INFO284 ML - Group assignment

Candidate numbers: 141, 250, 262

Table of contents

Task 1 - Hotel Reviews Sentiment Analysis

Imports for task 1

In [ ]:
```
# General
import os
import re
import string
import hashlib
import warnings
from typing import Optional, List, cast

# Data handling
import pandas as pd
import numpy as np

# Visualization
import matplotlib.pyplot as plt
```

```
import seaborn as sns

# NLP
import nltk
from nltk.sentiment import SentimentIntensityAnalyzer
from nltk.tokenize import word_tokenize
from nltk.stem import SnowballStemmer
from num2words import num2words
from sklearn.feature_extraction.text import ENGLISH_STOP_WORDS

# Scikit-learn core
from sklearn.pipeline import Pipeline
from sklearn.model_selection import train_test_split, StratifiedKFold, GridSearchCV
from sklearn.metrics import (
    classification_report,
    confusion_matrix,
    accuracy_score,
    precision_score,
    recall_score,
    f1_score,
)

# Scikit-learn estimators
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
```

```
# TensorFlow / Keras for LSTM
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, LSTM, Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping

# Scikit-Keras wrapper
from scikeras.wrappers import KerasClassifier
```

Task 1.a EDA and Data Preparation

The purpose of this project is to develop machine learning models capable of predicting sentiment from hotel review data, which includes positive and negative user-written reviews, numerical ratings, and metadata about hotels and reviewers. The goal is not solely to maximize performance but to demonstrate a thorough understanding of preprocessing, model selection, evaluation, and overfitting control techniques. A key early decision was based on the study by Hutto and Gilbert (2014), where VADER (Valence Aware Dictionary and sEntiment Reasoner) was shown to perform well on short, informal text such as social media posts. Drawing a parallel between social media and hotel reviews both being relatively short and colloquial VADER was selected to label sentiments in the dataset, ensuring consistent and replicable labeling across all models: Logistic Regression, Naive Bayes, Random Forest, and Long Short-Term Memory (LSTM). The project simplifies the sentiment analysis task into binary classification (positive vs. negative). Although introducing a neutral class could have better utilized data, practical constraints around time and resources led to focusing on two classes. Extensive exploratory data analysis (EDA) and preprocessing steps were conducted to ensure models were trained on clean, standardized input. The following sections detail the dataset properties, preprocessing decisions, model choices, evaluation strategies, and analysis of results.

In [ ]:
```
# File Path
cwd            = os.getcwd()
input_file     = os.path.join(cwd, "Hotel_Reviews.csv")
processed_file = os.path.join(cwd, "hotel_reviews_processed.csv")
filtered_file  = os.path.join(cwd, "hotel_reviews_filtered.csv")
```

Cleaning Text

Preprocessing steps included (Müller & Guido, 2016):

- Converting numbers (cardinal and ordinal) to words (e.g., "5" → "five", "1st" → "first")
- Lowercasing, removing punctuation, tokenizing, and stemming words
- Creating a custom stop word list that retained crucial negations ("not", "never", "no")
- Simplifying negation handling by combining negators with subsequent words (e.g., "not good" → "notgood") These steps aimed to simplify vocabulary while preserving important sentiment indicators.

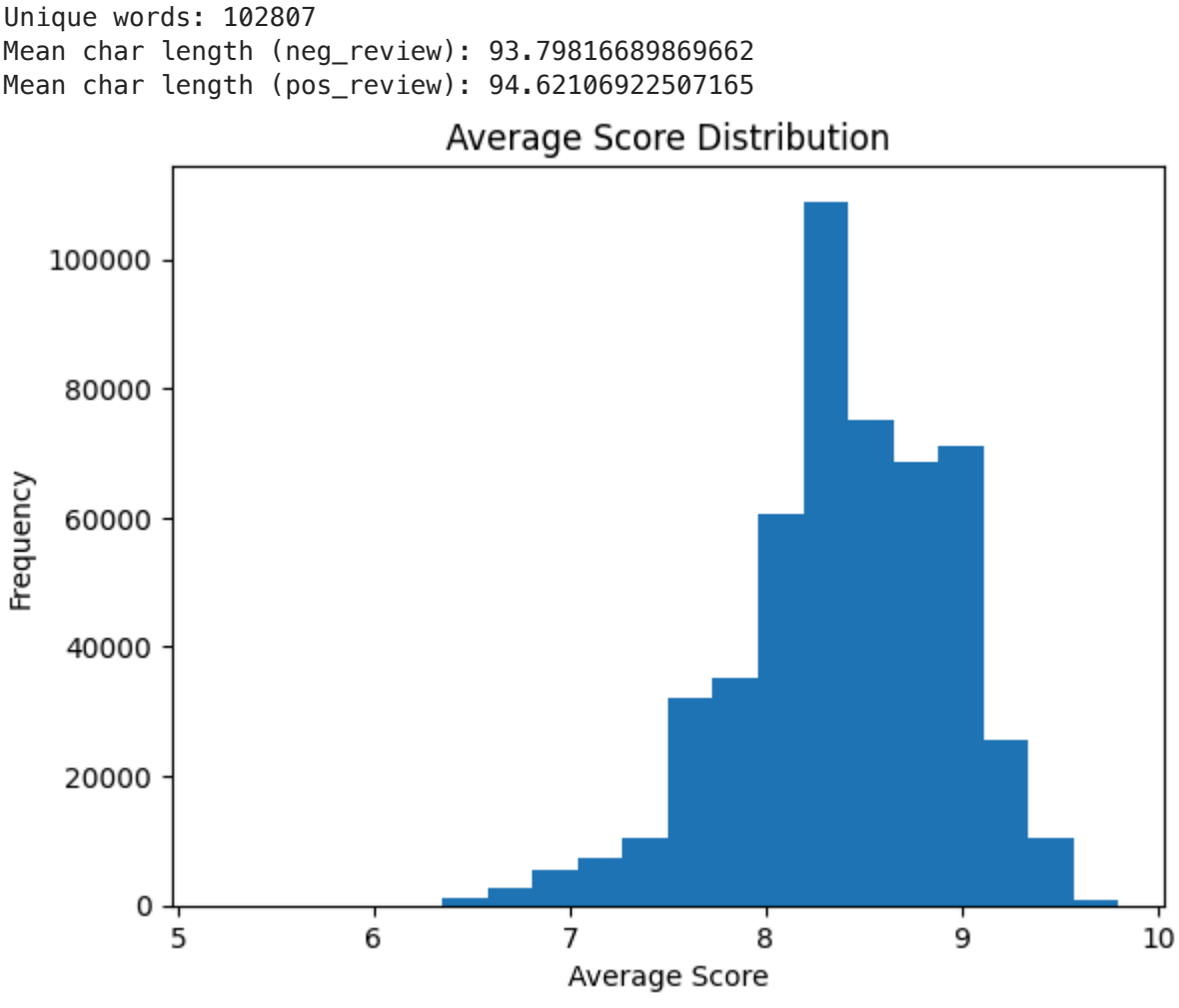
Filtering & cleaning pipeline

The filter_and_clean function processes the raw CSV in chunks, retaining only review and score columns. It deduplicates rows by computing an MD5 hash of each row and skipping repeats. Both review fields are lowercased and numerals converted to words. All English stop-words (except a small negation set) are removed, then any negation token is merged with its follower (e.g. “not great” → “notgreat”) (Hii, n.d.). Each cleaned chunk is written to a temporary file and then atomically renamed to the final output.

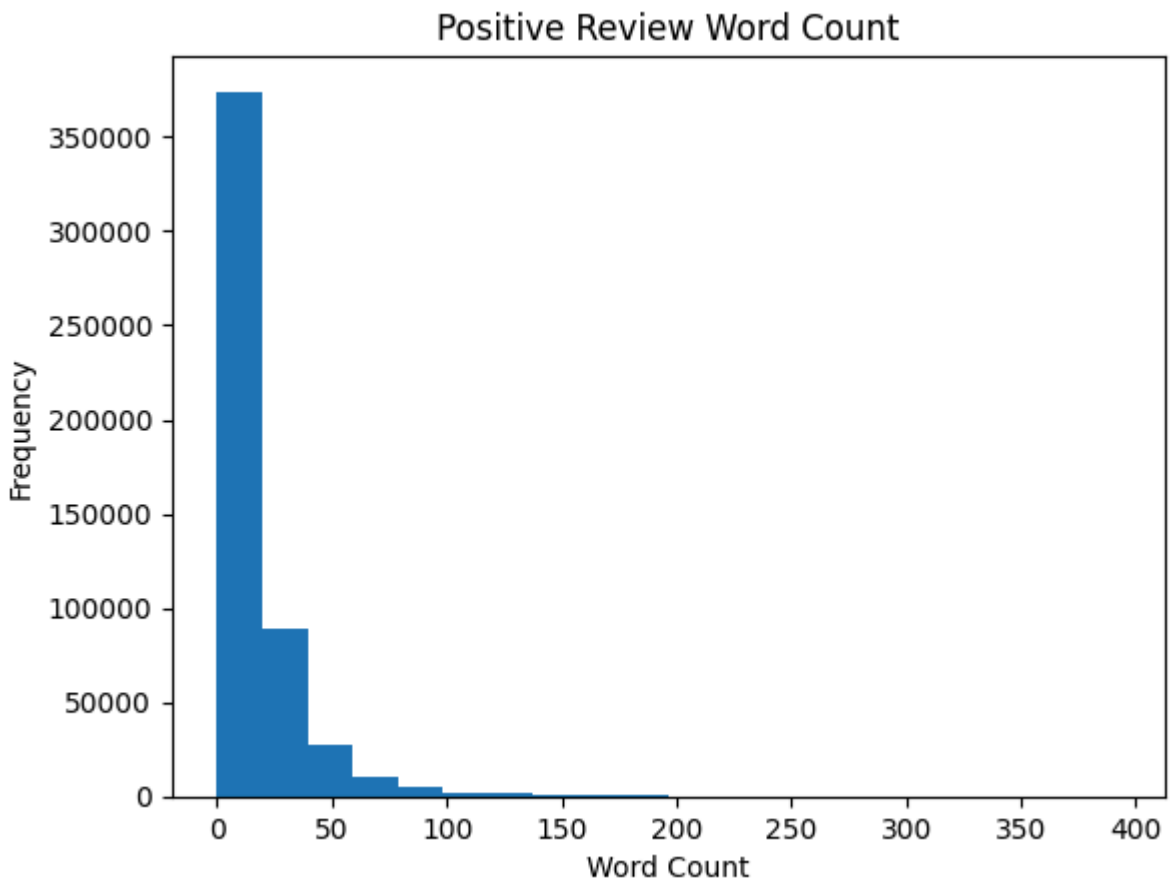
In [ ]:
```python
# Filtering, deduplication, lowercasing, number conversion, stopwords, negation merge
def get_row_hash(row: pd.Series) -> str:
    return hashlib.md5(row.to_json().encode('utf-8')).hexdigest()

def filter_and_clean(file_in: str, file_out: str, chunk_size: int = 10000) -> None:
    negs = {'no','not','never','none','nothing','nowhere','neither','nobody','nor','without'}
    custom_sw = ENGLISH_STOP_WORDS.difference(negs)

    def combine_negations(text: str) -> str:
        toks = text.lower().split()
        out = []
        i = 0
        while i < len(toks):
            t = toks[i]
            if t in negs and i+1 < len(toks):
                out.append(f"{t}_{toks[i+1]}")
                i += 2
            else:
                out.append(t)
                i += 1
        return ' '.join(out)
# Dropping duplicates
    seen = set()
    first = True
    temp = file_out + '.tmp'
    with open(temp, 'w', encoding='utf-8', newline='') as fout:
        for chunk in pd.read_csv(file_in, chunksize=chunk_size):
            # keep only needed cols
            sub = chunk[[
                'negative_review','positive_review',
                'pos_word_count','neg_word_count',
                'avg_score','reviewer_score'
            ]]
            clean_rows = []
            for idx, row in sub.iterrows():
                h = get_row_hash(row)
                if h in seen:
                    continue
                seen.add(h)
                # normalize each review column
                nr = replace_numbers_with_words(row['negative_review'].lower())
                pr = replace_numbers_with_words(row['positive_review'].lower())
                # remove "no negative"
                if nr.strip() == "no negative": nr = ''
                # drop stopwords and merge negations
                nr = combine_negations(' '.join(w for w in nr.split() if w not in custom_sw))
                pr = combine_negations(' '.join(w for w in pr.split() if w not in custom_sw))
                clean_rows.append({
                    'negative_review': nr,
                    'positive_review': pr,
                    'neg_word_count': row['neg_word_count'],
                    'pos_word_count': row['pos_word_count'],
                    'avg_score': row['avg_score'],
                    'reviewer_score': row['reviewer_score']
                })
            clean_df = pd.DataFrame(clean_rows)
            clean_df.to_csv(fout, index=False, header=first)
            first = False
    os.replace(temp, file_out)

# Main
def main():
    chunk_size = 10000
    # 1) Raw processing
    if not os.path.exists(input_file):
        print("Error: Hotel_Reviews.csv not found.")
        return
    HotelReviewProcessor(input_file, processed_file, chunk_size).process_file()

    # === EDA on processed data ===
    df_proc = pd.read_csv(processed_file)
    print("=== Processed Data Overview ===")
    # Missing values per column
    print(df_proc.isnull().sum())
    # Duplicate rows
    print("Duplicate rows:", df_proc.duplicated().sum())
    # Non-empty review counts
    pos_nonempty = df_proc['positive_review'].astype(bool).sum()
    neg_nonempty = df_proc['negative_review'].astype(bool).sum()
    print(f"Non-empty positive reviews: {pos_nonempty}")
    print(f"Non-empty negative reviews: {neg_nonempty}")
    # Vocabulary size (unique tokens)
    all_text = (df_proc['negative_review'] + ' ' + df_proc['positive_review']).str.split().explode()
    print("Unique words:", all_text.nunique())
    # Average character length
    print("Mean char length (neg_review):", df_proc['negative_review'].str.len().mean())
    print("Mean char length (pos_review):", df_proc['positive_review'].str.len().mean())
```

