Comparative SMS Spam Detection Using Statistical and Embedding-based Models

Module: CM3060 Natural Language Processing

Abstract

This notebook compares a traditional statistical text classifier with a modern embedding-based deep learning classifier on the task of SMS spam detection. The statistical model combines a term frequency-inverse document frequency (TF-IDF) representation with logistic regression. The embedding model combines pre-trained GloVe word vectors with a bidirectional long short-term memory (BiLSTM) network. A majority-class classifier provides a performance floor, and published results for the same dataset provide an external reference. All models are trained and assessed on the SMS Spam Collection, a small and imbalanced corpus of about 5,572 English messages of which roughly 13 percent are spam. Because the classes are imbalanced, accuracy alone is treated as insufficient, and precision, recall, and F1 on the spam class serve as the primary measures, supported by macro-averaged F1, confusion matrices, ROC-AUC, and precision-recall AUC. The work reports tuned settings, training behaviour, and a like-for-like comparison on one shared held-out test set, and closes with an evidence-based discussion of when each approach is preferable.

Setup: imports, reproducibility, and constants

The cell below imports the libraries, fixes a single global random seed across Python, NumPy, and TensorFlow, and prints the library versions so the environment is recorded. Every tunable value used later is defined here as a named constant, which keeps the code free of unexplained literals and makes the experimental settings easy to audit and change. A GPU runtime should be selected on Google Colab before running the notebook, because the BiLSTM trains considerably faster on a GPU. Residual non-determinism can remain on a GPU even with a fixed seed, so small run to run variation in the neural results is expected and is acknowledged here. The dataset and the GloVe vectors are fetched from external sources in code, so the first run needs internet access unless those files are already present in the data directory, in which case the guarded downloads are skipped.

```
# Standard lib imports
import os
import random
import zipfile
import urllib.request

# Scientific stack
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

```
# scikit-learn
import sklearn
from sklearn.model_selection import train_test_split, GridSearchCV,
StratifiedKFold
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
from sklearn.utils.class_weight import compute_class_weight
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, f1_score,
    classification_report, confusion_matrix, roc_auc_score,
    average_precision_score, precision_recall_curve,
)

# TensorFlow Keras
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import (
    Embedding, Bidirectional, LSTM, Dense, Dropout, SpatialDropout1D,
)
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping

# Reproducibility: one global seed applied everywhere feasible
RANDOM_SEED = 42
os.environ["PYTHONHASHSEED"] = str(RANDOM_SEED)
random.seed(RANDOM_SEED)
np.random.seed(RANDOM_SEED)
tf.random.set_seed(RANDOM_SEED)

# Class labelling convention
HAM_LABEL = 0
SPAM_LABEL = 1
CLASS_NAMES = ["ham", "spam"]

# Data acquisition
DATA_DIR = "data"
DATASET_URL =
"https://archive.ics.uci.edu/static/public/228/sms+spam+collection.zip
"
DATASET_ZIP = os.path.join(DATA_DIR, "sms_spam_collection.zip")
DATASET_FILE = os.path.join(DATA_DIR, "SMSSpamCollection")
GLOVE_URL = "https://downloads.cs.stanford.edu/nlp/data/glove.6B.zip"
GLOVE_ZIP = os.path.join(DATA_DIR, "glove.6B.zip")
GLOVE_FILE = os.path.join(DATA_DIR, "glove.6B.100d.txt")

# Shared evaluation protocol
TEST_SIZE = 0.20
```

```
VALIDATION_SIZE = 0.10
CV_FOLDS = 5

# Statistical pipeline (TF-IDF and logistic regression)
TFIDF_NGRAM_RANGE = (1, 2)
TFIDF_MIN_DF = 2
LOGREG_C_GRID = [0.1, 0.5, 1.0, 2.0, 5.0, 10.0]
LOGREG_MAX_ITER = 1000

# Embedding pipeline (GloVe and BiLSTM)
EMBEDDING_DIM = 100
SEQUENCE_LENGTH_PERCENTILE = 95
LSTM_UNITS = 64
SPATIAL_DROPOUT_RATE = 0.2
DROPOUT_RATE = 0.5
BATCH_SIZE = 32
MAX_EPOCHS = 30
EARLY_STOPPING_PATIENCE = 4
LEARNING_RATE = 1e-3
OOV_TOKEN = "<oov>"

# Presentation
DECIMALS = 4
FIGSIZE_WIDE = (8, 4)
FIGSIZE_SQUARE = (4.5, 4)

plt.rcParams["figure.dpi"] = 110
os.makedirs(DATA_DIR, exist_ok=True)

print("Library versions")
print(f"  numpy        {np.__version__}")
print(f"  pandas       {pd.__version__}")
print(f"  scikit-learn {sklearn.__version__}")
print(f"  tensorflow   {tf.__version__}")
print(f"  GPU visible
{bool(tf.config.list_physical_devices('GPU'))}")
```

```
Library versions
  numpy        2.1.2
  pandas       2.2.2
  scikit-learn 1.5.2
  tensorflow   2.18.0
  GPU visible  False
```

1. Introduction to the domain-specific area

Short Message Service (SMS) is a widely used messaging channel, and its reach makes it a target for unsolicited messages. SMS spam ranges from advertising to prize scams and phishing that tries to harvest credentials or push malware links. These messages waste user attention and expose recipients to fraud, so filtering them automatically has clear practical value. The task is a

binary text classification problem: given the words of a message, decide whether it is legitimate, conventionally called ham, or spam.

Content-based spam filtering has been studied for decades, first for email and later for SMS. The move to SMS is harder in specific ways. Messages are short, they use informal spelling, abbreviations, and slang, and they contain currency symbols, digits, and telephone numbers that often carry signal about intent. Almeida et al. (2011) assembled the SMS Spam Collection to support research on this setting and reported that established content-based methods transfer to SMS with strong results, which fixes the corpus used throughout this notebook.

The domain motivates a comparison between two model families. A statistical classifier over sparse lexical features is fast, interpretable, and well matched to the strong surface patterns of spam, such as the recurring vocabulary of prizes and premium-rate numbers. An embedding-based sequence model represents words as dense vectors and can use word order, which a frequency table discards (Pennington et al., 2014). Comparing the two on a shared benchmark clarifies what each contributes on short, noisy, and imbalanced text, and where the extra cost of a neural model is repaid.

2. Objectives of the project

The central objective is to compare the effectiveness and applicability of a traditional statistical model and a modern embedding-based deep learning model for SMS spam detection, using one dataset, one held-out test set, and one set of imbalance-aware metrics so that the comparison is fair. The study pursues five specific goals.

First, establish a meaningful baseline that acts as a performance floor. A majority-class classifier that always predicts ham quantifies the score obtainable with no learning and exposes why accuracy overstates competence on an imbalanced corpus. Because that floor is trivial to clear, published results for the same dataset supplement it with a competent target that a useful model should match or beat.

Second, implement and tune a statistical classifier that pairs a TF-IDF representation with logistic regression. TF-IDF weights informative terms and downweights ubiquitous ones (Manning et al., 2008), and logistic regression provides an interpretable linear decision over those features, with regularisation strength selected by cross-validation.

Third, implement and tune an embedding-based classifier that pairs pre-trained GloVe vectors (Pennington et al., 2014) with a bidirectional LSTM (Hochreiter and Schmidhuber, 1997). The design is trained first with the embedding layer frozen and then with fine-tuning enabled, so the effect of adapting the vectors to this corpus can be measured directly.

Fourth, evaluate every model on the same held-out test set with metrics chosen for class imbalance, namely precision, recall, and F1 on the spam class, macro-averaged F1, confusion matrices, ROC-AUC, and precision-recall AUC.

Fifth, draw evidence-based conclusions about when each approach is preferable, giving explicit attention to the asymmetric cost of a missed spam message against a wrongly blocked legitimate message.

The intended contribution is practical rather than theoretical. By reporting a careful, reproducible comparison on a public benchmark, the work clarifies the trade-off between a

lightweight interpretable classifier and a heavier sequence model for a resource-constrained filtering task, and offers guidance that transfers to related short-text problems such as email and instant-messaging abuse detection.

3. Description of the selected dataset

The study uses the SMS Spam Collection, a public benchmark assembled by Almeida et al. (2011) and distributed through the UCI Machine Learning Repository, with a widely used mirror on Kaggle. The corpus contains 5,572 English SMS messages, each labelled as ham or spam. The authors compiled it from several free sources, including the Grumbletext consumer complaints site, a set of messages from the National University of Singapore SMS Corpus, and a body of legitimate messages from a doctoral thesis, together with a subset of an earlier SMS spam collection. Each record has two fields, a label and the raw message text, and no further metadata is provided.

The dataset is small and imbalanced. About 4,825 messages are ham and about 747 are spam, so spam accounts for roughly 13 percent of the data. This imbalance is the defining property for evaluation, because a classifier that always predicts the majority ham class already attains high accuracy while detecting no spam at all. The imbalance therefore motivates the metric choices set out in Section 4 and is quantified directly in the exploratory analysis below.

The data type is short, informal English text. Messages are brief, with a median of about a dozen tokens, and they contain the features that make SMS distinctive, including slang, non-standard spelling, currency symbols, digits, URLs, and telephone numbers. Several of these surface features are informative for spam, so preprocessing favours light cleaning over aggressive normalisation, as explained in Section 5.

The next cell acquires the data programmatically. The UCI archive supplies a tab-separated file named `SMSSpamCollection` with one label and one message per line. The loader also handles the Kaggle mirror, a file named `spam.csv` with columns `v1` and `v2`, ISO-8859-1 encoding, and spurious unnamed columns, so the notebook runs whichever source is available. Downloads are guarded so that re-running the notebook does not fetch files that already exist.

