классе относительно всех слов в классе, при этом зачастую добавляют сглаживание Лапласа в качестве регуляризатора

$$P(w_i|class) = \frac{\text{частота слова в классе} + \alpha}{\text{сумма всех слов в классе} + \alpha \cdot |V|}$$

После этого, необходимо выбрать наиболее вероятный класс для данного текста:

$$class = \arg\max_c \Big[P(c) \cdot P(text|c)\Big] = \arg\max_c \Big[\log P(c) + \sum_{i=1}^{n} \log P(w_i|c)\Big]$$

Реализация

`fit(X, y)` - оценивает параметры распределения `p(x|y)` для каждого `y`.

`log_proba(X)` - для каждого элемента набора `X` считает логарифм вероятности отнести его к каждому классу.

In [25]:
```python
class NaiveBayes:

    def __init__(self, alpha: float = 1.0):
        """
        Args:
            alpha: regularization coefficient
        """
        self.alpha = alpha
        self._classes = None  # [n classes]
        self._vocab_size = None  # int
        self._log_p_y = None  # [n classes]
        self._log_p_x_y = None  # [n classes, vocab size]

    def fit(self, features: np.ndarray, targets: list[str]):
        """Estimate p(x|y) and p(y) based on data

        Args:
            features, [n samples; vocab size]: input features
            targets, [n samples]: targets
        """
        targets = np.array(targets)

        self._classes = np.unique(targets)
        n_classes = len(self._classes)
        n_samples, self._vocab_size = features.shape

        self._log_p_y = np.zeros(n_classes, dtype=np.float64)
        self._log_p_x_y = np.zeros((n_classes, self._vocab_size), dtype=np.float64)

        for i, cls in enumerate(self._classes):
            features_cls = features[targets == cls]

            n_samples_in_class = features_cls.shape[0]

            if n_samples_in_class == 0:
                self._log_p_y[i] = -np.inf
            else:
                self._log_p_y[i] = np.log(n_samples_in_class / n_samples)

            feature_counts_in_class = np.sum(features_cls, axis=0)
            total_features_in_class = np.sum(feature_counts_in_class)

            numerator = feature_counts_in_class + self.alpha
            denominator = total_features_in_class + self.alpha * self._vocab_size

            if denominator == 0:
                self._log_p_x_y[i, :] = -np.inf
            else:
                self._log_p_x_y[i, :] = np.log(numerator / denominator)


    def predict(self, features: np.ndarray) -> np.ndarray:
        """Predict class for each sample

        Args:
            features, [n samples; vocab size]: feature to predict
        Return:
            classes, [n samples]: predicted class
        """
        log_probabilities = self.log_proba(features)
        predicted_class_indices = np.argmax(log_probabilities, axis=1)
        predicted_classes = self._classes[predicted_class_indices]
        return predicted_classes

    def log_proba(self, features: np.ndarray) -> np.ndarray:
        """Calculate p(y|x) for each class and each sample

        Args:
            features, [n samples; vocab size]: feature to predict
        Return:
```

```
            classes, [n samples;  n classes]: log proba for each class
        """
        if self._vocab_size is None:
            raise RuntimeError("Fit classifier before predicting something")
        if features.shape[1] != self._vocab_size:
            raise RuntimeError(
                f"Unexpected size of vocabulary, expected {self._vocab_size}, actual {features.shape[1]}"
            )

        n_samples = features.shape[0]
        log_probabilities = (features @ self._log_p_x_y.T) + self._log_p_y[np.newaxis, :]
        log_p_x = logsumexp(log_probabilities, axis=1, keepdims=True)
        log_posterior_proba = log_probabilities - log_p_x
        return log_posterior_proba
```

In [34]:
```
params = [{"remove_stopwords": True, "ignore_stopwords": True},
          {"remove_stopwords": False, "ignore_stopwords": True},
          {"remove_stopwords": False, "ignore_stopwords": False}]

bayes_bows_best_param = get_best_param(BagOfWordsStem, params, model=NaiveBayes)
```

In [35]:
```
bayes_bows_best_param
```

Out[35]:
```
{'vocabulary_size': 1001, 'remove_stopwords': False, 'ignore_stopwords': True}
```

In [36]:
```
bow = BagOfWordsStem(**bayes_bows_best_param)
X_train_bow = bow.fit_transform(X_train)
X_test_bow = bow.transform(X_test)

X_train_bow.shape, X_test_bow.shape
```

Out[36]:
```
((5016, 1001), (558, 1001))
```

In [37]:
```
model = NaiveBayes(alpha=1.0)
model.fit(X_train_bow, y_train)

y_pred = model.predict(X_test_bow)
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

         ham       0.99      0.99      0.99       483
        spam       0.93      0.95      0.94        75

    accuracy                           0.98       558
   macro avg       0.96      0.97      0.97       558
weighted avg       0.98      0.98      0.98       558
```

Accuracy модели составила 0.98.

Recall для spam достиг 0.95, что выше, чем у моделей на основе LogisticRegression. Значит, что NaiveBayes лучше других идентифицирует спам-сообщения, пропуская меньше из них. При этом precision для spam ниже, чем у LogisticRegression, что указывает на большее количество ложных срабатываний. Метрики для ham остаются высокими.

Скорее всего это происходит из-за того, что NaiveBayes предполагает, что все слова-признаки независимы друг от друга при условии класса. Некоторые слова могут часто встречаться вместе в спам-сообщениях, и их комбинация является сильным индикатором спама. Логистическая регрессия, не делает такого предположения и может лучше улавливать такие зависимости, учась находить гиперплоскость, а комбинации признаков могут влиять на ее положение.