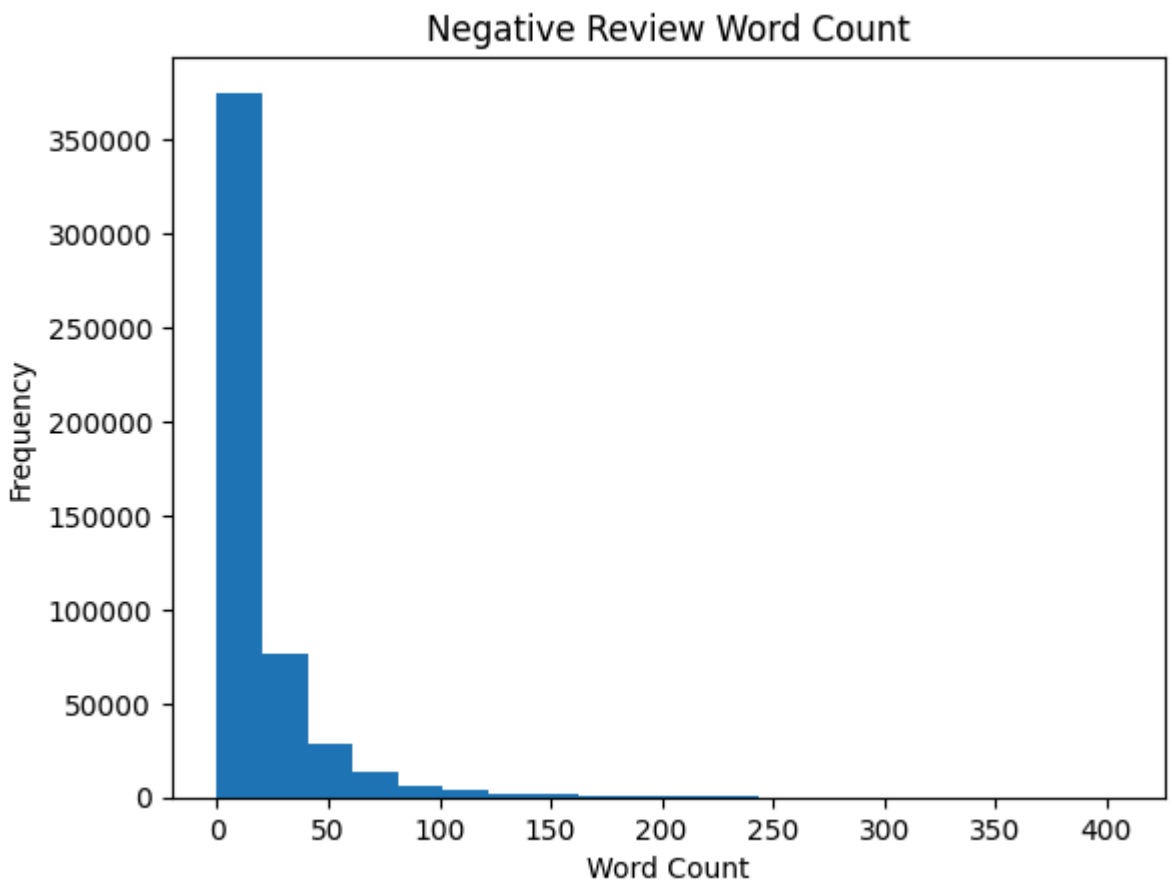
```
    # Histogram of avg_score
    import matplotlib.pyplot as plt
    plt.hist(df_proc['avg_score'], bins=20)
    plt.title('Average Score Distribution')
    plt.xlabel('Average Score')
    plt.ylabel('Frequency')
    plt.show()

    # 2) Filter and processing
    filter_and_clean(processed_file, filtered_file, chunk_size)

    # === EDA on filtered data ===
    df_filt = pd.read_csv(filtered_file)
    print("=== Filtered Data Overview ===")
    print(df_filt.isnull().sum())
    print("Duplicate rows:", df_filt.duplicated().sum())
    pos_nonempty = df_filt['positive_review'].astype(bool).sum()
    neg_nonempty = df_filt['negative_review'].astype(bool).sum()
    print(f"Non-empty positive reviews: {pos_nonempty}")
    print(f"Non-empty negative reviews: {neg_nonempty}")
    # Histograms of word counts
    plt.hist(df_filt['pos_word_count'], bins=20)
    plt.title('Positive Review Word Count')
    plt.xlabel('Word Count')
    plt.ylabel('Frequency')
    plt.show()

    plt.hist(df_filt['neg_word_count'], bins=20)
    plt.title('Negative Review Word Count')
    plt.xlabel('Word Count')
    plt.ylabel('Frequency')
    plt.show()

if __name__ == '__main__':
    main()
```

```
=== Processed Data Overview ===
address                                       0
scoring_extra                                 0
review_date                                   0
avg_score                                     0
hotel_name                                    0
reviewer_nationality                          0
neg_word_count                                0
Total_Number_of_Reviews                       0
pos_word_count                                0
Total_Number_of_Reviews_Reviewer_Has_Given    0
reviewer_score                                0
Tags                                          0
days_since_review                             0
lat                                        3268
lng                                        3268
negative_review                               0
positive_review                               0
dtype: int64
Duplicate rows: 526
Non-empty positive reviews: 515738
Non-empty negative reviews: 515738
Unique words: 102807
Mean char length (neg_review): 93.79816689869662
Mean char length (pos_review): 94.62106922507165
```



```
=== Filtered Data Overview ===
negative_review    126857
positive_review      2328
neg_word_count          0
pos_word_count          0
avg_score               0
reviewer_score          0
dtype: int64
Duplicate rows: 2186
Non-empty positive reviews: 511099
Non-empty negative reviews: 511099
```





EDA result

In Filtered Data Overview we can se that many negative reviews that get filterd out, the reason is when a review has "no negative" it is replaced to NaN to greater capure the meaning that there is no sentiment in that exact review. We also see that word count is very simular bewteen positvie and negative. We might expect a larger posetive class by looking at the avg score graph witch is a dominanly leaning positvily. After cleaning and de-duplication, we have 510317 reviews labeled via VADER as 399956 positive (~78 %) and 111143 negative (~22 %).

Task 1.b Four Macine Learning models

Controlling Overfitting & Underfitting

Overfitting and underfitting were systematically monitored throughout model development. Several strategies were applied to ensure that the models generalized well:

- Stratified Train/Test Split: An 78/22 split preserving label proportions ensured that both positive and negative classes were adequately represented during training and evaluation.
- Regularization and Class Weighting: Logistic Regression used L2 regularization, and class weights were balanced for Logistic Regression and Random Forest models to counteract the strong positive label bias identified during exploratory analysis.
- Cross-Validation (when applicable): Models such as Random Forest and Naive Bayes used 3-fold cross-validation during grid search, improving robustness against random splits.
- Early Stopping (LSTM): For the neural network model, early stopping based on validation loss was applied to prevent overfitting during extended training runs.

1.b.1 - Multinomial Naive Bayes

Naive Bayes was selected as a lightweight and computationally efficient model, particularly suitable when dealing with very large vocabularies, as in this project. Naive Bayes models are known to perform surprisingly well on text classification tasks despite their simplicity, making them a natural second choice. In addition, previous research, including studies comparing sentiment analysis methods (e.g., Hutto & Gilbert, 2014), has used Naive Bayes as a standard baseline for text sentiment tasks. Computational constraints were a practical consideration for including Naive Bayes.

Label Assignment

Since the dataset lacked sentiment labels, VADER's compound score was used for binary labeling:

- Compound ≥ 0.05 → positive
- Compound < 0.05 → negative This approach enabled consistent, objective labeling without manual intervention.

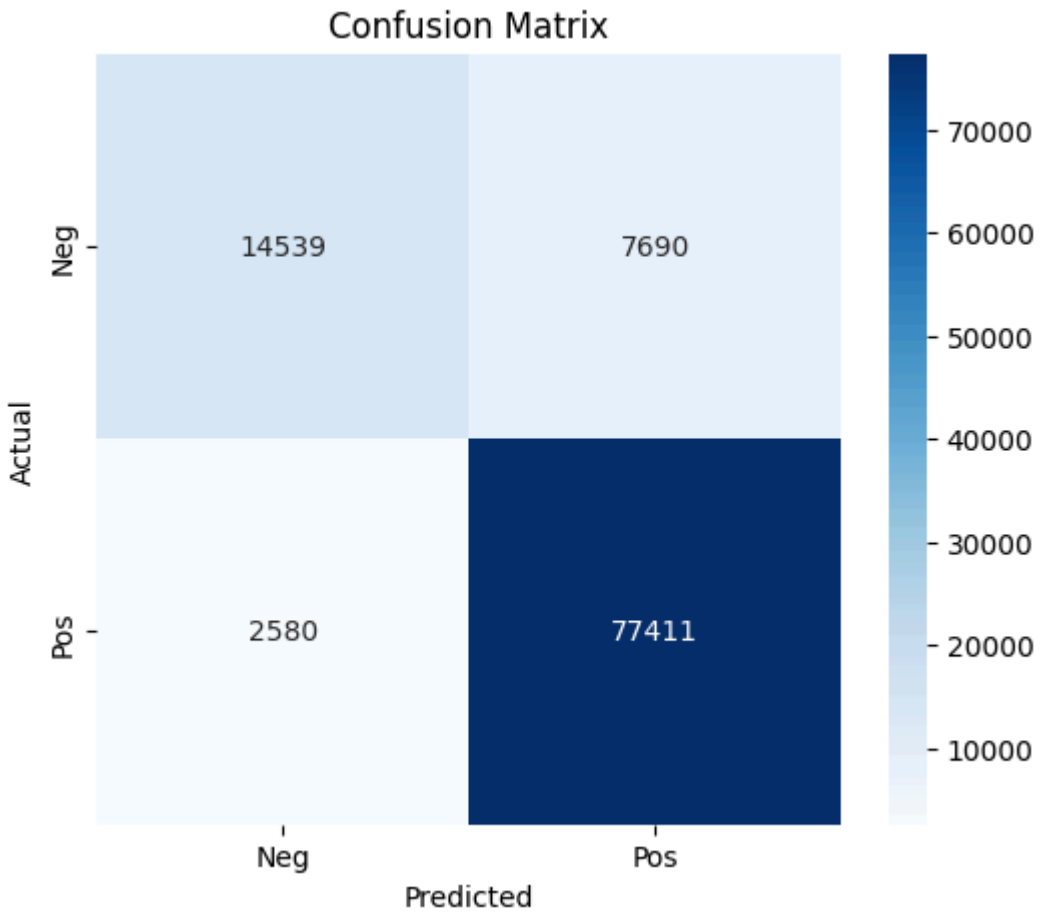
In [ ]:
```python
# Download necessary NLTK resources
nltk.download('vader_lexicon', quiet=True)
nltk.download('punkt', quiet=True)

class SentimentAnalyzerNB:
    def __init__(self, use_grid_search: bool = False) -> None:
        """
        Initialize the Naive Bayes sentiment analyzer.
        :param use_grid_search: Whether to tune hyperparameters with GridSearchCV.
        """
        self.use_grid_search = use_grid_search
        self.sia = SentimentIntensityAnalyzer()
        self.stemmer = SnowballStemmer("english")
        self.pipeline: Optional[Pipeline] = None

    def custom_tokenizer(self, text: str) -> list[str]:
        """
        Tokenize preprocessed text: split on whitespace and stem.
        Assumes text has already been lowercased, punctuation/stop-words removed,
        and negations merged in preprocessing.
        """
        tokens = text.split()
        return [self.stemmer.stem(t) for t in tokens if t]

    def assign_sentiment(self, row: pd.Series, neg_weight: float = 1.0, threshold: float = 0.05) -> int:
        """
        Assign sentiment using VADER scores.
        Combines weighted positive and negative scores, then compares against threshold.
        """
        pos = row.get("positive_review", "") or ""
        neg = row.get("negative_review", "") or ""
        pos_score = self.sia.polarity_scores(pos)["compound"] if pos else 0.0
        neg_score = self.sia.polarity_scores(neg)["compound"] if neg else 0.0
        final = pos_score + neg_weight * neg_score
        return 1 if final >= threshold else 0

    def build_pipeline(self) -> Pipeline:
        """
        Build a TF-IDF + Naive Bayes pipeline with custom tokenization and stemming.
        No stop-word removal here: preprocessing already handled stop-words and negations.
        """
        tfidf = TfidfVectorizer(
            tokenizer=self.custom_tokenizer,
            ngram_range=(1, 3),
            min_df=3,
            lowercase=False,
            preprocessor=None,
            token_pattern=None
        )
        clf = MultinomialNB(alpha=1.0, fit_prior=True, class_prior=[0.5, 0.5])
        self.pipeline = Pipeline([
            ("tfidf", tfidf),
            ("clf", clf),
```

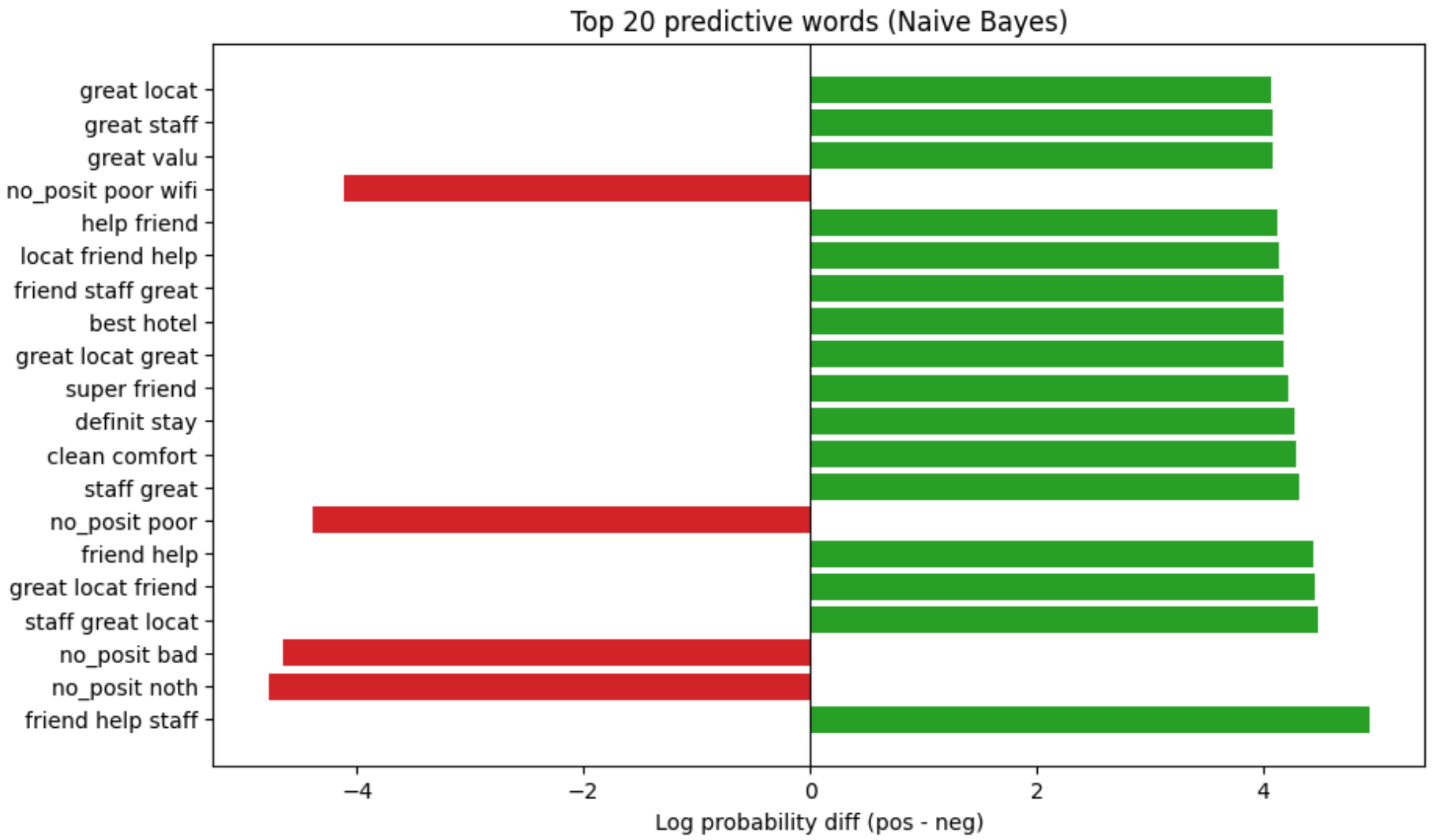
```
        ])
        return self.pipeline

    def run_grid_search(self, X_train: list[str], y_train: list[int]) -> dict:
        """
        Perform GridSearchCV to tune Naive Bayes smoothing and TF-IDF n-gram settings.
        """
        if self.pipeline is None:
            self.build_pipeline()
        param_grid = {
            "tfidf__ngram_range": [(1, 2), (1, 3)],
            "clf__alpha": [0.1, 1.0, 5.0],
        }
        grid = GridSearchCV(
            self.pipeline,
            param_grid,
            cv=3,
            n_jobs=-1,
            verbose=1,
            scoring="f1_macro"
        )
        grid.fit(X_train, y_train)
        self.pipeline = grid.best_estimator_
        return grid.best_params_

    def fit(self, X_train: list[str], y_train: list[int]) -> None:
        """Fit the model pipeline, optionally using grid search."""
        if self.pipeline is None:
            self.build_pipeline()
        if self.use_grid_search:
            best = self.run_grid_search(X_train, y_train)
            print("Best params:", best)
        else:
            self.pipeline.fit(X_train, y_train)

    def predict(self, X_test: list[str]) -> np.ndarray:
        """Predict sentiments for a list of texts."""
        assert self.pipeline is not None
        return self.pipeline.predict(X_test)

    def evaluate(self, X_test: list[str], y_test: list[int]) -> None:
        """Print classification report on the test set."""
        y_pred = self.predict(X_test)
        print("Classification Report:")
        print(classification_report(y_test, y_pred))

    def evaluate_overfitting(self, X_train: list[str], y_train: list[int], X_test: list[str], y_test: list[int]) ->
        """
        Compare performance on train vs test sets to detect overfitting.
        Outputs classification metrics and confusion matrices.
        """
        y_train_pred = self.predict(X_train)
        y_test_pred = self.predict(X_test)

        metrics = {
            "Accuracy": (accuracy_score, {}),
            "Precision (macro)": (precision_score, {"average": "macro"}),
            "Recall (macro)": (recall_score, {"average": "macro"}),
            "F1 (macro)": (f1_score, {"average": "macro"}),
        }

        print("\n=== Overfitting Check ===")
        print(f"{'Metric':<20} {'Train':>7} {'Test':>7} {'Δ':>7}")
        for name, (fn, kwargs) in metrics.items():
            tr = fn(y_train, y_train_pred, **kwargs)
            te = fn(y_test, y_test_pred, **kwargs)
            print(f"{name:<20} {tr:7.3f} {te:7.3f} {tr-te:7.3f}")

        print("\nTrain confusion matrix:")
        print(confusion_matrix(y_train, y_train_pred))
        print("Test confusion matrix:")
        print(confusion_matrix(y_test, y_test_pred))

    def plot_confusion_matrix(self, X_test: list[str], y_test: list[int]) -> None:
        """Plot the confusion matrix as a heatmap."""
        y_pred = self.predict(X_test)
        cm = confusion_matrix(y_test, y_pred)
        plt.figure(figsize=(6, 5))
        sns.heatmap(
            cm, annot=True, fmt="d", cmap="Blues",
            xticklabels=["Neg", "Pos"], yticklabels=["Neg", "Pos"]
        )
        plt.xlabel("Predicted")
        plt.ylabel("Actual")
        plt.title("Confusion Matrix")
        plt.show()
```

```
    def plot_feature_importances(self, top_n: int = 20) -> None:
        """
        Plot the most discriminative words according to Naive Bayes log-probabilities.
        Positive words appear in green; negative words in red.
        """
        assert self.pipeline is not None
        vectorizer = self.pipeline.named_steps['tfidf']
        clf = self.pipeline.named_steps['clf']
        log_prob = clf.feature_log_prob_
        diff = log_prob[1] - log_prob[0]
        feature_names = vectorizer.get_feature_names_out()
        idx = np.argsort(np.abs(diff))[-top_n:]
        top_features = feature_names[idx]
        top_diff = diff[idx]

        plt.figure(figsize=(10, 6))
        colors = ['tab:red' if d < 0 else 'tab:green' for d in top_diff]
        plt.barh(top_features, top_diff, color=colors)
        plt.xlabel("Log probability diff (pos - neg)")
        plt.title(f"Top {top_n} predictive words (Naive Bayes)")
        plt.axvline(0, color='black', linewidth=0.8)
        plt.gca().invert_yaxis()
        plt.show()


def main() -> None:
    warnings.filterwarnings("ignore", category=UserWarning)
```

```
    cwd = os.getcwd()
    data_file = os.path.join(cwd, "hotel_reviews_filtered.csv")
    if not os.path.exists(data_file):
        print("Error: hotel_reviews_filtered.csv not found.")
        return

    df = pd.read_csv(data_file)
    df["positive_review"] = df["positive_review"].fillna("")
    df["negative_review"] = df["negative_review"].fillna("")
    # use pre-cleaned text directly
    df["review_text"] = df["positive_review"] + " " + df["negative_review"]

    analyzer = SentimentAnalyzerNB(use_grid_search=False)
    df["sentiment"] = df.apply(analyzer.assign_sentiment, axis=1)
    print("Raw sentiment distribution:")
    print(df["sentiment"].value_counts())

    X = df["review_text"].tolist()
    y = df["sentiment"].tolist()
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42, stratify=y
    )

    analyzer.build_pipeline()
    analyzer.fit(X_train, y_train)
    analyzer.evaluate(X_test, y_test)
    analyzer.evaluate_overfitting(X_train, y_train, X_test, y_test)
    analyzer.plot_confusion_matrix(X_test, y_test)
    analyzer.plot_feature_importances(top_n=20)

if __name__ == "__main__":
    main()
```

```
Raw sentiment distribution:
sentiment
1    399956
0    111143
Name: count, dtype: int64
Classification Report:
              precision    recall  f1-score   support

           0       0.85      0.65      0.74     22229
           1       0.91      0.97      0.94     79991

    accuracy                           0.90    102220
   macro avg       0.88      0.81      0.84    102220
weighted avg       0.90      0.90      0.89    102220


=== Overfitting Check ===
Metric                 Train    Test       Δ
Accuracy               0.926   0.900   0.026
Precision (macro)      0.904   0.879   0.025
Recall (macro)         0.871   0.811   0.060
F1 (macro)             0.886   0.838   0.048

Train confusion matrix:
[[ 68794  20120]
 [ 10304 309661]]
Test confusion matrix:
[[14539  7690]
 [ 2580 77411]]
```