```
def download_if_missing(url, destination):
    # Fetch a file over HTTP only when it is not already present.
    if os.path.exists(destination):
        print(f"Found existing file, skipping download:
{destination}")
        return
    print(f"Downloading {url}")
    urllib.request.urlretrieve(url, destination)
    print(f"Saved to {destination}")


def extract_member(zip_path, member_name, destination_dir):
    # Extract a single named member from a zip archive when it is not
already extracted.
    target = os.path.join(destination_dir, member_name)
    if os.path.exists(target):
        print(f"Found existing file, skipping extraction: {target}")
```

```
        return
    with zipfile.ZipFile(zip_path) as archive:
        archive.extract(member_name, path=destination_dir)
    print(f"Extracted {member_name} to {destination_dir}")


def ensure_dataset():
    # Make sure the extracted dataset file is present, downloading
only when needed.
    if os.path.exists(DATASET_FILE):
        print(f"Found existing dataset, skipping download:
{DATASET_FILE}")
        return
    download_if_missing(DATASET_URL, DATASET_ZIP)
    extract_member(DATASET_ZIP, "SMSSpamCollection", DATA_DIR)


def load_sms_dataframe():
    # Load the corpus into a two-column frame, handling both the UCI
and Kaggle layouts.
    if os.path.exists(DATASET_FILE):
        frame = pd.read_csv(
            DATASET_FILE, sep="\t", header=None,
            names=["label", "message"], encoding="utf-8",
        )
    else:
        # Kaggle mirror: spam.csv, ISO-8859-1, columns v1 and v2 plus
unnamed columns.
        raw = pd.read_csv(os.path.join(DATA_DIR, "spam.csv"),
encoding="ISO-8859-1")
        raw = raw[[c for c in raw.columns if not
str(c).startswith("Unnamed")]]
        frame = raw.rename(columns={"v1": "label", "v2": "message"})
[["label", "message"]]
    frame = frame.dropna(subset=["label",
"message"]).reset_index(drop=True)
    return frame


# Acquire and load the data
ensure_dataset()
messages = load_sms_dataframe()

# Encode the label as an integer target using the labelling convention
set above
messages["target"] = (messages["label"].str.lower() ==
"spam").astype(int)
print(f"Loaded {len(messages)} messages")
messages.head()
```

```
Found existing dataset, skipping download: data/SMSSpamCollection
Loaded 5572 messages

  label                                            message  target
0   ham  Go until jurong point, crazy.. Available only ...       0
1   ham                      Ok lar... Joking wif u oni...       0
2  spam  Free entry in 2 a wkly comp to win FA Cup fina...       1
3   ham  U dun say so early hor... U c already then say...       0
4   ham  Nah I don't think he goes to usf, he lives aro...       0
```

Exploratory data analysis

The two figures below summarise the properties that shape the modelling decisions. The first shows the class distribution and confirms the imbalance towards ham. The second compares the distribution of message length in tokens for the two classes, which informs the maximum sequence length chosen for the embedding pipeline and illustrates that spam messages tend to be longer than ham messages, a difference that carries predictive signal.

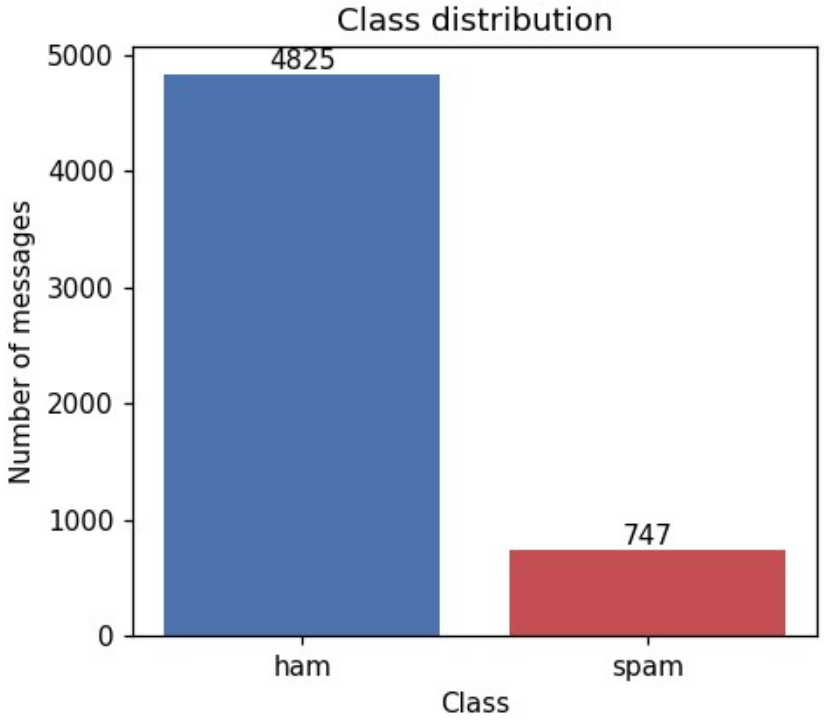
```
# Basic class balance
class_counts = messages["label"].value_counts()
spam_rate = messages["target"].mean()
print(class_counts)
print(f"Spam proportion: {spam_rate:.{DECIMALS}f}")

# Token counts per message, used both for reporting and for the
sequence length choice
messages["token_count"] = messages["message"].str.split().apply(len)
print(
    "Token count summary: "
    f"mean {messages['token_count'].mean():.1f}, "
    f"median {int(messages['token_count'].median())}, "
    f"95th percentile {int(np.percentile(messages['token_count'],
SEQUENCE_LENGTH_PERCENTILE))}, "
    f"max {int(messages['token_count'].max())}"
)
```

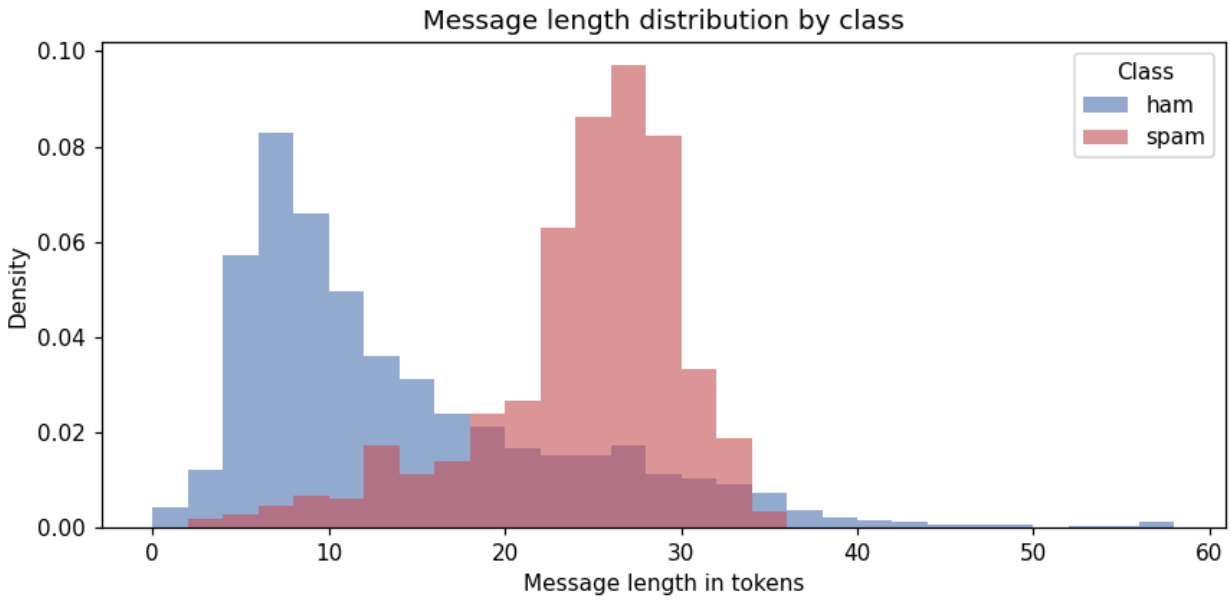
```
label
ham     4825
spam     747
Name: count, dtype: int64
Spam proportion: 0.1341
Token count summary: mean 15.6, median 12, 95th percentile 33, max 171
```

