---

1) Project chosen

Spam Email/SMS Detection — classify messages as spam or ham (not spam).

2) Dataset

Use the widely used SMS Spam Collection dataset (UCI / Kaggle).

Suggested dataset: "SMS Spam Collection" — contains labeled messages (ham, spam). (Search on Kaggle or UCI; if you want I can fetch a direct link.)

File expected: a CSV with two columns: label (ham/spam) and message.

---

3) Plan / Tasks mapping

Collect dataset: download CSV from Kaggle/UCI.

Preprocess + EDA:

Inspect class balance, message length distributions, most common words per class.

Clean text (lowercase, remove punctuation, URLs, emails, numbers), tokenize, remove stopwords, lemmatize/stem.

Feature extraction: TF-IDF (and try CountVectorizer as alternate).

Modeling:

Baseline: Multinomial Naive Bayes.

Stronger: Logistic Regression (with class weighting), RandomForest or XGBoost (optional).

Use train/test split and 5-fold cross-validation.

Evaluation: accuracy, precision, recall, F1, confusion matrix, ROC AUC (probabilities).

Report: Introduction, methodology, dataset, EDA, preprocessing, model selection, results, conclusion.

Deliver: code.py / notebook + report.pdf.

---

4) Ready-to-run Python code (single notebook/script)