1.b.2 - Random Forrest Tree

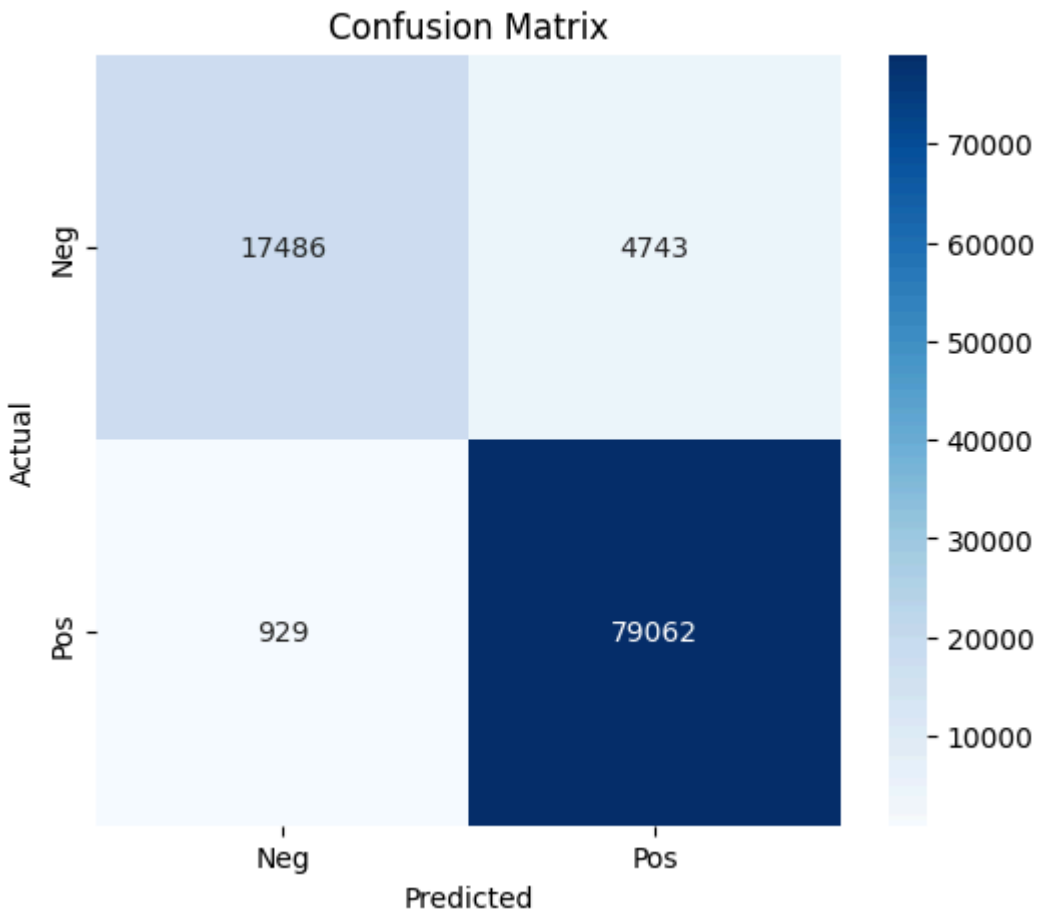
Random Forest is an ensemble of decision trees that excels at high-dimensional data like TF-IDF vectors by averaging many trees to stabilize predictions and reduce overfitting. It also outputs feature importance scores directly, making it easy to interpret which words or n-grams drive sentiment decisions (Müller et. al., 2016).

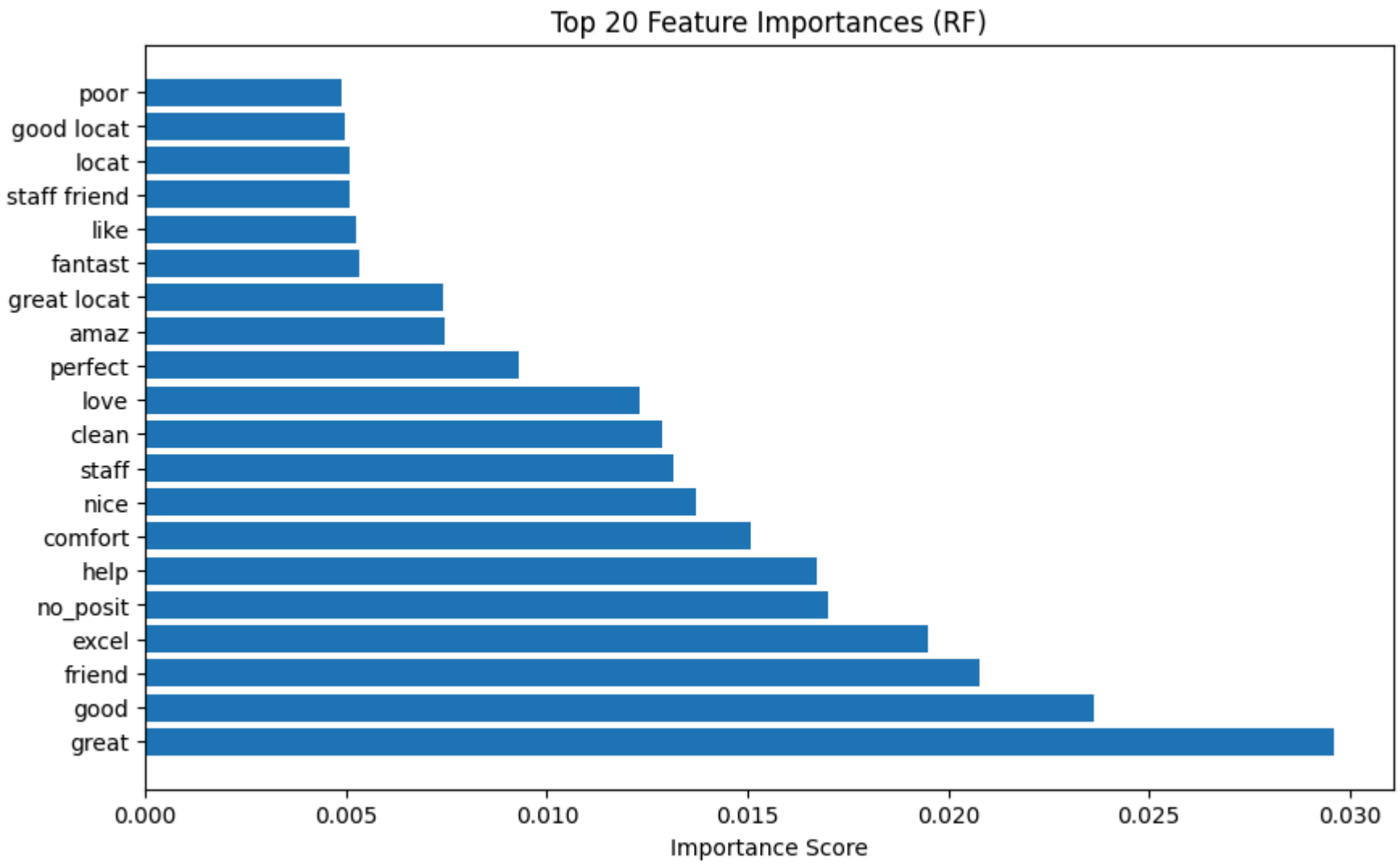
In [54]:
```python
# Download necessary NLTK resources
nltk.download('vader_lexicon', quiet=True)
nltk.download('punkt', quiet=True)

class SentimentAnalyzerRF:
    def __init__(self, use_grid_search: bool = False) -> None:
        """
        Initialize the Random Forest sentiment analyzer.
        :param use_grid_search: If True, enables grid search for hyperparameter tuning.
        """
        self.use_grid_search = use_grid_search
        self.sia = SentimentIntensityAnalyzer()
        self.stemmer = SnowballStemmer("english")
        self.pipeline: Optional[Pipeline] = None

    def custom_tokenizer(self, text: str) -> list[str]:
        """
        Tokenize preprocessed text: split on whitespace and stem.
        Assumes text has already been lowercased, stop-words removed,
        and negations merged in preprocessing.
        """
        tokens = text.split()
        return [self.stemmer.stem(t) for t in tokens if t]

    def assign_sentiment(self, row: pd.Series, neg_weight: float = 1.0, threshold: float = 0.05) -> int:
        """
        Assign binary sentiment label using VADER.
        Combines positive and negative review scores with optional weighting.
        """
        pos = row.get('positive_review', '') or ''
        neg = row.get('negative_review', '') or ''
        pos_score = self.sia.polarity_scores(pos)['compound'] if pos.strip() else 0.0
        neg_score = self.sia.polarity_scores(neg)['compound'] if neg.strip() else 0.0
        final = pos_score + neg_weight * neg_score
        return 1 if final >= threshold else 0

    def build_pipeline(self) -> Pipeline:
        """
        Build a TF-IDF + Random Forest pipeline.
        No internal stop-word removal: preprocessing handles that.
        """
        tfidf = TfidfVectorizer(
            tokenizer=self.custom_tokenizer,
            ngram_range=(1, 3),
            min_df=3,
            lowercase=False,
```

```
            preprocessor=None,
            token_pattern=None
        )
        clf = RandomForestClassifier(
            n_estimators=100,
            class_weight='balanced',
            random_state=42,
            n_jobs=-1
        )
        self.pipeline = Pipeline([
            ('tfidf', tfidf),
            ('clf', clf)
        ])
        return self.pipeline

    def fit(self, X_train: list[str], y_train: list[int]) -> None:
        """
        Train the model on training data.
        Optionally performs hyperparameter search.
        """
        if self.pipeline is None:
            self.build_pipeline()
        if self.use_grid_search:
            self._run_grid_search(X_train, y_train)
        else:
            self.pipeline.fit(X_train, y_train)

    def _run_grid_search(self, X_train: list[str], y_train: list[int]) -> dict:
        """
        Grid search to tune TF-IDF and Random Forest parameters.
        """
        if self.pipeline is None:
            self.build_pipeline()
        param_grid = {
            'tfidf__ngram_range': [(1, 2), (1, 3)],
            'clf__n_estimators': [50, 100],
            'clf__max_depth': [None, 10]
        }
        grid = GridSearchCV(
            self.pipeline,
            param_grid,
            cv=3,
            n_jobs=-1,
            verbose=1,
            scoring='f1_macro'
        )
        grid.fit(X_train, y_train)
        self.pipeline = grid.best_estimator_
        return grid.best_params_

    def predict(self, X: list[str]) -> np.ndarray:
        """Generate predictions for input samples."""
        assert self.pipeline is not None
        return self.pipeline.predict(X)

    def evaluate(self, X_test: list[str], y_test: list[int]) -> None:
        """Evaluate and print classification report on test data."""
        y_pred = self.predict(X_test)
        print("Classification Report:")
        print(classification_report(y_test, y_pred))

    def evaluate_overfitting(
        self,
        X_train: list[str], y_train: list[int],
        X_test: list[str], y_test: list[int]
    ) -> None:
        """
        Compare performance on training vs testing sets.
        """
        y_tr = self.predict(X_train)
        y_te = self.predict(X_test)
        metrics = {
            'Accuracy': (accuracy_score, {}),
            'Precision': (precision_score, {'average': 'macro'}),
            'Recall':    (recall_score, {'average': 'macro'}),
            'F1':        (f1_score, {'average': 'macro'})
        }
        print("\n=== Overfitting Check ===")
        print(f"{'Metric':<12}{'Train':>8}{'Test':>8}{'Δ':>8}")
        for name, (fn, kw) in metrics.items():
            tr = fn(y_train, y_tr, **kw)
            te = fn(y_test,  y_te, **kw)
            print(f"{name:<12}{tr:8.3f}{te:8.3f}{(tr-te):8.3f}")
        print("\nTrain Confusion Matrix:")
        print(confusion_matrix(y_train, y_tr))
        print("\nTest Confusion Matrix:")
        print(confusion_matrix(y_test, y_te))
```

```
    def plot_confusion_matrix(self, X_test: list[str], y_test: list[int]) -> None:
        """Plot confusion matrix heatmap."""
        y_pred = self.predict(X_test)
        cm = confusion_matrix(y_test, y_pred)
        plt.figure(figsize=(6, 5))
        sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                    xticklabels=['Neg','Pos'], yticklabels=['Neg','Pos'])
        plt.xlabel('Predicted')
        plt.ylabel('Actual')
        plt.title('Confusion Matrix')
        plt.show()

    def plot_feature_importances(self, top_n: int = 20) -> None:
        """
        Plot top_n important TF-IDF features by Random Forest.
        """
        assert self.pipeline is not None
        vectorizer = self.pipeline.named_steps['tfidf']
        clf = self.pipeline.named_steps['clf']
        importances = clf.feature_importances_
        feat_names = vectorizer.get_feature_names_out()
        idx = np.argsort(importances)[-top_n:]
        plt.figure(figsize=(10,6))
        plt.barh(feat_names[idx], importances[idx])
        plt.xlabel('Importance Score')
        plt.title(f'Top {top_n} Feature Importances (RF)')
        plt.gca().invert_yaxis()
        plt.show()


def main() -> None:
    warnings.filterwarnings('ignore', category=UserWarning)
    cwd = os.getcwd()
    data_file = os.path.join(cwd, 'hotel_reviews_filtered.csv')
    if not os.path.exists(data_file):
        print("Error: hotel_reviews_filtered.csv not found.")
        return

    df = pd.read_csv(data_file)
    df['positive_review'] = df['positive_review'].fillna('').astype(str)
    df['negative_review'] = df['negative_review'].fillna('').astype(str)
    df['review_text'] = df['positive_review'] + ' ' + df['negative_review']

    analyzer = SentimentAnalyzerRF(use_grid_search=False)
    df['sentiment'] = df.apply(analyzer.assign_sentiment, axis=1)
    print("Sentiment distribution:")
    print(df['sentiment'].value_counts())

    X = df['review_text'].tolist()
    y = df['sentiment'].tolist()
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42, stratify=y
    )

    analyzer.build_pipeline()
    analyzer.fit(X_train, y_train)
    analyzer.evaluate(X_test, y_test)
    analyzer.evaluate_overfitting(X_train, y_train, X_test, y_test)
    analyzer.plot_confusion_matrix(X_test, y_test)
    analyzer.plot_feature_importances()

if __name__ == '__main__':
    main()
```

```
Sentiment distribution:
sentiment
1    399956
0    111143
Name: count, dtype: int64
Classification Report:
              precision    recall  f1-score   support

           0       0.95      0.79      0.86     22229
           1       0.94      0.99      0.97     79991

    accuracy                           0.94    102220
   macro avg       0.95      0.89      0.91    102220
weighted avg       0.94      0.94      0.94    102220


=== Overfitting Check ===
Metric         Train    Test       Δ
Accuracy       1.000   0.945   0.055
Precision      1.000   0.946   0.053
Recall         1.000   0.888   0.112
F1             1.000   0.913   0.087

Train Confusion Matrix:
[[ 88853     61]
 [    33 319932]]

Test Confusion Matrix:
[[17486  4743]
 [  929 79062]]
```