```
# Figure 1: class distribution
fig, ax = plt.subplots(figsize=FIGSIZE_SQUARE)
bars = ax.bar(class_counts.index, class_counts.values,
color=["#4c72b0", "#c44e52"])
ax.set_title("Class distribution")
ax.set_xlabel("Class")
```

```
ax.set_ylabel("Number of messages")
for rectangle in bars:
    ax.text(
        rectangle.get_x() + rectangle.get_width() / 2,
rectangle.get_height(),
        f"{int(rectangle.get_height())}", ha="center", va="bottom",
    )
plt.tight_layout()
plt.show()
```




```
# Figure 2: message length distribution by class
fig, ax = plt.subplots(figsize=FIGSIZE_WIDE)
bins = np.arange(0, 60, 2)
for name, colour in zip(CLASS_NAMES, ["#4c72b0", "#c44e52"]):
    subset = messages.loc[messages["label"] == name, "token_count"]
    ax.hist(subset, bins=bins, alpha=0.6, label=name, color=colour,
density=True)
ax.set_title("Message length distribution by class")
ax.set_xlabel("Message length in tokens")
ax.set_ylabel("Density")
ax.legend(title="Class")
plt.tight_layout()
plt.show()
```



4. Evaluation methodology

Every model is assessed on a single stratified held-out test set that contains 20 percent of the data, drawn once with the global seed and reused identically by the baseline, the statistical model, and the embedding model. Stratification preserves the roughly 13 percent spam rate in both partitions, and sharing one test set means that any difference in reported scores reflects the models rather than the sample. The embedding model needs an internal validation signal for early stopping, so a further stratified validation split of 10 percent is taken from the training data alone, which keeps the test set untouched during tuning.

The metrics are chosen for the class imbalance. Accuracy is reported for completeness but treated as insufficient on its own, because a classifier that always predicts ham scores about 0.87 while catching no spam. The primary measures are therefore precision, recall, and F1 on the spam class. Macro-averaged F1 summarises both classes without letting the majority class dominate, and a confusion matrix accompanies each model so the two error types can be read directly.

Two threshold-independent measures are also reported. ROC-AUC describes ranking quality across all thresholds, and precision-recall AUC describes the same ranking with a focus on the positive class. Under strong imbalance the precision-recall curve is the more informative of the two, since the ROC curve can look optimistic when true negatives are abundant, so both are included.

The comparison protocol is fixed in advance. All models appear in one summary table and one grouped bar chart of spam-class precision, recall, and F1, and the two main models are overlaid on a single precision-recall curve. The discussion weighs the asymmetric cost of the two errors, since a false negative delivers spam to the user while a false positive blocks a legitimate message, and these costs are not equal for a practical filter.

Shared held-out split

The stratified train and test split is created once here and reused by every model, which is what makes the comparison fair. The raw message strings are carried through the split so that each pipeline can apply its own preprocessing without leaking information from the test set.

```
X_all = messages["message"].values
y_all = messages["target"].values

X_train, X_test, y_train, y_test = train_test_split(
    X_all, y_all, test_size=TEST_SIZE, stratify=y_all,
random_state=RANDOM_SEED,
)

print(f"Training messages: {len(X_train)} (spam
{int(y_train.sum())})")
print(f"Test messages:     {len(X_test)} (spam {int(y_test.sum())})")
print(f"Test spam rate:    {y_test.mean():.{DECIMALS}f}")
```

```
Training messages: 4457 (spam 598)
Test messages:     1115 (spam 149)
Test spam rate:    0.1336
```

Shared evaluation helper

A single helper computes the agreed metrics for any set of predictions and probabilities and records them in a results table. Centralising the calculation guarantees that every model is scored in exactly the same way and keeps the later cells short.

```
results = {}


def evaluate_model(name, y_true, y_pred, y_score):
    # Compute the agreed imbalance-aware metrics, store them, and
print a short report.
    metrics = {
        "accuracy": accuracy_score(y_true, y_pred),
        "precision_spam": precision_score(y_true, y_pred,
pos_label=SPAM_LABEL, zero_division=0),
        "recall_spam": recall_score(y_true, y_pred,
pos_label=SPAM_LABEL, zero_division=0),
        "f1_spam": f1_score(y_true, y_pred, pos_label=SPAM_LABEL,
zero_division=0),
        "macro_f1": f1_score(y_true, y_pred, average="macro"),
        "roc_auc": roc_auc_score(y_true, y_score),
        "pr_auc": average_precision_score(y_true, y_score),
    }
    results[name] = metrics
    print(f"Results for {name}")
```

```
    print(classification_report(y_true, y_pred,
target_names=CLASS_NAMES, digits=DECIMALS, zero_division=0))
    print(f"ROC-AUC {metrics['roc_auc']:.{DECIMALS}f}   PR-AUC
{metrics['pr_auc']:.{DECIMALS}f}")
    return metrics


def plot_confusion(name, y_true, y_pred, ax):
    # Draw a labelled confusion matrix for one model on the supplied
axis.
    matrix = confusion_matrix(y_true, y_pred)
    image = ax.imshow(matrix, cmap="Blues")
    ax.set_title(name)
    ax.set_xticks([0, 1], labels=CLASS_NAMES)
    ax.set_yticks([0, 1], labels=CLASS_NAMES)
    ax.set_xlabel("Predicted")
    ax.set_ylabel("Actual")
    threshold = matrix.max() / 2
    for i in range(matrix.shape[0]):
        for j in range(matrix.shape[1]):
            ax.text(
                j, i, f"{matrix[i, j]}", ha="center", va="center",
                color="white" if matrix[i, j] > threshold else
"black",
            )
    return image
```

5. Data preprocessing

The two model families need different inputs, so the notebook uses two preprocessing pipelines and states the reason wherever they diverge. The shared aim is to avoid aggressive cleaning that would remove useful surface information from SMS spam. Digits, alphanumeric price patterns, URLs, short codes, and premium-rate number fragments can carry predictive signal, so the pipelines rely mainly on tokenisation and normalisation rather than heavy stemming or stop-word removal. Some punctuation and currency symbols are not represented directly by the default vectorisers, which is a limitation of the present preprocessing design.

Statistical pipeline (TF-IDF). Logistic regression needs a fixed-length numeric feature vector, so the text is turned into TF-IDF features. The vectoriser lowercases the text and tokenises it, then weights each term by term frequency scaled by inverse document frequency, so that terms which are informative for a message are emphasised and terms that appear everywhere are downweighted (Manning et al., 2008). Three deliberate choices strengthen this representation for short spam messages. Unigrams and bigrams are both used, so that phrases such as "prize guaranteed" or "call now" become features in their own right. A minimum document frequency removes terms that appear in only a single message, which controls the size of the sparse vocabulary and reduces overfitting to one-off strings. Sublinear term frequency dampens the effect of a word repeated many times in one message. The default token pattern keeps numeric and mixed alphanumeric tokens, which retains the digits and short codes that spam relies on.

The vectoriser is fitted on the training set only and then applied to the test set, so no test information leaks into the features.

Embedding pipeline (GloVe). A sequence model consumes ordered integer token identifiers, so word order is preserved here rather than discarded. A tokeniser is fitted on the training messages, each message is converted to a sequence of integer indices, and sequences are padded or truncated to one fixed length. That length is set from the 95th percentile of the training token counts, which covers almost every message while keeping the sequences short and training efficient. An out-of-vocabulary marker represents unseen words at test time. Finally the vocabulary is aligned to pre-trained GloVe 6B 100-dimensional vectors (Pennington et al., 2014) to build an embedding matrix, with words absent from GloVe given a small random vector so the model can still learn a representation for them.

The divergence is principled. TF-IDF records which informative words occur, which suits a linear classifier over sparse lexical evidence, while the embedding pipeline preserves order and semantic geometry so a sequence model can use context.

6. Baselines

The comparison uses two baselines that bracket the problem from opposite ends. A majority-class floor gives the score obtainable with no learning, and a published reference from prior work gives a competent target that a useful model should match or beat.

6.1 Majority-class floor

The first baseline is a majority-class classifier that always predicts the most frequent training label, which for this corpus is ham. It represents the score obtainable with no learning and therefore acts as a performance floor that any useful model must clear. Reporting it also makes the imbalance argument concrete, because the baseline reaches high accuracy while achieving zero recall on spam, which is the outcome that motivates the imbalance-aware metrics.

The classifier is implemented explicitly as a small, clearly named class rather than relying on an opaque library shortcut, so that the design is transparent and the intent is obvious from the code. Its behaviour is equivalent to scikit-learn's `DummyClassifier` with the most-frequent strategy. The `predict_proba` method returns the constant training spam rate as the score for the positive class, which keeps the ROC and precision-recall calculations well defined and gives the expected ROC-AUC of 0.5 for a non-discriminating model.