> Copy this into a Jupyter notebook or spam_detection.py. If notebook, run cells; if script, adapt plotting (or save figures).

```python
# spam_detection.py (or notebook cells)
# Human-style, commented, step-by-step implementation

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns          # only for nicer plots (optional)
import re
import joblib                  # to save model
from sklearn.model_selection import train_test_split, cross_val_score, GridSearchCV
from sklearn.feature_extraction.text import TfidfVectorizer, CountVectorizer
from sklearn.pipeline import Pipeline
from sklearn.naive_bayes import MultinomialNB
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import (accuracy_score, precision_score, recall_score,
                    f1_score, confusion_matrix, classification_report,
                    roc_auc_score, roc_curve)
import nltk
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
```

```
# If nltk data not downloaded yet:

# nltk.download('stopwords')

# nltk.download('punkt')

# nltk.download('wordnet')


# 1. Load dataset (update path to your CSV)

# Expected CSV with columns: 'label' and 'message'

df = pd.read_csv('spam.csv', encoding='latin-
1')[['v1','v2']].rename(columns={'v1':'label','v2':'message'})

# If using SMS Spam Collection from UCI, v1/v2 are common column names. Adjust if
needed.


print("Dataset shape:", df.shape)

print(df.label.value_counts())


# 2. Quick EDA

df['msg_len'] = df['message'].apply(len)

print(df.groupby('label')['msg_len'].describe())


# Plot class balance

plt.figure(figsize=(5,4))

sns.countplot(x='label', data=df)

plt.title('Class distribution')

plt.savefig('class_distribution.png', bbox_inches='tight')


# Message length histograms
```

```
plt.figure(figsize=(8,4))
sns.histplot(data=df, x='msg_len', hue='label', bins=50, element='step', stat='density')
plt.title('Message length distribution by class')
plt.savefig('msglen_distribution.png', bbox_inches='tight')

# 3. Preprocessing function
stop_words = set(stopwords.words('english'))
lemmatizer = WordNetLemmatizer()

def clean_text(text):
  text = str(text).lower()
  # remove URLs, emails, numbers
  text = re.sub(r'http\S+|www\S+|https\S+', ' ', text)
  text = re.sub(r'\S+@\S+', ' ', text)
  text = re.sub(r'\d+', ' ', text)
  # remove punctuation
  text = re.sub(r'[^a-z\s]', ' ', text)
  # tokenize
  tokens = text.split()
  # remove stopwords and short words, lemmatize
  tokens = [lemmatizer.lemmatize(t) for t in tokens if t not in stop_words and len(t) > 2]
  return " ".join(tokens)

# Apply cleaning (this may take a moment)
df['clean'] = df['message'].apply(clean_text)
print(df[['message','clean']].head())
```

```
# 4. Train/test split
X = df['clean']
y = df['label'].map({'ham':0,'spam':1})  # convert labels to 0/1
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=42)

# 5. Baseline pipeline: TF-IDF + MultinomialNB
baseline_pipe = Pipeline([
  ('tfidf', TfidfVectorizer(max_df=0.9, min_df=2, ngram_range=(1,2))),
  ('clf', MultinomialNB())
])
baseline_pipe.fit(X_train, y_train)
y_pred = baseline_pipe.predict(X_test)
y_proba = baseline_pipe.predict_proba(X_test)[:,1]

print("Baseline (MultinomialNB) metrics:")
print("Accuracy:", accuracy_score(y_test, y_pred))
print("Precision:", precision_score(y_test, y_pred))
print("Recall:", recall_score(y_test, y_pred))
print("F1:", f1_score(y_test, y_pred))
print(classification_report(y_test, y_pred))

# Confusion matrix
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(4,3))
```

```
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['ham','spam'], yticklabels=['ham','spam'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix - Baseline')
plt.savefig('confusion_baseline.png', bbox_inches='tight')

# ROC AUC
roc_auc = roc_auc_score(y_test, y_proba)
print("ROC AUC:", roc_auc)
fpr, tpr, _ = roc_curve(y_test, y_proba)
plt.figure()
plt.plot(fpr, tpr)
plt.xlabel('FPR')
plt.ylabel('TPR')
plt.title('ROC curve - Baseline (AUC={:.3f})'.format(roc_auc))
plt.savefig('roc_baseline.png', bbox_inches='tight')

# 6. Stronger model: TF-IDF + Logistic Regression with GridSearchCV
pipe_lr = Pipeline([
    ('tfidf', TfidfVectorizer()),
    ('clf', LogisticRegression(max_iter=1000, class_weight='balanced', solver='liblinear'))
])

param_grid = {
    'tfidf__max_df': [0.9, 0.95],
```

```
    'tfidf__min_df': [1,2],
    'tfidf__ngram_range': [(1,1),(1,2)],
    'clf__C': [0.1, 1, 5]
}

grid = GridSearchCV(pipe_lr, param_grid, cv=5, scoring='f1', n_jobs=-1, verbose=1)
grid.fit(X_train, y_train)

print("Best params:", grid.best_params_)
best_lr = grid.best_estimator_

# Evaluate best LR
y_pred_lr = best_lr.predict(X_test)
y_proba_lr = best_lr.predict_proba(X_test)[:,1]
print("Logistic Regression metrics:")
print("Accuracy:", accuracy_score(y_test, y_pred_lr))
print("Precision:", precision_score(y_test, y_pred_lr))
print("Recall:", recall_score(y_test, y_pred_lr))
print("F1:", f1_score(y_test, y_pred_lr))
print("ROC AUC:", roc_auc_score(y_test, y_proba_lr))

print(classification_report(y_test, y_pred_lr))

# Save model
joblib.dump(best_lr, 'spam_detector_lr.joblib')
```

```
# 7. Feature importance (top tokens from logistic regression)
tfidf = best_lr.named_steps['tfidf']
clf = best_lr.named_steps['clf']
if hasattr(clf, 'coef_'):
    feature_names = tfidf.get_feature_names_out()
    coef = clf.coef_[0]
    top_pos = np.argsort(coef)[-20:]
    top_neg = np.argsort(coef)[:20]
    print("Top tokens indicating SPAM:")
    for i in top_pos[::-1]:
        print(feature_names[i], coef[i])
    print("\nTop tokens indicating HAM:")
    for i in top_neg:
        print(feature_names[i], coef[i])

# 8. Save a small report CSV with predictions for inspection
test_df = pd.DataFrame({'message': X_test, 'label': y_test, 'pred': y_pred_lr, 'prob_spam': y_proba_lr})
test_df.to_csv('test_predictions.csv', index=False)

print("Done. Models and images saved.")
```

Notes on code

Adjust the CSV filename/path to your downloaded dataset.

NLTK downloads: run nltk.download('stopwords') etc. once if needed.

TfidfVectorizer parameters were tuned with GridSearch; you can extend.

Model files and plots are saved (so you can include them in the report).

The pipeline saves preprocessor + model so you can deploy easily.

---

5) Interpretations & Example results (what to expect)

Baseline MultinomialNB usually gives high recall for spam but sometimes lower precision (depends on dataset).

Logistic Regression with TF-IDF and simple cleaning typically yields F1 > 0.95 on this SMS dataset (because it’s small and clean).

Important tokens for spam usually include words like free, win, claim, urgent, numbers (discounts) — after cleaning you’ll see these.

Confusion matrix: false negatives (spam predicted as ham) are most important to minimize in real systems.