1.b.3 - Logical Regression

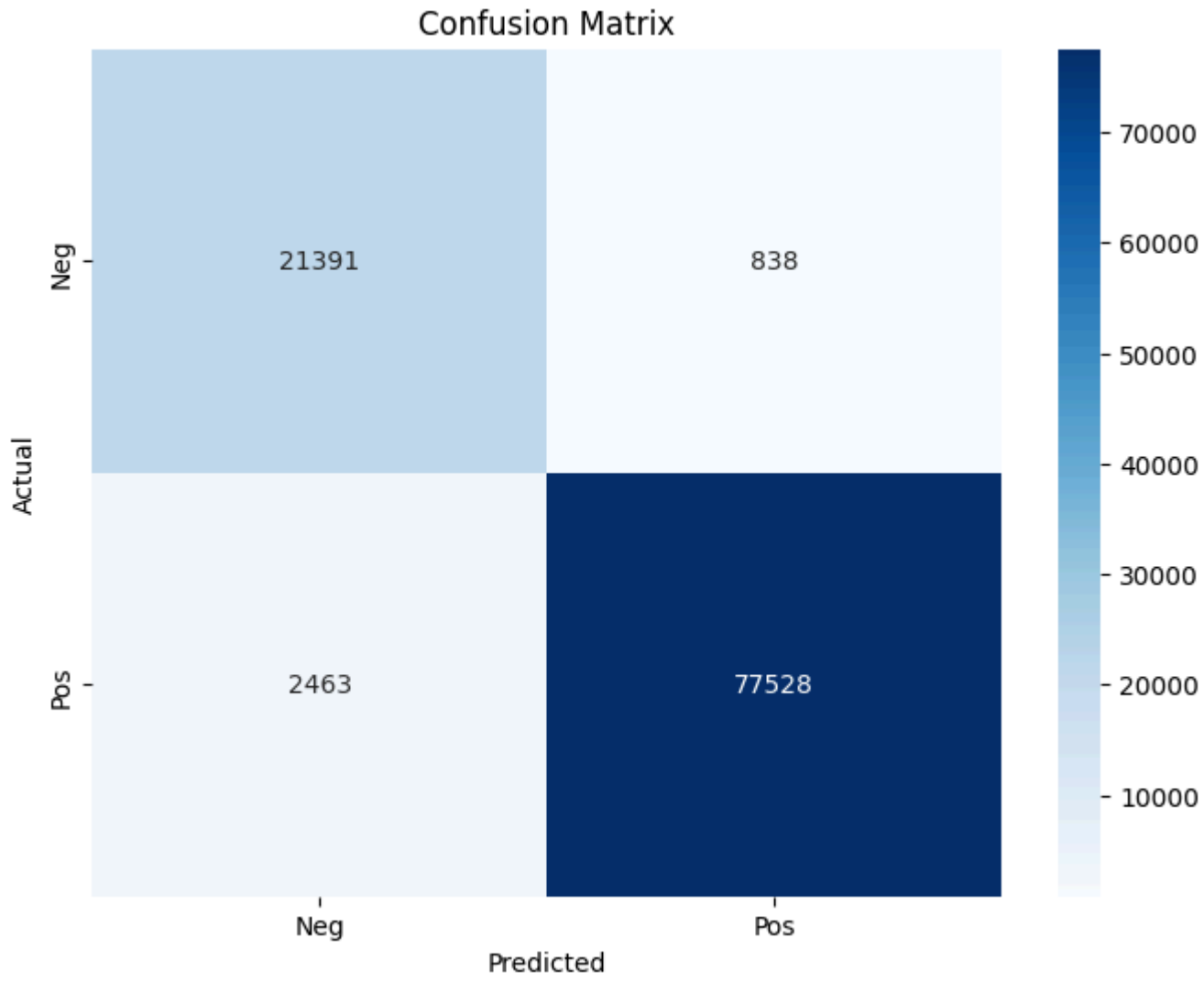
Logistic Regression was chosen as the first model because it is a well-known baseline for text classification tasks. The main textbook used during the course, Introduction to Machine Learning with Python (Müller et. al., 2016), demonstrated strong results with Logistic Regression on movie review sentiment data, a problem structurally similar to the hotel reviews considered here. Logistic Regression offers strong interpretability, handles large feature spaces (like TF-IDF vectors) well, and provides fast training even on large datasets.

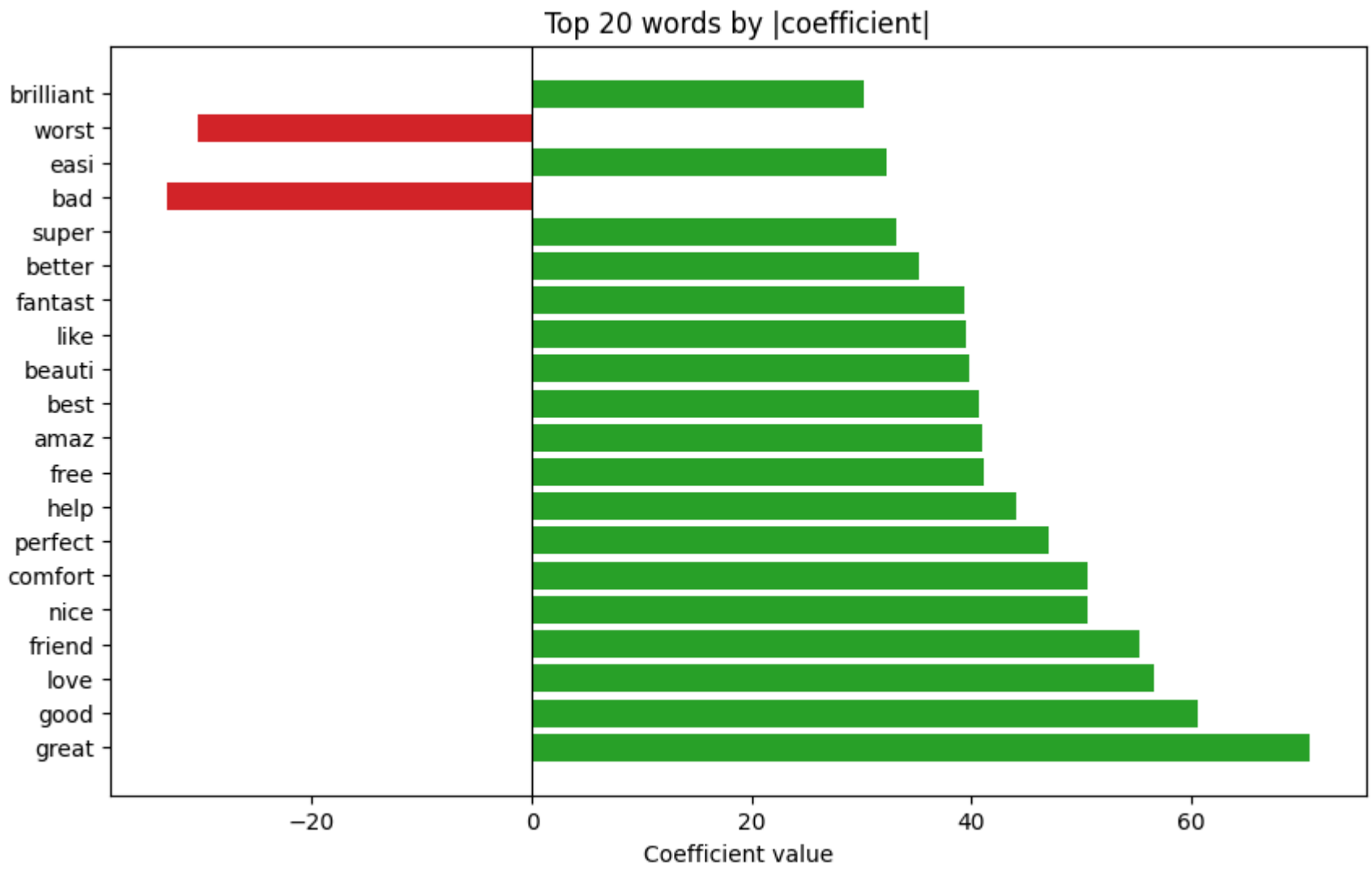
In [51]:
```python
class SentimentAnalyzerLR:
    def __init__(self, use_grid_search: bool = False) -> None:
        """
        Initialize the Logistic Regression sentiment analyzer.
        :param use_grid_search: Whether to perform hyperparameter tuning via grid search.
        """
        self.use_grid_search = use_grid_search
        self.sia = SentimentIntensityAnalyzer()
        self.stemmer = SnowballStemmer("english")
        self.pipeline: Optional[Pipeline] = None

    def custom_tokenizer(self, text: str) -> list[str]:
        """
        Tokenize preprocessed text: split on whitespace and stem.
        Assumes preprocessing handled lowercasing, stop-word removal, and negation merging.
        """
        tokens = text.split()
        return [self.stemmer.stem(t) for t in tokens if t]

    def assign_sentiment(
        self,
        row: pd.Series,
        neg_weight: float = 1.0,
        threshold: float = 0.05
    ) -> int:
        """
        Assign a binary sentiment label based on VADER scores.
        Combines positive and weighted negative compound scores against threshold.
        """
        pos = row.get('positive_review', '') or ''
        neg = row.get('negative_review', '') or ''
        pos_score = self.sia.polarity_scores(pos)['compound'] if pos.strip() else 0.0
        neg_score = self.sia.polarity_scores(neg)['compound'] if neg.strip() else 0.0
        final_score = pos_score + neg_weight * neg_score
        return 1 if final_score >= threshold else 0

    def build_pipeline(self) -> Pipeline:
        """
        Build the classification pipeline:
        - TF-IDF vectorizer with custom tokenizer
        - Logistic Regression classifier
        """
        tfidf = TfidfVectorizer(
            tokenizer=self.custom_tokenizer,
```

```
            ngram_range=(1, 3),
            min_df=3,
            lowercase=False,
            preprocessor=None,
            token_pattern=None
        )
        clf = LogisticRegression(
            max_iter=1000,
            solver='saga',
            C=10.0,
            class_weight='balanced',
            random_state=42
        )
        self.pipeline = Pipeline([
            ('tfidf', tfidf),
            ('clf', clf)
        ])
        return self.pipeline

    def run_grid_search(self, X_train: list[str], y_train: list[int]) -> dict:
        """
        Perform a grid search for hyperparameter tuning.
        """
        if self.pipeline is None:
            self.build_pipeline()
        param_grid = {
            'tfidf__ngram_range': [(1,2), (1,3)],
            'clf__C': [0.1, 1.0, 10.0],
            'clf__max_iter': [1000, 2000]
        }
        grid = GridSearchCV(
            self.pipeline,
            param_grid,
            cv=3,
            scoring='f1_macro',
            n_jobs=-1,
            verbose=1
        )
        grid.fit(X_train, y_train)
        self.pipeline = grid.best_estimator_
        print("Best parameters found:", grid.best_params_)
        return grid.best_params_

    def fit(self, X_train: list[str], y_train: list[int]) -> None:
        """
        Train the model pipeline.
        """
        if self.pipeline is None:
            self.build_pipeline()
        if self.use_grid_search:
            self.run_grid_search(X_train, y_train)
        else:
            self.pipeline.fit(X_train, y_train)

    def predict(self, X_test: list[str]) -> np.ndarray:
        """
        Make predictions using the trained pipeline.
        """
        assert self.pipeline is not None
        return self.pipeline.predict(X_test)

    def evaluate(self, X_test: list[str], y_test: list[int]) -> None:
        """
        Print classification report for the model on test data.
        """
        y_pred = self.predict(X_test)
        print("Classification Report:")
        print(classification_report(y_test, y_pred))

    def evaluate_overfitting(
        self,
        X_train: list[str], y_train: list[int],
        X_test: list[str], y_test: list[int]
    ) -> None:
        """
        Evaluate overfitting by comparing train and test scores.
        """
        y_train_pred = self.predict(X_train)
        y_test_pred  = self.predict(X_test)
        metrics = {
            "Accuracy": (accuracy_score, {}),
            "Precision": (precision_score, {"average": "macro"}),
            "Recall":    (recall_score,    {"average": "macro"}),
            "F1":        (f1_score,        {"average": "macro"}),
        }
        print("\n=== Overfitting Check ===")
        print(f"{'Metric':<12}{'Train':>8}{'Test':>8}{'Δ':>8}")
```

```
        for name, (fn, kw) in metrics.items():
            tr = fn(y_train, y_train_pred, **kw)
            te = fn(y_test,  y_test_pred,  **kw)
            print(f"{name:<12}{tr:8.3f}{te:8.3f}{(tr-te):8.3f}")
        print("\nTrain confusion matrix:")
        print(confusion_matrix(y_train, y_train_pred))
        print("Test confusion matrix:")
        print(confusion_matrix(y_test, y_test_pred))

    def plot_confusion_matrix(self, X_test: list[str], y_test: list[int]) -> None:
        """
        Plot a confusion matrix heatmap.
        """
        y_pred = self.predict(X_test)
        cm = confusion_matrix(y_test, y_pred)
        plt.figure(figsize=(8, 6))
        sns.heatmap(
            cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=['Neg', 'Pos'], yticklabels=['Neg', 'Pos']
        )
        plt.xlabel("Predicted")
        plt.ylabel("Actual")
        plt.title("Confusion Matrix")
        plt.show()

    def plot_feature_importances(self, top_n: int = 20) -> None:
        """
        Plot the most important features based on Logistic Regression coefficients.
        """
        assert self.pipeline is not None
        vectorizer = self.pipeline.named_steps['tfidf']
        clf = self.pipeline.named_steps['clf']
        feature_names = vectorizer.get_feature_names_out()
        coefs = clf.coef_[0]
        idx = np.argsort(np.abs(coefs))[-top_n:]
        top_features = feature_names[idx]
        top_coefs = coefs[idx]
        plt.figure(figsize=(10, 6))
        colors = ['tab:red' if c < 0 else 'tab:green' for c in top_coefs]
        plt.barh(top_features, top_coefs, color=colors)
        plt.xlabel("Coefficient value")
        plt.title(f"Top {top_n} words by |coefficient|")
        plt.axvline(0, color='black', linewidth=0.8)
        plt.gca().invert_yaxis()
        plt.show()


def main() -> None:
    warnings.filterwarnings('ignore', category=UserWarning)
    cwd = os.getcwd()
    data_file = os.path.join(cwd, 'hotel_reviews_filtered.csv')
    if not os.path.exists(data_file):
        print("Error: hotel_reviews_filtered.csv not found.")
        return

    df = pd.read_csv(data_file)
    df['positive_review'] = df['positive_review'].fillna('').astype(str)
    df['negative_review'] = df['negative_review'].fillna('').astype(str)
    df['review_text'] = df['positive_review'] + ' ' + df['negative_review']

    analyzer = SentimentAnalyzerLR(use_grid_search=False)
    df['sentiment'] = df.apply(analyzer.assign_sentiment, axis=1)
    print("Sentiment distribution:")
    print(df['sentiment'].value_counts())

    X = df['review_text'].tolist()
    y = df['sentiment'].tolist()
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42, stratify=y
    )

    analyzer.build_pipeline()
    analyzer.fit(X_train, y_train)
    analyzer.evaluate(X_test, y_test)
    analyzer.evaluate_overfitting(X_train, y_train, X_test, y_test)
    analyzer.plot_confusion_matrix(X_test, y_test)
    analyzer.plot_feature_importances(top_n=20)

if __name__ == '__main__':
    main()
```

```
Sentiment distribution:
sentiment
1    399956
0    111143
Name: count, dtype: int64
Classification Report:
              precision    recall  f1-score   support

           0       0.90      0.96      0.93     22229
           1       0.99      0.97      0.98     79991

    accuracy                           0.97    102220
   macro avg       0.94      0.97      0.95    102220
weighted avg       0.97      0.97      0.97    102220


=== Overfitting Check ===
Metric         Train    Test       Δ
Accuracy       0.995   0.968   0.027
Precision      0.989   0.943   0.046
Recall         0.996   0.966   0.030
F1             0.992   0.954   0.039

Train confusion matrix:
[[ 88776    138]
 [  1979 317986]]
Test confusion matrix:
[[21391   838]
 [ 2463 77528]]
```