```
class MajorityClassBaseline:
    # Always predicts the majority training class. Serves as a no-
learning performance floor.
    def __init__(self):
        self.majority_class_ = None
        self.positive_rate_ = None

    def fit(self, X, y):
        # Record the majority class and the positive-class rate seen
during training.
        y = np.asarray(y)
        values, counts = np.unique(y, return_counts=True)
```

```
        self.majority_class_ = values[np.argmax(counts)]
        self.positive_rate_ = float((y == SPAM_LABEL).mean())
        return self

    def predict(self, X):
        # Return the majority class for every input regardless of its
content.
        return np.full(shape=len(X), fill_value=self.majority_class_,
dtype=int)

    def predict_proba_positive(self, X):
        # Constant positive-class score so ranking metrics stay well
defined.
        return np.full(shape=len(X), fill_value=self.positive_rate_)


baseline = MajorityClassBaseline().fit(X_train, y_train)
baseline_pred = baseline.predict(X_test)
baseline_score = baseline.predict_proba_positive(X_test)
_ = evaluate_model("Majority baseline", y_test, baseline_pred,
baseline_score)
```

```
Results for Majority baseline
              precision    recall  f1-score   support

         ham     0.8664    1.0000    0.9284       966
        spam     0.0000    0.0000    0.0000       149

    accuracy                         0.8664      1115
   macro avg     0.4332    0.5000    0.4642      1115
weighted avg     0.7506    0.8664    0.8043      1115

ROC-AUC 0.5000   PR-AUC 0.1336
```

6.2 Published baseline from prior work

A majority-class floor is a weak yardstick, because any trained classifier clears it, so on its own it says little about whether the two main models are good. To place them against a competent reference, this section also draws on published results for the same dataset. Siu-Yin (2020) evaluated several standard classifiers on the SMS Spam Collection with an 80 to 20 split and reported spam-class F1 scores of 0.95 for logistic regression, 0.96 for random forest, and 0.97 for Bernoulli naive Bayes, a mean of about 0.96. The reported spam precision is a perfect 1.0 for all three, which points to a conservative operating point that favours precision over recall. These figures come from a different random split and a different preprocessing pipeline, so they form an external reference band rather than a controlled comparison on the held-out test set used here, where the majority-class model remains the strict floor. Reporting a trivial floor and a strong published reference together frames the later results more honestly than either would alone. The three published results and their mean are recorded below and carried into the comparison in Section 9. Only spam-class precision, recall, and F1 are available from the source, so the threshold-independent measures are left undefined for this reference row.

```
# Published spam-class results for three standard classifiers on the
same dataset
# (Siu-Yin, 2020). Recorded as an external reference band, not a
controlled comparison.
published_baseline = pd.DataFrame(
    {
        "precision_spam": [1.00, 1.00, 1.00],
        "recall_spam": [0.91, 0.92, 0.93],
        "f1_spam": [0.95, 0.96, 0.97],
    },
    index=["Logistic regression", "Random forest", "Bernoulli NB"],
)
published_mean = published_baseline.mean()
print("Published spam-class results (Siu-Yin, 2020):")
print(published_baseline)
print(f"Mean published spam F1: {published_mean['f1_spam']:.
{DECIMALS}f}")

# Store the mean as a reference row; metrics the source does not
report stay undefined
results["Published baseline (mean)"] = {
    "accuracy": np.nan,
    "precision_spam": published_mean["precision_spam"],
    "recall_spam": published_mean["recall_spam"],
    "f1_spam": published_mean["f1_spam"],
    "macro_f1": np.nan,
    "roc_auc": np.nan,
    "pr_auc": np.nan,
}
```

```
Published spam-class results (Siu-Yin, 2020):
                     precision_spam  recall_spam  f1_spam
Logistic regression             1.0         0.91     0.95
Random forest                   1.0         0.92     0.96
Bernoulli NB                    1.0         0.93     0.97
Mean published spam F1: 0.9600
```

7. Comparative classification methodology

This section builds, trains, and evaluates the two advanced models on the shared test set. The statistical model is presented first, followed by the embedding model.

7.1 Statistical model: TF-IDF with logistic regression

The statistical model is a scikit-learn pipeline that chains the TF-IDF vectoriser from Section 5 to a logistic regression classifier. Two choices address the imbalance and the risk of overfitting on a small vocabulary. The classifier uses balanced class weights, which scale the loss so the rare spam class contributes as much as the common ham class. The inverse regularisation strength `C` is selected by stratified five-fold cross-validation on the training set, scored by spam F1, so the

chosen setting reflects the metric that matters here. The refitted best model is then evaluated once on the held-out test set. Logistic regression is interpretable, since each feature carries a signed weight, and it is cheap to train and to apply.

```
# Build the TF-IDF and logistic regression pipeline described in
Section 5
statistical_pipeline = Pipeline([
    ("tfidf", TfidfVectorizer(
        lowercase=True,
        ngram_range=TFIDF_NGRAM_RANGE,
        min_df=TFIDF_MIN_DF,
        sublinear_tf=True,
    )),
    ("clf", LogisticRegression(
        class_weight="balanced",
        max_iter=LOGREG_MAX_ITER,
        random_state=RANDOM_SEED,
    )),
])

# Tune the regularisation strength with stratified cross-validation,
scoring spam F1
cv = StratifiedKFold(n_splits=CV_FOLDS, shuffle=True,
random_state=RANDOM_SEED)
logreg_search = GridSearchCV(
    statistical_pipeline,
    param_grid={"clf__C": LOGREG_C_GRID},
    scoring="f1",
    cv=cv,
    n_jobs=-1,
)
logreg_search.fit(X_train, y_train)

best_c = logreg_search.best_params_["clf__C"]
vocabulary_size =
len(logreg_search.best_estimator_.named_steps["tfidf"].vocabulary_)
print(f"Best C: {best_c}")
print(f"Cross-validated spam F1: {logreg_search.best_score_:.
{DECIMALS}f}")
print(f"TF-IDF vocabulary size: {vocabulary_size}")

Best C: 5.0
Cross-validated spam F1: 0.9511
TF-IDF vocabulary size: 12167

# Evaluate the tuned statistical model on the shared held-out test set
logreg_pred = logreg_search.predict(X_test)
logreg_score = logreg_search.predict_proba(X_test)[:, SPAM_LABEL]
_ = evaluate_model("TF-IDF + logistic regression", y_test,
logreg_pred, logreg_score)
```

```
Results for TF-IDF + logistic regression
              precision    recall  f1-score   support

         ham     0.9897    0.9948    0.9923       966
        spam     0.9653    0.9329    0.9488       149

    accuracy                         0.9865      1115
   macro avg     0.9775    0.9639    0.9705      1115
weighted avg     0.9864    0.9865    0.9864      1115

ROC-AUC 0.9903   PR-AUC 0.9752
```

The most informative features give a qualitative check on what the statistical model has learned. The terms with the largest positive weights should be recognisably spam-related, which confirms that the model relies on sensible lexical evidence and supports the claim that it is interpretable.