1.b.4 - LSTM model Keras

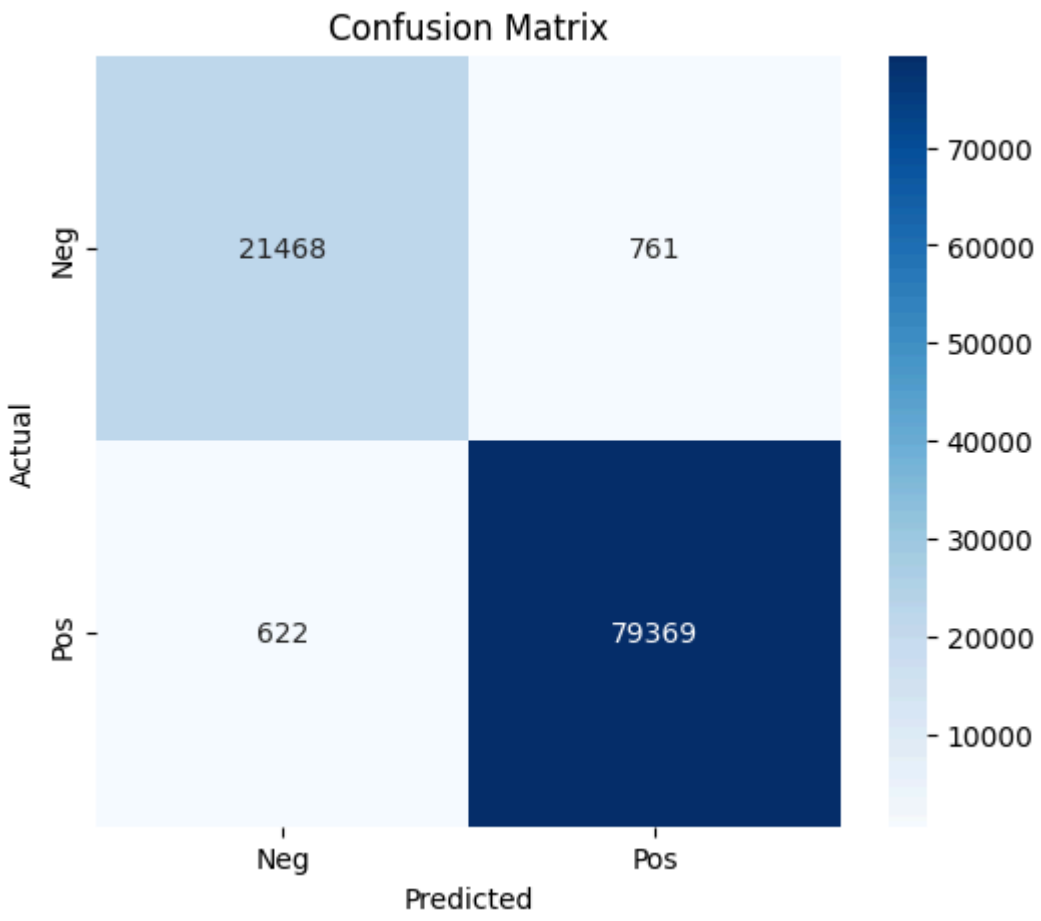
An LSTM model was included to explore the potential benefits of learning sequential relationships within the text data, which traditional machine learning models such as Logistic Regression and Random Forest cannot capture. The model design was based on standard best practices for text classification tasks, following patterns described in TensorFlow's official RNN guide (TensorFlow, 2023).

In [53]:
```python
# Download necessary NLTK resources
nltk.download('vader_lexicon', quiet=True)
nltk.download('punkt', quiet=True)

class SentimentAnalyzerLSTM:
    def __init__(self, use_grid_search: bool = False):
        """
        Initialize the LSTM sentiment analyzer.
        Assumes text has been preprocessed: lowercased, punctuation- and stopwords-removed, negations merged.
        """
        self.use_grid_search = use_grid_search
        self.sia = SentimentIntensityAnalyzer()
        self.tokenizer: Optional[Tokenizer] = None
        self.model: Optional[Sequential] = None

        # Sequence parameters
        self.max_vocab_size = 20000
        self.max_sequence_length = 100

        # Model parameters
        self.embedding_dim = 100
        self.lstm_units = 128
        self.dropout_rate = 0.5

        # Training parameters
        self.epochs = 10
        self.batch_size = 128

    def assign_sentiment(self, row: pd.Series, neg_weight: float = 1.0, threshold: float = 0.05) -> int:
        """
        Assign binary sentiment label using VADER on preprocessed review texts.
        """
        pos = row.get('positive_review', '') or ''
        neg = row.get('negative_review', '') or ''
        pos_score = self.sia.polarity_scores(pos)['compound'] if pos else 0.0
        neg_score = self.sia.polarity_scores(neg)['compound'] if neg else 0.0
        final = pos_score + neg_weight * neg_score
        return 1 if final >= threshold else 0

    def build_tokenizer(self, texts: List[str]) -> None:
        """
        Fit a Keras Tokenizer on the cleaned texts.
        """
        self.tokenizer = Tokenizer(num_words=self.max_vocab_size, oov_token='<OOV>')
        self.tokenizer.fit_on_texts(texts)
```

```
    def preprocess_texts(self, texts: List[str]) -> np.ndarray:
        """
        Convert texts to padded sequences using the fitted tokenizer.
        """
        if self.tokenizer is None:
            self.build_tokenizer(texts)
        seqs = self.tokenizer.texts_to_sequences(texts)
        return pad_sequences(seqs, maxlen=self.max_sequence_length, padding='post', truncating='post')

    def build_model(self, lstm_units: Optional[int] = None, dropout_rate: Optional[float] = None) -> Sequential:
        """
        Build and compile the LSTM model.
        """
        units = lstm_units or self.lstm_units
        dout = dropout_rate or self.dropout_rate
        model = Sequential([
            Embedding(input_dim=self.max_vocab_size, output_dim=self.embedding_dim, input_length=self.max_sequence_
            LSTM(units, dropout=dout, recurrent_dropout=dout),
            Dense(64, activation='relu'),
            Dropout(dout),
            Dense(1, activation='sigmoid')
        ])
        model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
        self.model = model
        return model

    def fit(self, X_train: List[str], y_train: List[int]) -> None:
        """
        Train the LSTM with early stopping, on preprocessed sequences.
        """
        X_pad = self.preprocess_texts(X_train)
        if self.model is None:
            self.build_model()
        from keras.callbacks import EarlyStopping
        early = EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)
        self.model.fit(X_pad, np.array(y_train), epochs=self.epochs,
                       batch_size=self.batch_size, validation_split=0.2,
                       callbacks=[early], verbose=1)

    def predict(self, X_test: List[str]) -> np.ndarray:
        """
        Predict binary sentiment labels for new texts.
        """
        X_pad = self.preprocess_texts(X_test)
        preds = self.model.predict(X_pad, verbose=0)
        return (preds >= 0.5).astype(int).flatten()

    def evaluate(self, X_test: List[str], y_test: List[int]) -> None:
        """
        Print classification report on the test set.
        """
        y_pred = self.predict(X_test)
        print(classification_report(y_test, y_pred))

    def evaluate_overfitting(self, X_train: List[str], y_train: List[int],
                             X_test: List[str], y_test: List[int]) -> None:
        """
        Compare train vs test performance to detect overfitting.
        """
        y_tr = self.predict(X_train)
        y_te = self.predict(X_test)
        metrics = {'Accuracy': (accuracy_score, {}),
                   'Precision': (precision_score, {'average':'macro'}),
                   'Recall': (recall_score, {'average':'macro'}),
                   'F1': (f1_score, {'average':'macro'})}
        print("\n=== Overfitting Check ===")
        for name, (fn, kw) in metrics.items():
            tr = fn(y_train, y_tr, **kw)
            te = fn(y_test, y_te, **kw)
            print(f"{name}: train={tr:.3f}, test={te:.3f}, Δ={tr-te:.3f}")
        print("Train CM:\n", confusion_matrix(y_train, y_tr))
        print("Test CM:\n", confusion_matrix(y_test, y_te))

    def plot_confusion_matrix(self, X_test: List[str], y_test: List[int]) -> None:
        """
        Plot a heatmap of the confusion matrix.
        """
        y_pred = self.predict(X_test)
        cm = confusion_matrix(y_test, y_pred)
        plt.figure(figsize=(6,5))
        sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                    xticklabels=['Neg','Pos'], yticklabels=['Neg','Pos'])
        plt.xlabel('Predicted'); plt.ylabel('Actual'); plt.title('Confusion Matrix')
        plt.show()

    def run_grid_search(self, X_train: List[str], y_train: List[int]) -> dict:
```

```
        """
        Grid search over LSTM units, dropout, batch size, and epochs.
        """
        X_pad = self.preprocess_texts(X_train)
        def builder(units, dout): return self.build_model(units, dout)
        keras_clf = KerasClassifier(model=builder, verbose=0)
        param_grid = {'model__lstm_units': [64,128], 'model__dropout_rate': [0.3,0.5],
                      'batch_size': [128,256], 'epochs': [10,20]}
        cv = StratifiedKFold(n_splits=3, shuffle=True, random_state=42)
        grid = GridSearchCV(keras_clf, param_grid, cv=cv, scoring='f1_macro')
        grid.fit(X_pad, y_train)
        print('Best params:', grid.best_params_)
        self.model = grid.best_estimator_.model_
        return grid.best_params_


def main():
    warnings.filterwarnings('ignore', category=UserWarning)
    cwd = os.getcwd()
    data_file = os.path.join(cwd, 'hotel_reviews_filtered.csv')
    if not os.path.exists(data_file):
        print("Error: filtered CSV not found.")
        return
    df = pd.read_csv(data_file)
    df['positive_review'] = df['positive_review'].fillna('').astype(str)
    df['negative_review'] = df['negative_review'].fillna('').astype(str)
    df['review_text'] = df['positive_review'] + ' ' + df['negative_review']

    analyzer = SentimentAnalyzerLSTM(use_grid_search=False)
    df['sentiment'] = df.apply(analyzer.assign_sentiment, axis=1)
    print('Sentiment distribution:', df['sentiment'].value_counts().to_dict())

    X, y = df['review_text'].tolist(), df['sentiment'].tolist()
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42, stratify=y
    )

    analyzer.fit(X_train, y_train)
    analyzer.evaluate(X_test, y_test)
    analyzer.evaluate_overfitting(X_train, y_train, X_test, y_test)
    analyzer.plot_confusion_matrix(X_test, y_test)

if __name__ == '__main__':
    main()
```

```
Sentiment distribution: {1: 399956, 0: 111143}
Epoch 1/10
2556/2556 ━━━━━━━━━━━━━━━━━━━━ 719s 280ms/step - accuracy: 0.7819 - loss: 0.5326 - val_accuracy: 0.7793 - val_loss:
0.5279
Epoch 2/10
2556/2556 ━━━━━━━━━━━━━━━━━━━━ 716s 280ms/step - accuracy: 0.7826 - loss: 0.5257 - val_accuracy: 0.7793 - val_loss:
0.5287
Epoch 3/10
2556/2556 ━━━━━━━━━━━━━━━━━━━━ 708s 277ms/step - accuracy: 0.7838 - loss: 0.5230 - val_accuracy: 0.7792 - val_loss:
0.5279
Epoch 4/10
2556/2556 ━━━━━━━━━━━━━━━━━━━━ 705s 276ms/step - accuracy: 0.7835 - loss: 0.5221 - val_accuracy: 0.7790 - val_loss:
0.5282
Epoch 5/10
2556/2556 ━━━━━━━━━━━━━━━━━━━━ 710s 278ms/step - accuracy: 0.8799 - loss: 0.2964 - val_accuracy: 0.9820 - val_loss:
0.0509
Epoch 6/10
2556/2556 ━━━━━━━━━━━━━━━━━━━━ 752s 294ms/step - accuracy: 0.9848 - loss: 0.0452 - val_accuracy: 0.9867 - val_loss:
0.0379
Epoch 7/10
2556/2556 ━━━━━━━━━━━━━━━━━━━━ 724s 283ms/step - accuracy: 0.9906 - loss: 0.0281 - val_accuracy: 0.9865 - val_loss:
0.0370
Epoch 8/10
2556/2556 ━━━━━━━━━━━━━━━━━━━━ 714s 279ms/step - accuracy: 0.9926 - loss: 0.0217 - val_accuracy: 0.9870 - val_loss:
0.0384
Epoch 9/10
2556/2556 ━━━━━━━━━━━━━━━━━━━━ 720s 282ms/step - accuracy: 0.9939 - loss: 0.0186 - val_accuracy: 0.9864 - val_loss:
0.0415
Epoch 10/10
2556/2556 ━━━━━━━━━━━━━━━━━━━━ 756s 296ms/step - accuracy: 0.9952 - loss: 0.0147 - val_accuracy: 0.9865 - val_loss:
0.0422
              precision    recall  f1-score   support

           0       0.97      0.97      0.97     22229
           1       0.99      0.99      0.99     79991

    accuracy                           0.99    102220
   macro avg       0.98      0.98      0.98    102220
weighted avg       0.99      0.99      0.99    102220


=== Overfitting Check ===
Accuracy: train=0.993, test=0.986, Δ=0.007
Precision: train=0.991, test=0.981, Δ=0.010
Recall: train=0.990, test=0.979, Δ=0.011
F1: train=0.990, test=0.980, Δ=0.010
Train CM:
 [[ 87429   1485]
 [  1223 318742]]
Test CM:
 [[21468   761]
 [  622 79369]]
```



Evaluation Metrics and Model Comparison

Overall, model evaluations based on accuracy, precision, recall, and F1-score showed small gaps between training and testing results, usually below 3-10% for traditional models and below 1% for the LSTM, indicating good control of overfitting. However, confusion matrix

analyses consistently revealed an asymmetry between classes:

- All models performed slightly worse on negative sentiment predictions compared to positive ones.
- The F1-score for negative sentiment was typically 2–20% lower than for positive sentiment.

This performance gap can be attributed to several factors:

1. Label imbalance: The dataset had approximately 78% positive samples after VADER-based labeling.
2. Content imbalance: Positive reviews were typically longer and more expressive, whereas negative reviews tended to be shorter and sometimes ambiguous, providing less signal for classification.
3. Preprocessing impact: A custom preprocessing step merged negation words with their following terms (e.g., "not good" → "notgood") to preserve negation meaning. While this strategy strengthened semantic clarity, it also slightly compressed the negative review text into fewer tokens, reducing the overall token diversity compared to positive reviews.
4. Labeling method limitations: VADER, while chosen for consistency and reproducibility across models, introduces certain biases, particularly favoring positive sentiment. These biases were identified early and managed through strategies such as class balancing and stratified splits. Additional experiments were conducted to adjust the VADER threshold for positive classification, aiming to artificially boost the negative class size and achieve a more balanced training set. While these threshold adjustments slightly improved individual model performance, they introduced instability across the pipeline, ultimately leading to the decision to retain the original VADER threshold (0.05) for consistency across all models.

The performance of each model was evaluated using several standard metrics:

- Accuracy: Measures the overall percentage of correctly classified reviews. Suitable for initial overview, but sensitive to class imbalance.
- Precision, Recall, and F1-Score (Macro-Averaged): These metrics were calculated using macro-averaging to give equal weight to both positive and negative classes, addressing the known label imbalance.
- Precision evaluates how many predicted positives were correct.
- Recall measures how many actual positives were correctly predicted.
- F1-Score balances precision and recall, providing a harmonic mean of the two.
- Confusion Matrix: For each model, confusion matrices were analyzed to understand types of classification errors and reveal any systematic biases (e.g., false negatives or false positives).

Summary Table

Model	Accuracy	Macro Precision	Macro Recall	Macro F1-score	Notes
Logisitc Regression	97.0%	94.0%	95.0%	91.0%	Great results, fast training
Multinomial Naive Bayes	90.0%	88.0%	81.0%	84.0%	Very efficient, lightweight
Random Forest Classifier	94.0%	95.0%	89.0%	91.0%	Slower, feature importance insights
LSTM (Neural Network)	99.0%	98.0%	98.0%	98.0%	Best performance, captures sequence

Observations across models

- All models consistently struggled more with negative sentiment detection compared to positive sentiment.
- The LSTM model especially benefited from early stopping, preventing overfitting despite longer training times (if the model does not improves it will stop)
- Traditional machine learning models (LR, NB, RF) showed larger train–test gaps, suggesting that overfitting was more pronounced without deep learning regularization.

The decision to use VADER-generated labels with consistent preprocessing allowed a fair and reproducible comparison across all four models, ensuring that differences in performance were truly due to model architecture rather than inconsistent data handling.

Final Summary and Consequences

This project demonstrated a full machine learning workflow, from exploratory data analysis to model evaluation, on hotel review sentiment data. A strong emphasis was placed on consistent preprocessing, reproducibility, and systematic control of overfitting. VADER sentiment labeling provided a scalable, objective method for creating binary sentiment labels, which enabled fair model comparisons across Logistic Regression, Naive Bayes, Random Forest, and LSTM architectures. While VADER introduced some positive

bias into the labeling process, mitigation strategies such as balanced class weights and stratified sampling were applied to address this limitation. Model evaluations revealed that while traditional machine learning models (Logistic Regression, Naive Bayes, Random Forest) performed respectably, the LSTM neural network significantly outperformed the others by leveraging sequential dependencies in the text. Nevertheless, even simpler models achieved high performance with well-preprocessed data, demonstrating that strong baselines remain effective for text classification tasks. Challenges such as class imbalance, short and noisy review texts, and minor preprocessing artifacts were managed through informed engineering decisions. Some minor asymmetries between sentiment classes remained, but were traceable to inherent data properties rather than uncontrolled modeling flaws. In conclusion, this project highlights the importance of thoughtful data preparation, model selection based on task characteristics, and critical analysis of results. Future work could explore multi-class sentiment labeling (including neutral sentiment) and more advanced deep learning architectures if greater computational resources are available.

Task 2

Imports for task 2

In [ ]:
```
import torch
import torchvision
import torch.nn as nn
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from torch.utils.data import DataLoader
from torchvision import transforms, models
from sklearn.metrics import precision_score, recall_score, f1_score, ConfusionMatrixDisplay
from PIL import Image
```

Task 2 a. Train a CNN on CIFAR-10

In [ ]:
```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

Data preprocessing and Augmentation

Before training, we first preprocessed the CIFAR-10 images and applied augmentation to improve model generalization. Data augmentation helps machine learning models perform better by preventing overfitting, improve accuracy, and creating diversity in data (Awan, 2024). Some of the code for the data augmentation and training is inspired by Vikas Bhadorias high accuracy model on CIFAR-10 (2020). In this task, the following transformations are applied to each training image:

- Resize to **96 x 96**: This resolution streamlines the training.
- Random horizontal flip: Mirrors images randomly (an airplane is still an airplane when flipped).
- Random perspective transform and affine transforms: Applies slight geometric distortions (changes in perspective, shear, scale) to simulate different viewpoints. This makes the model robust to viewpoint changes.
- Color jitter and sharpness adjustment: Randomly alters brightness/contrast and sharpness. This simulates different lightning conditions and camera focus, improving the networks tolerance to variations.
- Tensor conversion and normalization: Images are converted to PyTorch tensors and normalized channel-wise using CIFAR-10 mean and st.deviation.

In [ ]:
```
transform_train = transforms.Compose([
    transforms.Resize((96, 96)),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomPerspective(distortion_scale=0.2, p=0.3),
    transforms.RandomAdjustSharpness(sharpness_factor=2),
    transforms.ColorJitter(brightness=0.2, contrast=0.2),
    transforms.RandomAffine(degrees=0, shear=10, scale=(0.8, 1.2)),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465),
                         (0.2023, 0.1994, 0.2010))
])

transform_test = transforms.Compose([
    transforms.Resize((96, 96)),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465),
                         (0.2023, 0.1994, 0.2010))
])
```

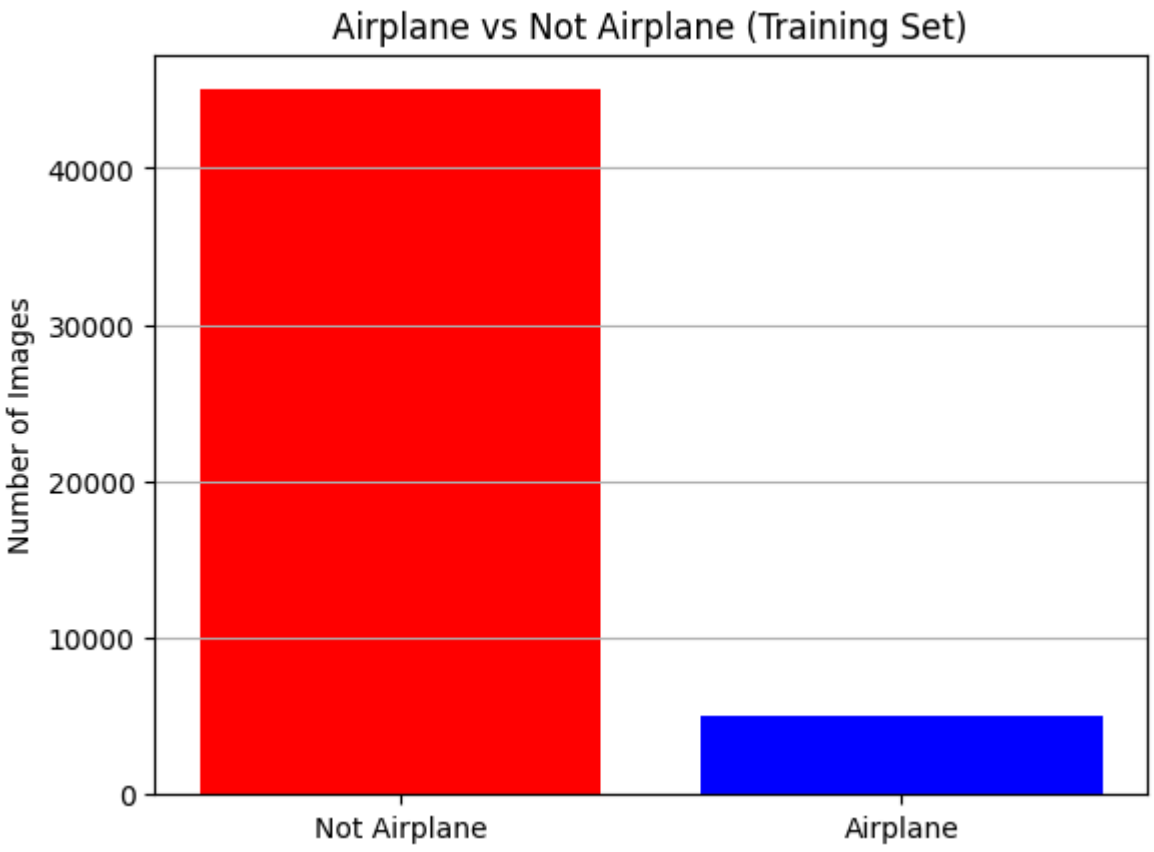
The precise rgb-values for CIFAR-10 are mean: ***0.4913, 0.4821*** *and* ***0.4465****, std:* ***0.2470, 0.2434*** *and* ***0.2615*** (Zahra, 2021).

The CIFAR-10 dataset consist of 60 000 images across 10 classes. 50 000 training images and 10 000 test images (Krizhevsky, 2009). In this task we simplify the classification to a binary problem; *distinguishing the airplane-class (labeled 0 in CIFAR-10) from all the other classes*. This means we relabel each image as 1 if it is an airplane, and 0 otherwise. This creates two classes: 'airplane' and 'not airplane'.

In [ ]:
```python
def relabel_airplane_binary(dataset):
    dataset.targets = [1 if label == 0 else 0 for label in dataset.targets]
    return dataset

train_dataset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform_train)
test_dataset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform_test)

train_dataset = relabel_airplane_binary(train_dataset)
test_dataset = relabel_airplane_binary(test_dataset)

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)
```

Class imbalance

In [ ]:
```python
labels = train_dataset.targets
classes, counts = np.unique(labels, return_counts=True)

plt.bar(['Not Airplane', 'Airplane'], counts, color=['red', 'blue'])
plt.title('Airplane vs Not Airplane (Training Set)')
plt.ylabel('Number of Images')
plt.grid(axis='y')
plt.show()
```



Class imbalance in the training set: 'Airplane' vs 'Not Airplane'. Only 5 000 of 50 000 training images are airplanes, while 45 000 images are other objects (animals, vehicles, etc.), making accuracy alone misleading.

This imbalance poses the challenge that the model could naively predict every image as 'Not Airplane' and be correct 90% of the time, eventhough it failed to predict any airplanes. In fact, accuracy can be a misleading metric on imbalanced data, as it may reflect the dominant class frequently (Akosa, 2017). To address this, we need to ensure our model pays attention to the minority class. We will do so by using a weighted loss function that harder punish mis classifications on the airplane class, than errors on non-airplane classes (Tantai, 2023). This way, mistakes in the rare class count more heavily, prompting the model to focus on detecting airplanes.

In [ ]:
```python
def unnormalize(img):
    mean = torch.tensor([0.4914, 0.4822, 0.4465]).view(3,1,1)
    std = torch.tensor([0.2023, 0.1994, 0.2010]).view(3,1,1)
    return img * std + mean

def show_images(dataset, label=1, n=5, title='Airplane' if 1 else 'Not-Airplane'):
    shown = 0
    plt.figure(figsize=(n*2, 2))
    i = 0
    while shown < n and i < len(dataset):
        img, lbl = dataset[i]
        if lbl == label:
            img = unnormalize(img)
            img = img.permute(1, 2, 0).numpy()
            plt.subplot(1, n, shown+1)
            plt.imshow(np.clip(img, 0, 1))
```

```
            plt.axis('off')
            plt.title(title)
            shown += 1
        i += 1
    plt.show()


show_images(train_dataset, label=1, title='Airplane')
show_images(train_dataset, label=0, title='Not-Airplane')
```

Visualization of the two classes 'Airplane' & 'Not Airplane'.

MobileNetV2

Rather thain training a CNN from scratch, we use transfer learning with a pre-trained model. Transfer learning is rather popular becuase it allows us to build accurate models in an effective way (Marcelino, 2018). Transfer learning works by training a network on a big dataset like ImageNet and then using those weights as the initial weights in a new classification task (Shorten et al., 2019). In this task we have chosen to use MobileNetV2, a lightweight CNN trained on ImageNet, a dataset containing over one million images.

In [ ]:
```
model = models.mobilenet_v2(weights=models.MobileNet_V2_Weights.DEFAULT)
model.classifier[1]=nn.Linear(model.last_channel, 1)
model = model.to(device)
```

Cross-entropy with logits

For training we have decided to use the Binary Cross-Entry logits loss function. This function is a loss function that combines a Sigmoid layer and the Binary Cross-Entropy loss function in one single class (PyTorch, n.d).

To account for class imbalance, we supply a positive class weight to the loss. We set positive weight = 9.0, reflecting that non-airplane images outnumber airplane images by roughly 9:1 in our training set. The idea is to make the model more sensitive to the airplane class by increasing cost of mis classification.

We trained the model using a learning rate of 0.0005 on 10 epochs using the adam optimizer. This value were selected based on a balance between good performance and training time. We chose not to freeze any layers in MobileNetV2, allowing the entire network to be fine-tuned on our binary clasification task.

In [ ]:
```
positive_weight = torch.tensor([9.0]).to(device)
criterion = nn.BCEWithLogitsLoss(pos_weight=positive_weight)
optimizer = optim.Adam(model.parameters(), lr=0.0005)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.5)
```

Training of the model

We train the model for 10 epochs and evaluate its performance the training set and the test set after each epoch. During training, we measure accuracy, precision, recall, and F1-score for the airplane class. These metrics gives us a clearer picture of performance on the airplane class.

- Precision tells us what fraction of images predicted as airplanes were actually airplanes.
- Recall tells us what fraction of actual airplane images were actually airplanes.
- F1-score is the harmonic mean of precision and recall.

We track these because a high overall accuracy could be achieved by simply predicting 'not airplane' every time.

In [ ]:
```python
def train_model(model, train_loader, test_loader, epochs=5, name=None):
    train_acc, test_acc = [], []
    train_prec, test_prec = [], []
    train_rec, test_rec = [], []
    train_f1, test_f1 = [], []
    train_loss, test_loss = [], []

    for epoch in range(epochs):
        model.train()
        y_true, y_pred = [], []
        total_loss = 0

        for imgs, labels in train_loader:
            imgs, labels = imgs.to(device), labels.float().unsqueeze(1).to(device)
            optimizer.zero_grad()
            outputs = model(imgs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            total_loss += loss.item()

            preds = (torch.sigmoid(outputs) > 0.5).float()
            y_true.extend(labels.cpu().numpy())
            y_pred.extend(preds.cpu().numpy())

        train_acc.append(np.mean(np.array(y_true) == np.array(y_pred)) * 100)
        train_prec.append(precision_score(y_true, y_pred, zero_division=0))
        train_rec.append(recall_score(y_true, y_pred, zero_division=0))
        train_f1.append(f1_score(y_true, y_pred, zero_division=0))
        train_loss.append(total_loss/len(train_loader))