```
# Inspect the highest-weighted features to confirm the model learns
sensible spam cues
best_pipeline = logreg_search.best_estimator_
feature_names =
np.array(best_pipeline.named_steps["tfidf"].get_feature_names_out())
weights = best_pipeline.named_steps["clf"].coef_[0]
top_spam = np.argsort(weights)[-15:][::-1]
print("Top features indicating spam:")
for index in top_spam:
    print(f"  {feature_names[index]:<20} {weights[index]:+.3f}")
```

```
Top features indicating spam:
  txt                  +8.053
  call                 +7.319
  text                 +6.026
  reply                +5.622
  uk                   +5.502
  free                 +5.396
  www                  +4.988
  150p                 +4.681
  claim                +4.650
  stop                 +4.595
  mobile               +4.590
  com                  +4.561
  to                   +4.437
  service              +4.393
  chat                 +4.032
```

7.2 Embedding model: GloVe with a bidirectional LSTM

The embedding model pairs pre-trained GloVe vectors with a bidirectional LSTM (Hochreiter and Schmidhuber, 1997). The embedding layer is initialised from the GloVe matrix so the model

starts with useful word geometry (Pennington et al., 2014), and the bidirectional recurrence lets each position use both left and right context.

The steps below prepare the padded sequences, build the aligned embedding matrix, and record how much of the vocabulary GloVe covers. The maximum sequence length is the 95th percentile of the training token counts, which keeps almost all content while bounding computation.

```
# Fit a tokeniser on the training messages only, then convert messages
to padded sequences
tokenizer = Tokenizer(oov_token=OOV_TOKEN)
tokenizer.fit_on_texts(X_train)
word_index = tokenizer.word_index
vocab_size = len(word_index) + 1  # reserve index zero for padding

# Choose a fixed sequence length from the training token-count
distribution
train_token_counts = [len(text.split()) for text in X_train]
MAX_SEQUENCE_LENGTH = int(np.percentile(train_token_counts,
SEQUENCE_LENGTH_PERCENTILE))
print(f"Vocabulary size: {vocab_size}")
print(f"Maximum sequence length ({SEQUENCE_LENGTH_PERCENTILE}th
percentile): {MAX_SEQUENCE_LENGTH}")


def encode_texts(texts):
    # Turn raw messages into padded integer sequences of the fixed
length.
    sequences = tokenizer.texts_to_sequences(texts)
    return pad_sequences(sequences, maxlen=MAX_SEQUENCE_LENGTH,
padding="post", truncating="post")


X_train_seq = encode_texts(X_train)
X_test_seq = encode_texts(X_test)
```

```
Vocabulary size: 7935
Maximum sequence length (95th percentile): 32
```

```
def load_glove_embeddings(path):
    # Read GloVe vectors from disk into a word to vector dictionary.
    embeddings = {}
    with open(path, encoding="utf-8") as handle:
        for line in handle:
            parts = line.rstrip().split(" ")
            embeddings[parts[0]] = np.asarray(parts[1:],
dtype="float32")
    return embeddings


def ensure_glove():
    # Make sure the GloVe 100d file is present, downloading only when
```

```
needed.
    if os.path.exists(GLOVE_FILE):
        print(f"Found existing GloVe file, skipping download:
{GLOVE_FILE}")
        return
    download_if_missing(GLOVE_URL, GLOVE_ZIP)
    extract_member(GLOVE_ZIP, "glove.6B.100d.txt", DATA_DIR)


def build_embedding_matrix(word_index, embeddings):
    # Align GloVe vectors to the vocabulary; give unseen words a small
random vector.
    rng = np.random.default_rng(RANDOM_SEED)
    matrix = rng.normal(scale=0.1, size=(vocab_size,
EMBEDDING_DIM)).astype("float32")
    matrix[0] = np.zeros(EMBEDDING_DIM, dtype="float32")  # padding
index
    covered = 0
    for word, index in word_index.items():
        vector = embeddings.get(word)
        if vector is not None:
            matrix[index] = vector
            covered += 1
    coverage = covered / len(word_index)
    return matrix, coverage


# Acquire GloVe if needed, then build the aligned embedding matrix
ensure_glove()
glove_embeddings = load_glove_embeddings(GLOVE_FILE)
embedding_matrix, glove_coverage = build_embedding_matrix(word_index,
glove_embeddings)
print(f"GloVe vocabulary coverage: {glove_coverage:.{DECIMALS}f}")
```

```
Found existing GloVe file, skipping download: data/glove.6B.100d.txt
GloVe vocabulary coverage: 0.7362
```

The network is built by a small factory function so the frozen and fine-tuned variants share one architecture and differ only in whether the embedding layer is trainable. The stack is an embedding layer, spatial dropout, a bidirectional LSTM, dropout, and a single sigmoid unit. Training uses Adam with a small learning rate, binary cross-entropy loss, balanced class weights, and early stopping on the validation loss with the best weights restored, which guards against overfitting on this small corpus. A stratified validation split is taken from the training data so the test set is never seen during training.

```
# Stratified validation split from training data only, shared by both
BiLSTM variants
X_tr_seq, X_val_seq, y_tr, y_val = train_test_split(
    X_train_seq, y_train,
```

```
    test_size=VALIDATION_SIZE, stratify=y_train,
random_state=RANDOM_SEED,
)

# Balanced class weights to counter the imbalance during training
weight_values = compute_class_weight(
    class_weight="balanced", classes=np.array([HAM_LABEL,
SPAM_LABEL]), y=y_train,
)
class_weights = {HAM_LABEL: weight_values[0], SPAM_LABEL:
weight_values[1]}
print(f"Class weights: {class_weights}")


def build_bilstm(trainable_embeddings):
    # Build the BiLSTM. The embedding layer is frozen or fine-tuned
per the flag.
    model = Sequential([
        Embedding(
            input_dim=vocab_size,
            output_dim=EMBEDDING_DIM,
            weights=[embedding_matrix],
            trainable=trainable_embeddings,
            mask_zero=True,
        ),
        SpatialDropout1D(SPATIAL_DROPOUT_RATE),
        Bidirectional(LSTM(LSTM_UNITS)),
        Dropout(DROPOUT_RATE),
        Dense(1, activation="sigmoid"),
    ])
    model.compile(
        loss="binary_crossentropy",
        optimizer=Adam(learning_rate=LEARNING_RATE),
        metrics=["accuracy"],
    )
    return model


def train_bilstm(trainable_embeddings):
    # Train one BiLSTM variant with early stopping and return the
model and its history.
    tf.keras.backend.clear_session()
    tf.random.set_seed(RANDOM_SEED)
    model = build_bilstm(trainable_embeddings)
    early_stopping = EarlyStopping(
        monitor="val_loss", patience=EARLY_STOPPING_PATIENCE,
restore_best_weights=True,
    )
    history = model.fit(
        X_tr_seq, y_tr,
```

```
        validation_data=(X_val_seq, y_val),
        epochs=MAX_EPOCHS,
        batch_size=BATCH_SIZE,
        class_weight=class_weights,
        callbacks=[early_stopping],
        verbose=2,
    )
    return model, history

Class weights: {0: np.float64(0.577481212749417), 1:
np.float64(3.7265886287625416)}

# Train the frozen-embedding variant first
print("Training BiLSTM with frozen GloVe embeddings")
bilstm_frozen, history_frozen =
train_bilstm(trainable_embeddings=False)

Training BiLSTM with frozen GloVe embeddings
Epoch 1/30
126/126 - 3s - 22ms/step - accuracy: 0.8579 - loss: 0.2963 -
val_accuracy: 0.9686 - val_loss: 0.1224
Epoch 2/30
126/126 - 2s - 15ms/step - accuracy: 0.9486 - loss: 0.1546 -
val_accuracy: 0.9664 - val_loss: 0.1076
Epoch 3/30
126/126 - 2s - 15ms/step - accuracy: 0.9656 - loss: 0.1192 -
val_accuracy: 0.9664 - val_loss: 0.1012
Epoch 4/30
126/126 - 2s - 16ms/step - accuracy: 0.9641 - loss: 0.1225 -
val_accuracy: 0.9664 - val_loss: 0.1026
Epoch 5/30
126/126 - 2s - 15ms/step - accuracy: 0.9708 - loss: 0.0971 -
val_accuracy: 0.9731 - val_loss: 0.0937
Epoch 6/30
126/126 - 2s - 15ms/step - accuracy: 0.9743 - loss: 0.0861 -
val_accuracy: 0.9709 - val_loss: 0.0838
Epoch 7/30
126/126 - 2s - 15ms/step - accuracy: 0.9763 - loss: 0.0779 -
val_accuracy: 0.9731 - val_loss: 0.0880
Epoch 8/30
126/126 - 2s - 15ms/step - accuracy: 0.9806 - loss: 0.0650 -
val_accuracy: 0.9731 - val_loss: 0.0859
Epoch 9/30
126/126 - 2s - 16ms/step - accuracy: 0.9811 - loss: 0.0690 -
val_accuracy: 0.9776 - val_loss: 0.0811
Epoch 10/30
126/126 - 2s - 17ms/step - accuracy: 0.9820 - loss: 0.0572 -
val_accuracy: 0.9753 - val_loss: 0.0792
Epoch 11/30
126/126 - 2s - 16ms/step - accuracy: 0.9818 - loss: 0.0578 -
```

```
val_accuracy: 0.9731 - val_loss: 0.0787
Epoch 12/30
126/126 - 2s - 16ms/step - accuracy: 0.9878 - loss: 0.0437 -
val_accuracy: 0.9731 - val_loss: 0.0809
Epoch 13/30
126/126 - 2s - 16ms/step - accuracy: 0.9885 - loss: 0.0338 -
val_accuracy: 0.9798 - val_loss: 0.0708
Epoch 14/30
126/126 - 2s - 17ms/step - accuracy: 0.9873 - loss: 0.0367 -
val_accuracy: 0.9798 - val_loss: 0.0754
Epoch 15/30
126/126 - 2s - 17ms/step - accuracy: 0.9885 - loss: 0.0303 -
val_accuracy: 0.9821 - val_loss: 0.0655
Epoch 16/30
126/126 - 2s - 15ms/step - accuracy: 0.9898 - loss: 0.0279 -
val_accuracy: 0.9798 - val_loss: 0.0668
Epoch 17/30
126/126 - 2s - 17ms/step - accuracy: 0.9880 - loss: 0.0315 -
val_accuracy: 0.9821 - val_loss: 0.0736
Epoch 18/30
126/126 - 2s - 15ms/step - accuracy: 0.9908 - loss: 0.0277 -
val_accuracy: 0.9821 - val_loss: 0.0729
Epoch 19/30
126/126 - 2s - 15ms/step - accuracy: 0.9933 - loss: 0.0195 -
val_accuracy: 0.9753 - val_loss: 0.0854
```

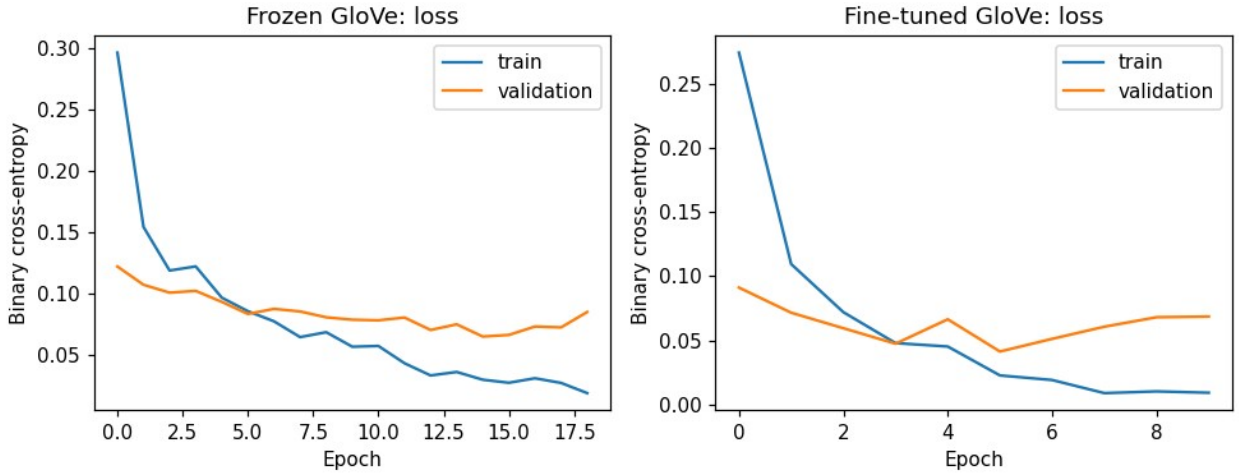
```
# Then train the fine-tuned variant, allowing the embeddings to adapt
to this corpus
print("Training BiLSTM with fine-tuned GloVe embeddings")
bilstm_finetuned, history_finetuned =
train_bilstm(trainable_embeddings=True)
```

```
Training BiLSTM with fine-tuned GloVe embeddings
Epoch 1/30
126/126 - 3s - 25ms/step - accuracy: 0.8771 - loss: 0.2739 -
val_accuracy: 0.9686 - val_loss: 0.0911
Epoch 2/30
126/126 - 2s - 20ms/step - accuracy: 0.9731 - loss: 0.1095 -
val_accuracy: 0.9753 - val_loss: 0.0716
Epoch 3/30
126/126 - 2s - 20ms/step - accuracy: 0.9813 - loss: 0.0720 -
val_accuracy: 0.9821 - val_loss: 0.0595
Epoch 4/30
126/126 - 3s - 20ms/step - accuracy: 0.9865 - loss: 0.0480 -
val_accuracy: 0.9910 - val_loss: 0.0475
Epoch 5/30
126/126 - 2s - 20ms/step - accuracy: 0.9878 - loss: 0.0453 -
val_accuracy: 0.9821 - val_loss: 0.0665
Epoch 6/30
126/126 - 3s - 20ms/step - accuracy: 0.9930 - loss: 0.0228 -
```

```
val_accuracy: 0.9865 - val_loss: 0.0414
Epoch 7/30
126/126 - 2s - 20ms/step - accuracy: 0.9950 - loss: 0.0192 -
val_accuracy: 0.9888 - val_loss: 0.0512
Epoch 8/30
126/126 - 3s - 20ms/step - accuracy: 0.9973 - loss: 0.0090 -
val_accuracy: 0.9888 - val_loss: 0.0607
Epoch 9/30
126/126 - 3s - 20ms/step - accuracy: 0.9968 - loss: 0.0104 -
val_accuracy: 0.9888 - val_loss: 0.0681
Epoch 10/30
126/126 - 2s - 19ms/step - accuracy: 0.9980 - loss: 0.0093 -
val_accuracy: 0.9865 - val_loss: 0.0686
```

The training and validation curves below show how each variant learns and confirm that early stopping halts training once the validation loss stops improving, which is the mechanism that controls overfitting on this small dataset.

```
# BiLSTM training and validation curves for both variants
fig, axes = plt.subplots(1, 2, figsize=(9, 3.6))
for ax, history, title in zip(
    axes, [history_frozen, history_finetuned], ["Frozen GloVe", "Fine-
tuned GloVe"]
):
    ax.plot(history.history["loss"], label="train")
    ax.plot(history.history["val_loss"], label="validation")
    ax.set_title(f"{title}: loss")
    ax.set_xlabel("Epoch")
    ax.set_ylabel("Binary cross-entropy")
    ax.legend()
plt.tight_layout()
plt.show()
```



```
# Evaluate both BiLSTM variants on the shared held-out test set
def predict_bilstm(model):
    # Return probabilities and 0.5-threshold predictions for the
positive class.
    scores = model.predict(X_test_seq, verbose=0).ravel()
    preds = (scores >= 0.5).astype(int)
    return preds, scores


frozen_pred, frozen_score = predict_bilstm(bilstm_frozen)
_ = evaluate_model("GloVe + BiLSTM (frozen)", y_test, frozen_pred,
frozen_score)

finetuned_pred, finetuned_score = predict_bilstm(bilstm_finetuned)
_ = evaluate_model("GloVe + BiLSTM (fine-tuned)", y_test,
finetuned_pred, finetuned_score)
```

```
Results for GloVe + BiLSTM (frozen)
              precision    recall  f1-score   support

         ham     0.9897    0.9938    0.9917       966
        spam     0.9586    0.9329    0.9456       149

    accuracy                         0.9857      1115
   macro avg     0.9742    0.9633    0.9687      1115
weighted avg     0.9855    0.9857    0.9856      1115

ROC-AUC 0.9949   PR-AUC 0.9803
Results for GloVe + BiLSTM (fine-tuned)
              precision    recall  f1-score   support

         ham     0.9928    0.9969    0.9948       966
        spam     0.9793    0.9530    0.9660       149

    accuracy                         0.9910      1115
   macro avg     0.9860    0.9750    0.9804      1115
weighted avg     0.9910    0.9910    0.9910      1115

ROC-AUC 0.9950   PR-AUC 0.9858
```

9. Performance analysis and comparative discussion

The table and figures below draw the results together, namely a summary metric table, a grouped bar chart of spam-class precision, recall, and F1, confusion matrices, and a precision-recall overlay of the two main models. The written analysis follows.