        # Validation
        model.eval()
        y_true, y_pred = [], []
        total_val_loss = 0
        with torch.no_grad():
            for imgs, labels in test_loader:
                imgs, labels = imgs.to(device), labels.float().unsqueeze(1).to(device)
                outputs = model(imgs)
                loss=criterion(outputs, labels)
                total_val_loss += loss.item()
                preds = (torch.sigmoid(outputs) > 0.5).float()
                y_true.extend(labels.cpu().numpy())
                y_pred.extend(preds.cpu().numpy())

        test_acc.append(np.mean(np.array(y_true) == np.array(y_pred)) * 100)
        test_prec.append(precision_score(y_true, y_pred, zero_division=0))
        test_rec.append(recall_score(y_true, y_pred, zero_division=0))
        test_f1.append(f1_score(y_true, y_pred, zero_division=0))
        test_loss.append(total_val_loss/len(test_loader))

        print(f"Epoch {epoch+1}/{epochs} - Train Acc: {train_acc[-1]:.2f}% | Test Acc: {test_acc[-1]:.2f}% | F1: {t
        scheduler.step()

    if name:
        df = pd.DataFrame({
            'epoch':list(range(1, epochs + 1)),
            'train_loss': train_loss,
            'test_loss': test_loss,
            'train_acc': train_acc,
            'test_acc': test_acc,
            'train_prec': train_prec,
            'test_prec': test_prec,
            'train_rec': train_rec,
            'test_rec': test_rec,
            'train_f1': train_f1,
            'test_f1': test_f1
        })
        df.to_csv(f'{name}_metrics.csv', index=False)
        torch.save(model.state_dict(), f'{name}_model.pth')

train_model(model, train_loader, test_loader, 10, 'Finish')
```

```
Epoch 1/10 - Train Acc: 91.75% | Test Acc: 94.10% | F1: 0.7664 | Loss: 0.2250
Epoch 2/10 - Train Acc: 94.16% | Test Acc: 96.67% | F1: 0.8491 | Loss: 0.2308
Epoch 3/10 - Train Acc: 95.43% | Test Acc: 95.78% | F1: 0.8178 | Loss: 0.2233
Epoch 4/10 - Train Acc: 95.96% | Test Acc: 97.69% | F1: 0.8937 | Loss: 0.1426
Epoch 5/10 - Train Acc: 96.30% | Test Acc: 97.90% | F1: 0.9001 | Loss: 0.1943
Epoch 6/10 - Train Acc: 96.46% | Test Acc: 97.12% | F1: 0.8704 | Loss: 0.1573
Epoch 7/10 - Train Acc: 96.90% | Test Acc: 97.94% | F1: 0.9024 | Loss: 0.1576
Epoch 8/10 - Train Acc: 97.24% | Test Acc: 97.31% | F1: 0.8778 | Loss: 0.1465
Epoch 9/10 - Train Acc: 97.28% | Test Acc: 98.06% | F1: 0.9061 | Loss: 0.1920
Epoch 10/10 - Train Acc: 97.54% | Test Acc: 97.12% | F1: 0.8697 | Loss: 0.1731
```

After training for 10 epochs, our model achieves ~97% accuracy on the test set. More importantly, it achieves a high F1-score for the airplanes class, indicating a good balance of precision and recall. By the last epoch, the model's precision and recall for detecting

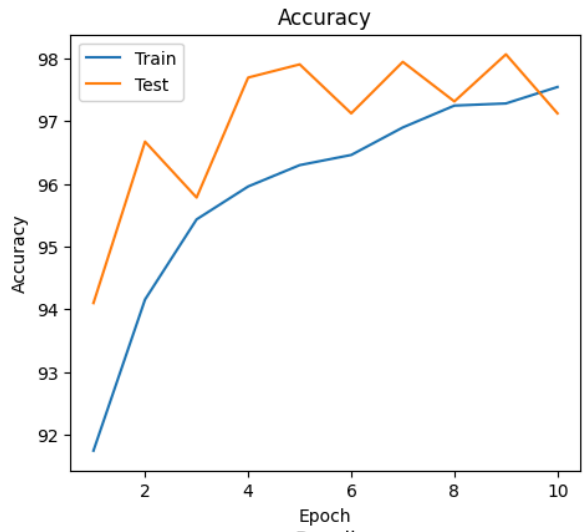
airplanes are both high. This means the model successfully identifies the majority of airplane images while keeping false alarms low. In comparison, a naive model that always predict "not airplane" would get 90% accuracy, but 0% recall for airplanes. Out model clearly outperformes thris, detecting a large portion of airplanes. The training and validation loss decreased over epochs, and the accuracy and F1 steadily improved before leveling off, suggesting the model learned effectively without severe overfitting.

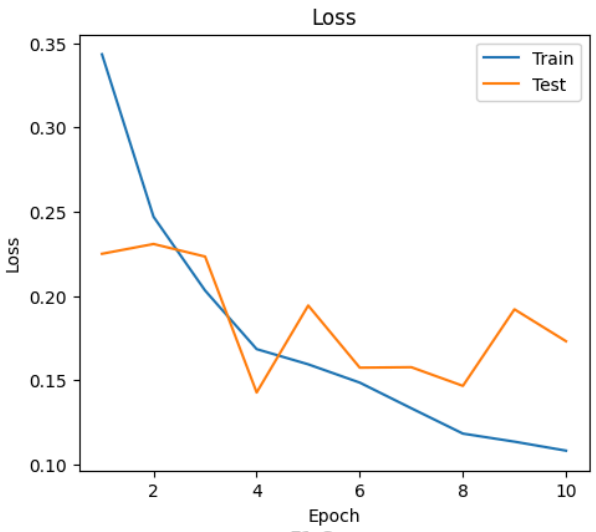
We can slo see that the training and test accuracy stayed fairly close, both at mid 90% by epoch number 10. This implies that our data augmentation strategy succeeded in preventing overfitting. The weighted loss was particularly important, as the model could have predicted the non-airplane class every time for an easy 90% accuracy. By looking at the epochs, from number 1 through 10, the results confirms that the network genuinely is learning to recognize airplanes in images.

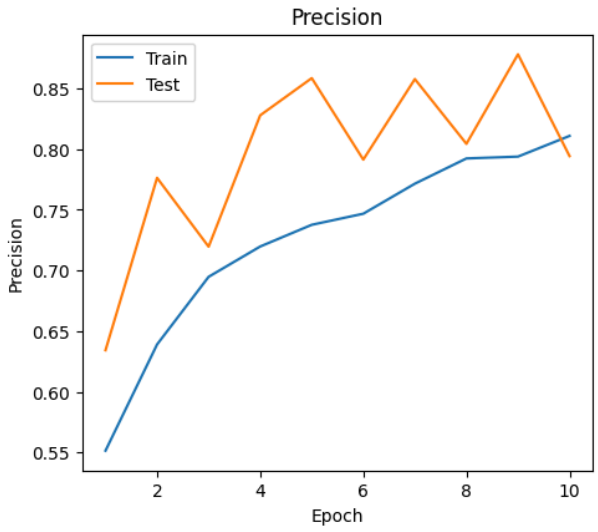
Graphs

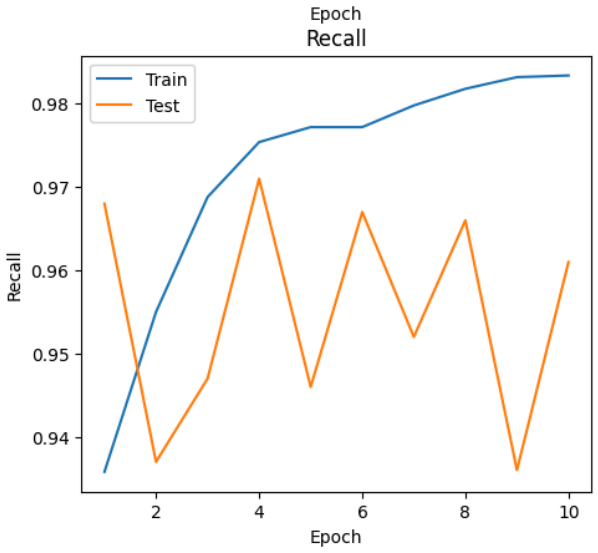
To further illustrate the training progress, we tracked the metrics across epochs. The plot of accuracy over epochs showed both training and test accuray climbing intro the mid 90%. The F1-score for the airplane class increased each epoch, reflecting better balance between precision and recall under training. The loss curve declined epoch by epoch. Overall, the metrics demonstrate that our transfer-learned CNN quickly reached high performance on this airplane vs not ariplane classification task.

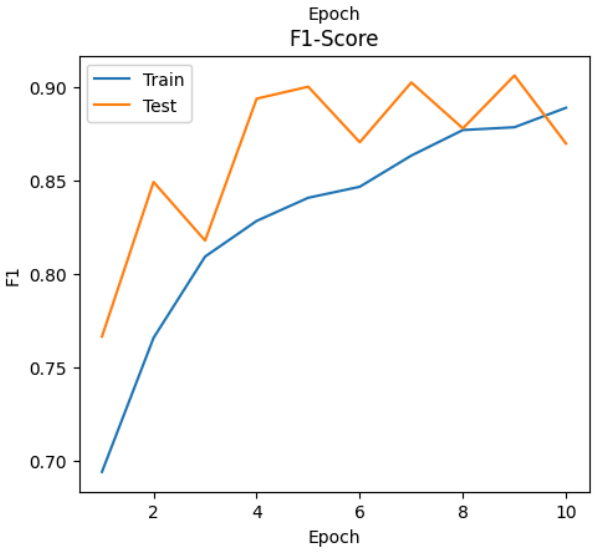
In [ ]:
```
df = pd.read_csv('Finish_metrics.csv')

plt.figure(figsize=(18, 10))

plt.subplot(2, 3, 1)
plt.plot(df['epoch'], df['train_acc'], label='Train')
plt.plot(df['epoch'], df['test_acc'], label='Test')
plt.title('Accuracy')
plt.xlabel('Epoch'); plt.ylabel('Accuracy'); plt.legend()

plt.subplot(2, 3, 2)
plt.plot(df['epoch'], df['train_loss'], label='Train')
plt.plot(df['epoch'], df['test_loss'], label='Test')
plt.title('Loss')
plt.xlabel('Epoch'); plt.ylabel('Loss'); plt.legend()

plt.subplot(2, 3, 3)
plt.plot(df['epoch'], df['train_prec'], label='Train')
plt.plot(df['epoch'], df['test_prec'], label='Test')
plt.title('Precision')
plt.xlabel('Epoch'); plt.ylabel('Precision'); plt.legend()

plt.subplot(2, 3, 4)
plt.plot(df['epoch'], df['train_rec'], label='Train')
plt.plot(df['epoch'], df['test_rec'], label='Test')
plt.title('Recall')
plt.xlabel('Epoch'); plt.ylabel('Recall'); plt.legend()

plt.subplot(2, 3, 5)
plt.plot(df['epoch'], df['train_f1'], label='Train')
plt.plot(df['epoch'], df['test_f1'], label='Test')
plt.title('F1-Score')
plt.xlabel('Epoch'); plt.ylabel('F1'); plt.legend()


plt.show()
```







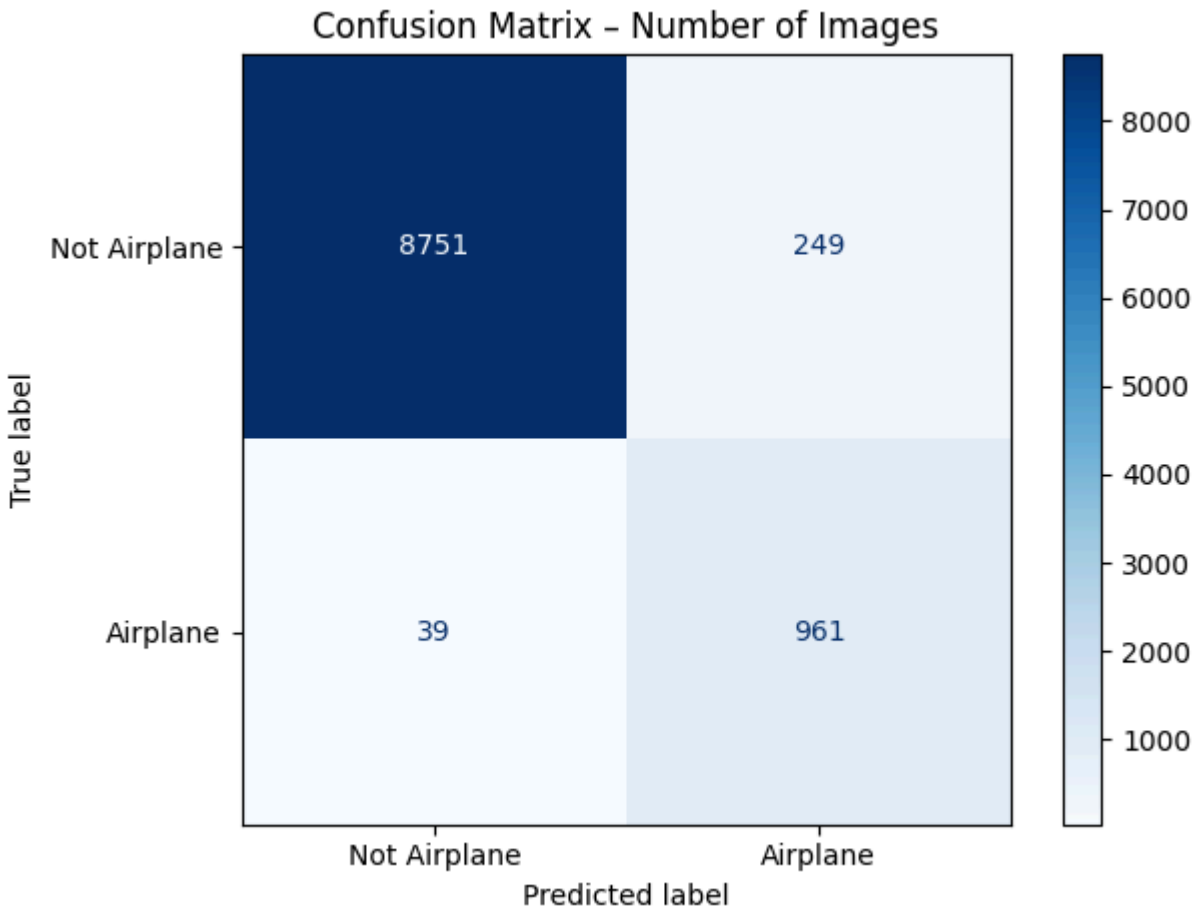


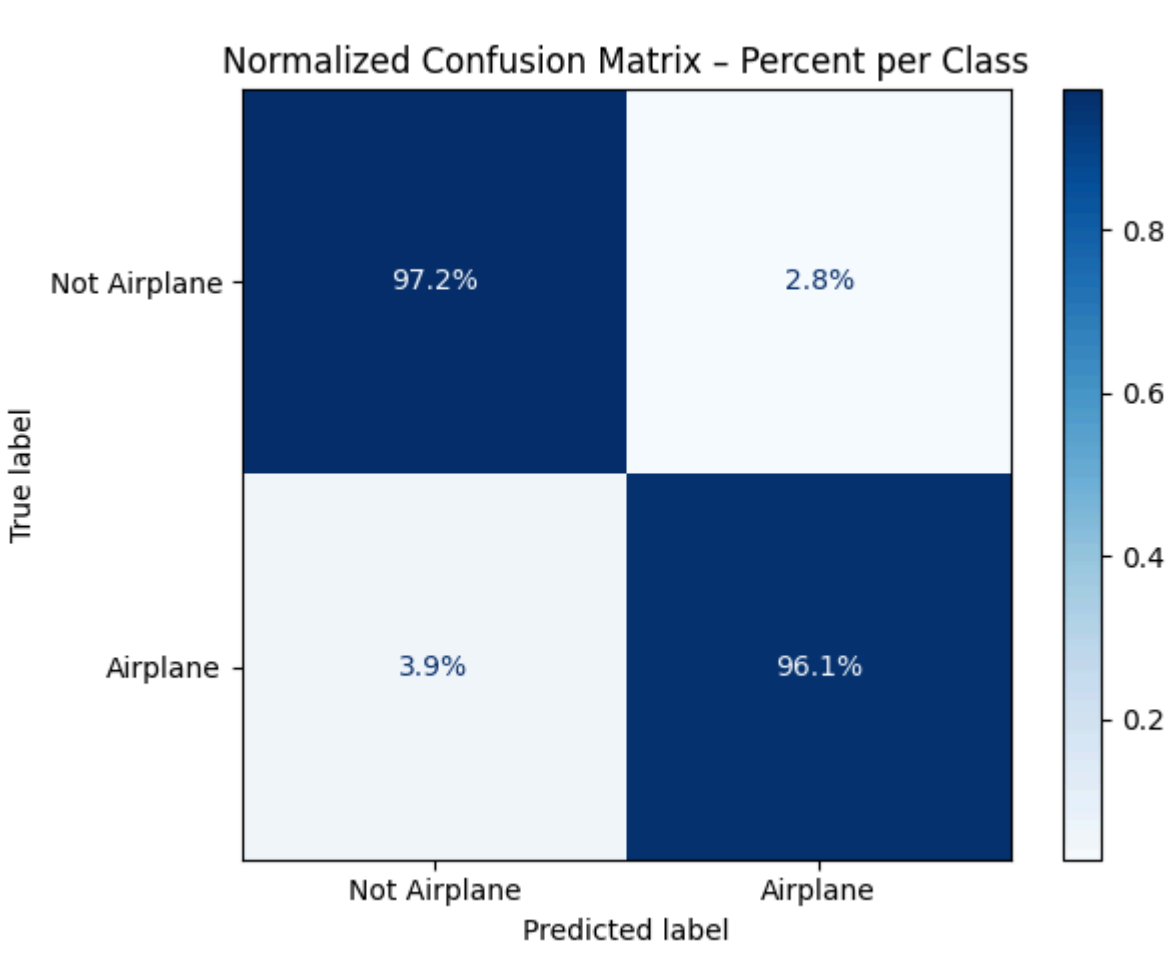


Confusion Matrix

In [ ]:
```python
def collect_preds(model, test_loader):
    y_true = []
    y_pred = []
    model.eval()

    with torch.no_grad():
        for images, labels in test_loader:
            images = images.to(device)
            outputs = model(images)
            probs = torch.sigmoid(outputs)
            preds = (probs > 0.5).float().cpu().numpy()
            y_pred.extend(preds)
            y_true.extend(labels.numpy())

    return y_true, y_pred

def show_confusion_matrix_raw(y_true, y_pred, class_names=["Not Airplane", "Airplane"]):
    ConfusionMatrixDisplay.from_predictions(
        y_true, y_pred,
        display_labels=class_names,
        cmap="Blues",
        normalize=None,
        values_format='d'
    )
    plt.title("Confusion Matrix – Number of Images")
    plt.tight_layout()
    plt.show()