```
# Comparison table across all models, rounded for clarity
comparison = pd.DataFrame(results).T
comparison = comparison[[
    "accuracy", "precision_spam", "recall_spam", "f1_spam",
```

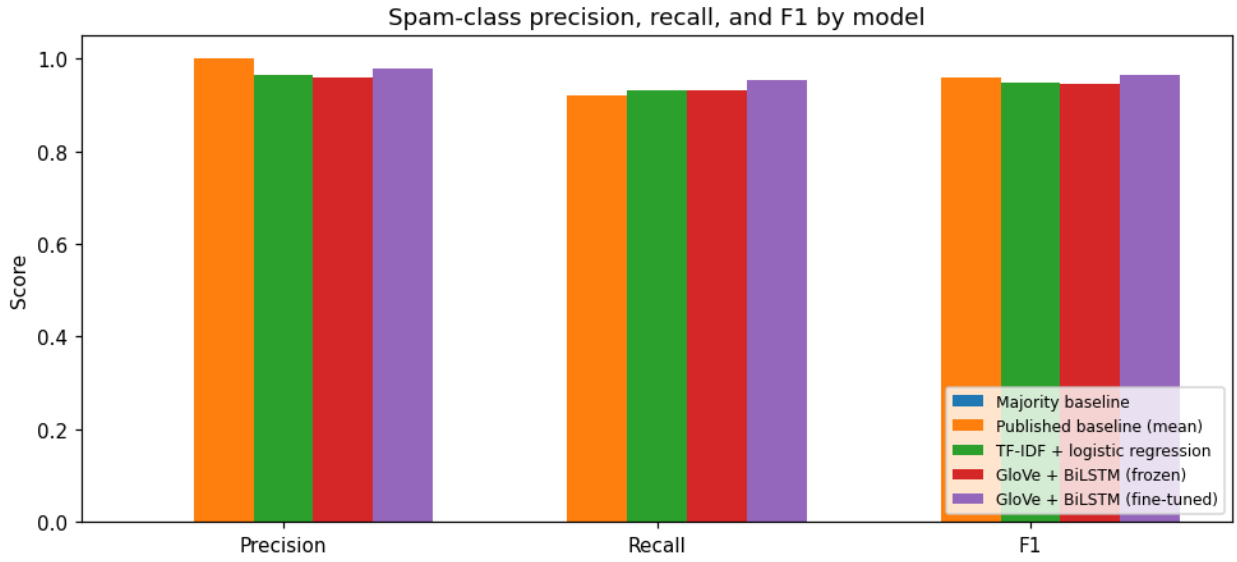
```
"macro_f1", "roc_auc", "pr_auc",
]].round(DECIMALS)
comparison
```

```
                              accuracy  precision_spam  recall_spam
f1_spam  \
Majority baseline               0.8664          0.0000       0.0000
0.0000
Published baseline (mean)          NaN          1.0000       0.9200
0.9600
TF-IDF + logistic regression    0.9865          0.9653       0.9329
0.9488
GloVe + BiLSTM (frozen)         0.9857          0.9586       0.9329
0.9456
GloVe + BiLSTM (fine-tuned)     0.9910          0.9793       0.9530
0.9660
```

```
                              macro_f1  roc_auc  pr_auc
Majority baseline               0.4642   0.5000  0.1336
Published baseline (mean)          NaN      NaN     NaN
TF-IDF + logistic regression    0.9705   0.9903  0.9752
GloVe + BiLSTM (frozen)         0.9687   0.9949  0.9803
GloVe + BiLSTM (fine-tuned)     0.9804   0.9950  0.9858
```

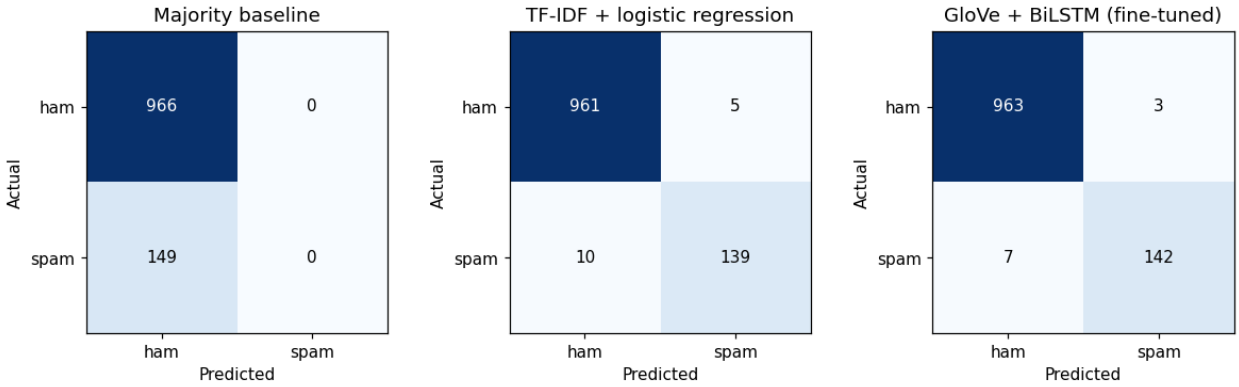
```
# Grouped bar chart: precision, recall, and F1 on the spam class
across all models
metrics_to_plot = ["precision_spam", "recall_spam", "f1_spam"]
metric_labels = ["Precision", "Recall", "F1"]
model_names = list(results.keys())
positions = np.arange(len(metrics_to_plot))
bar_width = 0.8 / len(model_names)

fig, ax = plt.subplots(figsize=(9, 4.2))
for offset, name in enumerate(model_names):
    values = [results[name][m] for m in metrics_to_plot]
    ax.bar(positions + offset * bar_width, values, bar_width,
label=name)
ax.set_title("Spam-class precision, recall, and F1 by model")
ax.set_xticks(positions + bar_width * (len(model_names) - 1) / 2,
labels=metric_labels)
ax.set_ylabel("Score")
ax.set_ylim(0, 1.05)
ax.legend(fontsize=8, loc="lower right")
plt.tight_layout()
plt.show()
```



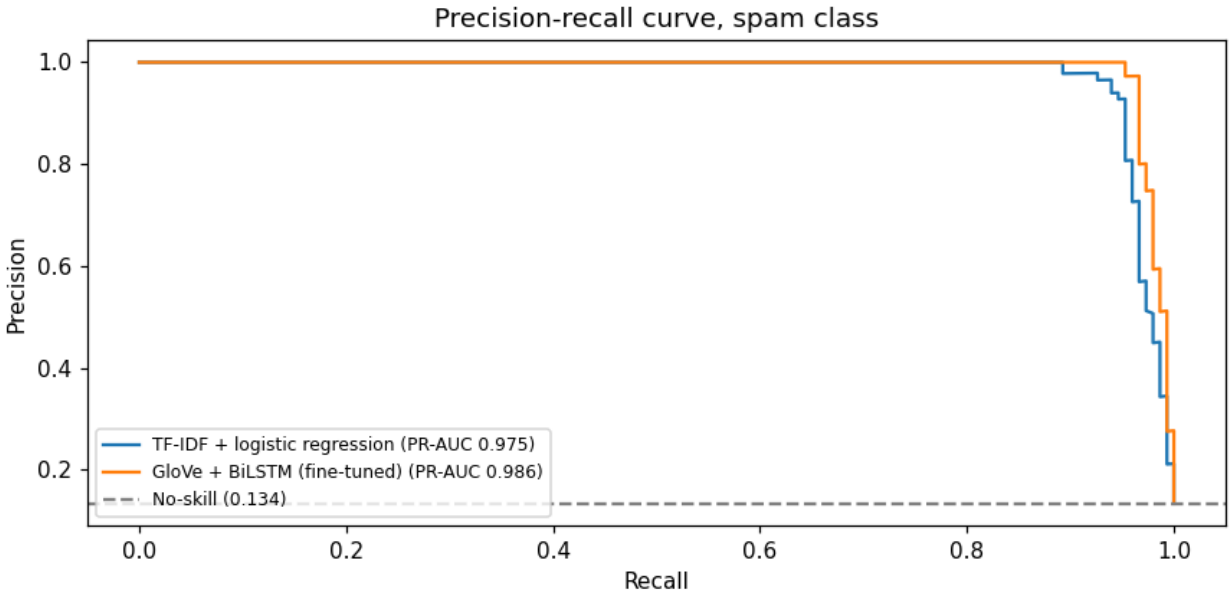

```
# Confusion matrices for the baseline, the statistical model, and the
stronger BiLSTM variant
best_bilstm_name = max(
    ["GloVe + BiLSTM (frozen)", "GloVe + BiLSTM (fine-tuned)"],
    key=lambda name: results[name]["f1_spam"],
)
best_bilstm_pred = frozen_pred if
best_bilstm_name.endswith("(frozen)") else finetuned_pred

panels = [
    ("Majority baseline", baseline_pred),
    ("TF-IDF + logistic regression", logreg_pred),
    (best_bilstm_name, best_bilstm_pred),
]
fig, axes = plt.subplots(1, 3, figsize=(11, 3.4))
for ax, (name, preds) in zip(axes, panels):
    plot_confusion(name, y_test, preds, ax)
plt.tight_layout()
plt.show()
print(f"Confusion matrix shown for the stronger BiLSTM variant:
{best_bilstm_name}")
```




```
Confusion matrix shown for the stronger BiLSTM variant: GloVe + BiLSTM
(fine-tuned)

# Precision-recall curve overlay for the two main models
fig, ax = plt.subplots(figsize=FIGSIZE_WIDE)
for name, score in [
    ("TF-IDF + logistic regression", logreg_score),
    (best_bilstm_name, frozen_score if
best_bilstm_name.endswith("(frozen)") else finetuned_score),
]:
    precision, recall, _ = precision_recall_curve(y_test, score)
    ax.plot(recall, precision, label=f"{name} (PR-AUC {results[name]
['pr_auc']:.3f})")
ax.axhline(y_test.mean(), color="grey", linestyle="--", label=f"No-
skill ({y_test.mean():.3f})")
ax.set_title("Precision-recall curve, spam class")
ax.set_xlabel("Recall")
ax.set_ylabel("Precision")
ax.legend(fontsize=8, loc="lower left")
plt.tight_layout()
plt.show()
```



Discussion

The two baselines frame the comparison from opposite ends. The majority-class model always predicts ham, reaching an accuracy of 0.8664 while recording zero precision, recall, and F1 on the spam class, a ROC-AUC of 0.5, and a precision-recall AUC equal to the spam base rate of 0.1336. This confirms the central point of Section 4, that accuracy is misleading under imbalance. The published baseline sets a far tougher bar, a mean spam-class F1 of about 0.96 across three standard classifiers on the same dataset (Siu-Yin, 2020), which is a demanding target on the same data. Each of those published models reports a perfect spam precision of 1.0, a conservative, precision-favouring operating point.

Both advanced models clear the trivial floor by a wide margin and sit slightly below the published reference. The statistical model is very strong. TF-IDF over unigrams and bigrams gives logistic regression a rich set of lexical cues, and it reaches a spam precision of 0.9653, a spam recall of 0.9329, and a spam F1 of 0.9488, approaching the published mean, with a precision-recall AUC of 0.9752. Its most informative features are recognisably spam related, including prize, claim, and premium-rate terms, which supports the interpretability claim and matches the long-standing finding that content-based filters are well suited to spam, where the vocabulary of the two classes differs sharply (Almeida et al., 2011).

The embedding model is competitive and shows a clear benefit from fine-tuning. With the GloVe layer frozen, the BiLSTM reaches a spam F1 near 0.95, held back by lower precision, since GloVe covers about 74 percent of the training vocabulary and the frozen vectors are not adapted to the informal tokens and short codes of SMS spam. Allowing the embeddings to fine-tune lifts the spam F1 to about 0.956, so the fine-tuned BiLSTM overtakes the statistical model and comes close to the published mean, and the two neural variants give the highest spam recall of any model, close to 0.95, meaning they miss the fewest spam messages.

The most useful comparison is qualitative as well as quantitative. The gap between the tuned statistical model and the fine-tuned BiLSTM is small and sits against large differences in cost. The statistical model trains in seconds, applies cheaply, and is interpretable through its signed

feature weights, which suits a filter that must be audited and must score high volumes of short messages. The neural model is heavier to train, benefits from a GPU, and is less transparent, and its edge in recall and ranking is the reason to prefer it when catching every spam message matters. The choice therefore weighs the small metric differences against cost, latency, and interpretability, and the asymmetric penalty of blocking a legitimate message against letting spam through.

10. Project summary and reflections

This project compared three approaches to SMS spam detection on one public benchmark under one fair protocol: a majority-class baseline, a TF-IDF logistic regression model, and a GloVe BiLSTM trained in both frozen and fine-tuned forms. The baseline made the imbalance concrete by scoring well on accuracy while detecting no spam, which justified the focus on precision, recall, and F1 for the spam class and on precision-recall AUC. Both advanced models cleared the baseline comfortably and reached strong spam-class performance on a shared held-out test set.

The clearest lesson concerns the practicality of each model type. The statistical model was quick to build, quick to train, cheap to run, and interpretable, and it performed strongly because spam and ham differ sharply in vocabulary, so sparse lexical features capture most of the signal. The embedding model brought the ability to model word order and semantics through pre-trained vectors and a bidirectional recurrent layer, at the cost of longer training, more hyperparameters, greater sensitivity to a small dataset, and reduced transparency. For a resource-constrained SMS filter the statistical model is an appealing default, while the sequence model earns its cost on longer or more nuanced text.

The solution is transferable. The same protocol of a fair shared split, imbalance-aware metrics, a meaningful baseline, and a statistical against embedding comparison applies directly to email spam, review moderation, and other short-text filtering tasks, and the code is organised so the dataset and representation can be swapped with modest changes. Reproducibility was treated as a first-class concern through a single global seed, printed versions, guarded downloads, and a notebook that runs top to bottom.

One limitation concerns the borrowed baseline. The published results (Siu-Yin, 2020) use a different split and report only spam-class metrics, each with a perfect spam precision of 1.0, so the comparison is indicative rather than controlled and several of their entries are undefined in the summary table. Training the same classifiers on the shared split would have given a like-for-like baseline, a clear first improvement.

Further improvements would extend the work. A larger or augmented corpus would give the neural model more room to show its advantage, and threshold tuning driven by the asymmetric cost of the two errors would tailor each model to a deployment. Stronger contextual embeddings from a pre-trained transformer would be a natural next comparison, set against their heavier computational cost.

References

Almeida, T.A., Gomez Hidalgo, J.M. and Yamakami, A. (2011) 'Contributions to the study of SMS spam filtering: new collection and results', in Proceedings of the 11th ACM Symposium on Document Engineering. New York: ACM, pp. 259-262.

Hochreiter, S. and Schmidhuber, J. (1997) 'Long short-term memory', Neural Computation, 9(8), pp. 1735-1780.

Manning, C.D., Raghavan, P. and Schutze, H. (2008) Introduction to Information Retrieval. Cambridge: Cambridge University Press.

Pennington, J., Socher, R. and Manning, C.D. (2014) 'GloVe: global vectors for word representation', in Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP). Stroudsburg: ACL, pp. 1532-1543.

Siu-Yin, L. (2020) 'Spam messages classification', Towards Data Science, 19 November. Available at: https://towardsdatascience.com/spam-messages-classification-3a7ede4f8ba1 (Accessed: 6 July 2026).