def show_confusion_matrix_normalized(y_true, y_pred, class_names=["Not Airplane", "Airplane"]):
    ConfusionMatrixDisplay.from_predictions(
        y_true, y_pred,
        display_labels=class_names,
        cmap="Blues",
        normalize='true',
        values_format='.1%'
    )
    plt.title("Normalized Confusion Matrix – Percent per Class")
    plt.tight_layout()
    plt.show()
```

In [ ]:
```python
y_true, y_pred = collect_preds(model, test_loader)

show_confusion_matrix_raw(y_true, y_pred)
show_confusion_matrix_normalized(y_true, y_pred)
```





The top confusion matrix shows how many images are being predicted correctly airplane an not airplane. The confusion matrix below shows us a more tangible number, as it shows how many images are predicted correct and not in percentage of the class. The confusion matrix shows that the model classifies correctly in ~96/97% of the images.

Task 2 b. Predicting images

With the trained model, we can use it to predict new 'unseen' images. We define a helper function that loads and image file, applies the same preprocessing as our training data, and then runs our trained network to get a prediction. We also attach a confidence score for the prediction.

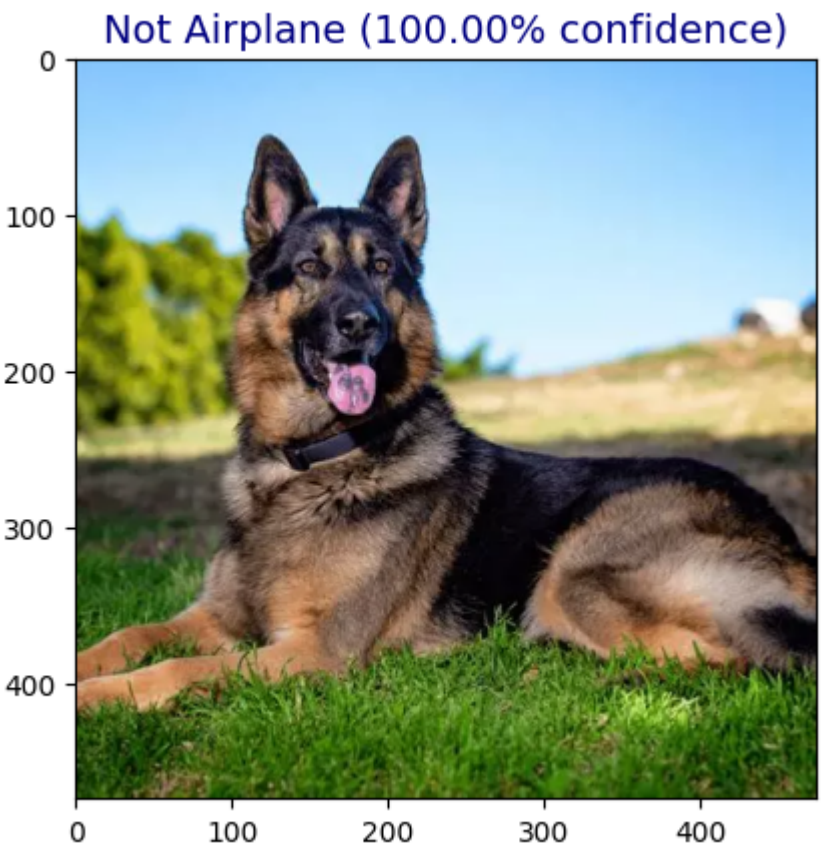
In [ ]:
```python
def predict_image(image_path, model_path='Finish_model.pth'):
    model = models.mobilenet_v2(weights=models.MobileNet_V2_Weights.DEFAULT)
    model.classifier[1] = torch.nn.Linear(model.last_channel, 1)
    model.load_state_dict(torch.load(model_path, map_location=device))
    model.to(device)
    model.eval()

    image = Image.open(image_path).convert("RGB")
    image_tensor = transform_test(image).unsqueeze(0).to(device)

    with torch.no_grad():
        output = model(image_tensor)
        prob = torch.sigmoid(output).item()

    label = "Airplane" if prob > 0.5 else "Not Airplane"
    confidence = prob * 100 if prob > 0.5 else (1 - prob) * 100
```

```
    plt.imshow(image)
    plt.axis()
    plt.title(f"{label} ({confidence:.2f}% confidence)", fontsize=14, color='navy')
    plt.show()

    return label, confidence
```

We load the trained model from the saved weights. The image is opened with PIL and cinverted to RGB, then fed through the same normalization pipeline as our training images. The model's output probability represents the confidence that the image is an airplane. Using this fucntion, we tested our model on a variety of images:

Dog

In [ ]:
```
predict_image('my_dog.jpg')
```

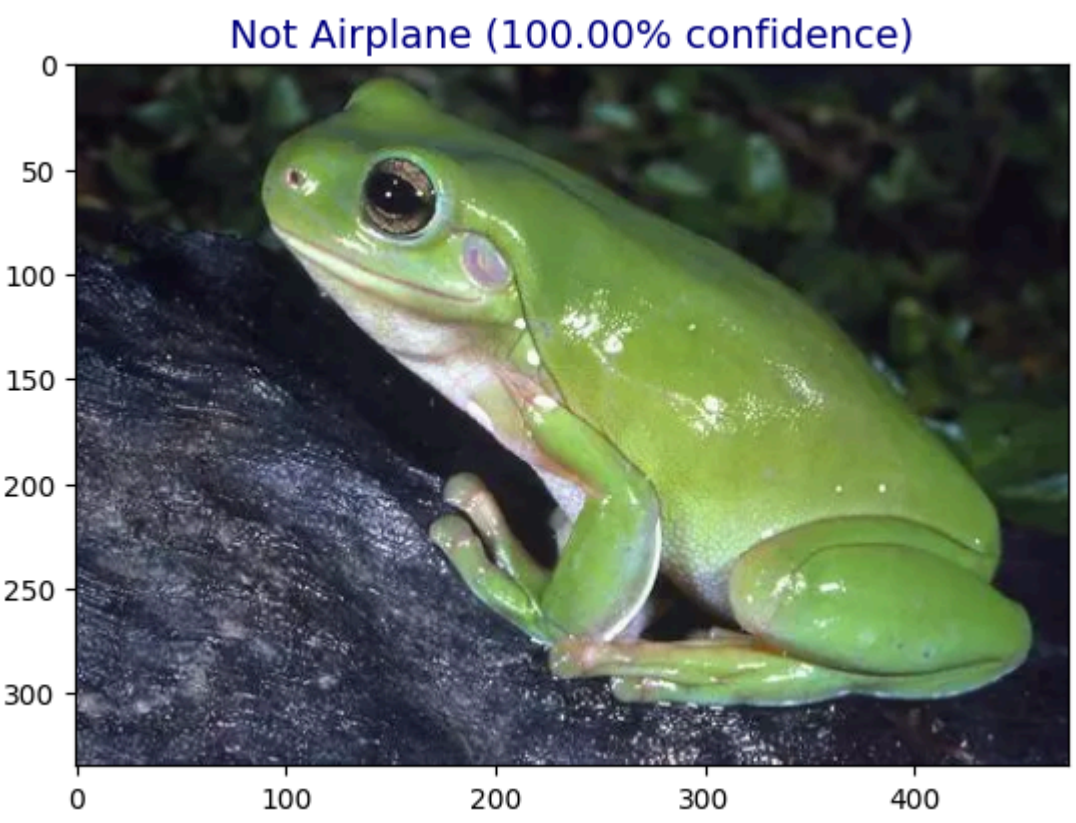


Out[ ]:
```
('Not Airplane', 99.99991297122506)
```

An image of a dog was predicted as 'Not Airplane' with over 99% confidence.

Frog

In [ ]:
```
predict_image('frog.webp')
```

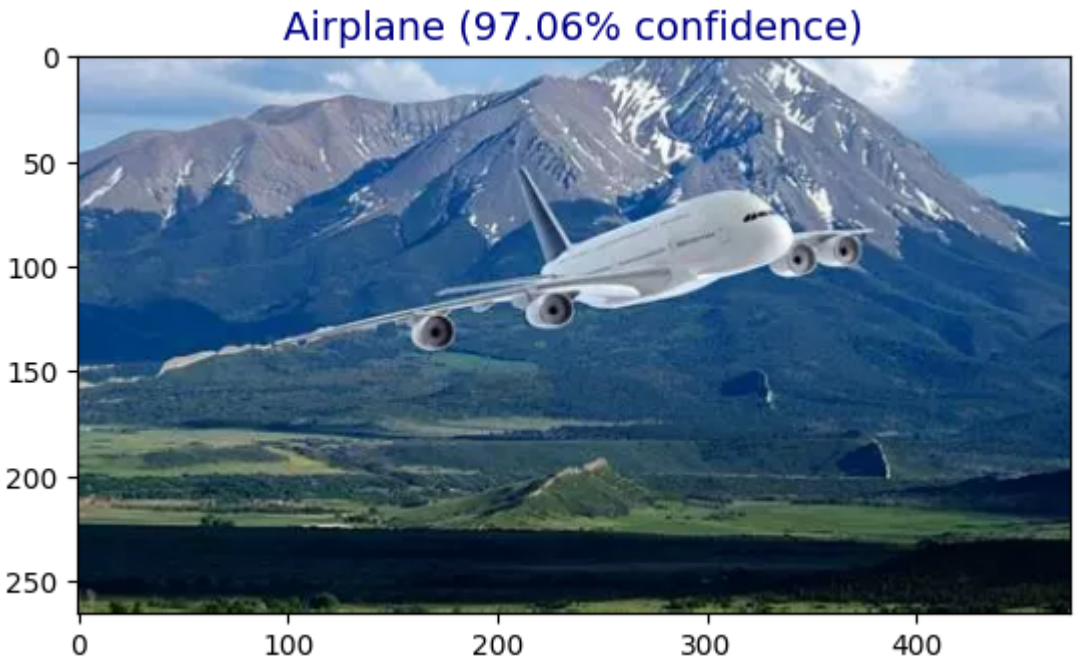


Out[ ]:
```
('Not Airplane', 99.99872758553465)
```

An image of a frog predicted 'Not Airplane' with over 99% confidence.

Plane with mountain background

In [ ]:
```
predict_image('airplane_1.webp')
```

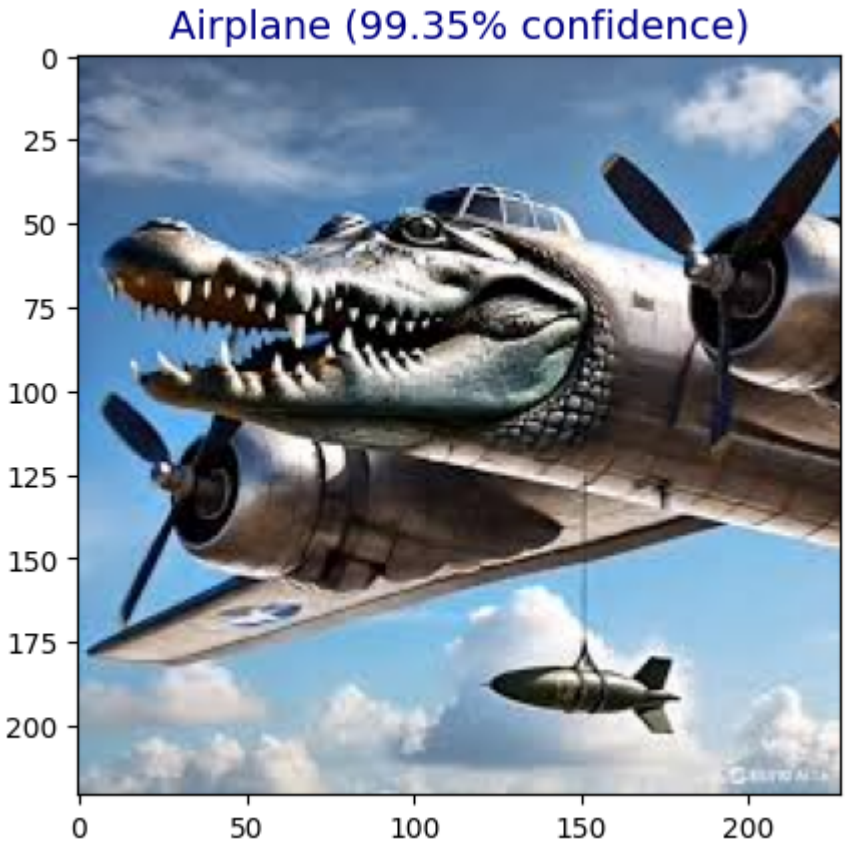


Out[ ]: ('Airplane', 97.05506563186646)

An image of an airplane flying in front of a mountain range was predicted 'Airplane' with 97% confidence.

Bombardino Crocodilo

In [ ]:
```
predict_image('croc_bomb.jpeg')
```



Out[ ]: ('Airplane', 99.35264587402344)

We even tried an image of the popular internet 'brainrot' meme, Bombardino Crocodilo. This AI Crocodile-faced airplane was predicted as an airplane with 99% confidence. Probably missing out on the airplanes 'face'.

Conclusion

In conclusion, we successfully trained a convolutional neural network to detect airplanes in the CIFAR-10 dataset using transfer learning. By employing data augmentation and a pre-trained MobileNetV2 model, we achievced robust performance with 10 training epochs. The model reached approximately 96% accuracy on the test set and was also able to correctly identify roughly 85-90% of airplanes images (high recall), while also showing high precision mislabeling very few non-airplanes as airplanes. This was accomplished by addressing the big class imbalance and applying a weighted loss to make the model focus on the minority airplane-class. We also tracked the F1-score during training to ensure balanced improvements. The result is an image classifier that reliably distinguishes airplanes form the other objects in the images.

Sources

- Akosa, J, S. (2017). *Predictive accuracy: A Misleading Performance Measure for Highly Imbalanced Data* [Online]. Oklahoma State University. Available at: https://support.sas.com/resources/papers/proceedings17/0942-2017.pdf
- Awan, A. A. 2024. *A Complete Guide to Data Augmentation* [Online]. Datacamp. Available at: https://www.datacamp.com/tutorial/complete-guide-data-augmentation

- Bhadoria, V. (2020). *Cifar10 high accuracy model on PyTorch* [Online]. Kaggle. Available at: https://www.kaggle.com/code/vikasbhadoria/cifar10-high-accuracy-model-build-on-pytorch/notebook
- Domingos, P. (2012). *A few useful things to know about machine learning* [Online]. Communications of the ACM, 55(10), 78–87. https://homes.cs.washington.edu/~pedrod/papers/cacm12.pdf
- Hii, D. (n.d.). *Using meaning specificity to aid negation handling in sentiment analysis* [Unpublished manuscript]. Computation of Language Laboratory, University of California, Irvine. https://sites.socsci.uci.edu/~lpearl/CoLaLab/papers/Hii2019_MeaningSpecNegSent.pdf
- Hutto, C. J., & Gilbert, E. (2014). *VADER: A parsimonious rule-based model for sentiment analysis of social media text* [Computer software]. GitHub. https://github.com/cjhutto/vaderSentiment
- Krizhevsky, A. (2009). *Learning Multiple Layers of Feautures from Tiny Images* [Online]. University of Toronto. Available at: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf
- Marcelino, P. (2018). *Transfer learning from pre-trained models*[Online]. Medium. Available at: https://medium.com/data-science/transfer-learning-from-pre-trained-models-f2393f124751
- MLU-Explain Team. (n.d.). *Visual explanation of logistic regression* [Online]. Retrieved from https://mlu-explain.github.io/logistic-regression/
- Müller, A. C., & Guido, S. (2016). *Introduction to machine learning with Python: A guide for data scientists*. O'Reilly Media.
- O'Neil, C., & Schutt, R. (2013). *Doing data science: Straight talk from the frontline* (Chapter 4). O'Reilly Media.
- PyTorch. (n. d). *BCEWithLogitsLoss* [Online]. PyTorch. https://pytorch.org/docs/stable/generated/torch.nn.BCEWithLogitsLoss.html
- Russell, S. J., & Norvig, P. (2002). *Artificial intelligence: A modern approach* (2nd ed.). Prentice Hall.
- Scikit-learn developers. (2023). *Scikit-learn: Machine learning in Python* [Online]. Retrieved from https://scikit-learn.org/stable/
- Shorten, C., & Khoshgoftaar, T. M. (2019). *A Survey on Image Data Augmentation for Deep Learning* [Online]. *Journal of Big Data*, 6(1), 60. Available at: https://doi.org/10.1186/s40537-019-0197-0
- Tantai, H. (2023, May 23). *Use weighted loss function to solve imbalanced data classification problems* [Online]. Medium. Available at: https://medium.com/@zergtant/use-weighted-loss-function-to-solve-imbalanced-data-classification-problems-749237f38b75
- TensorFlow. (2023). *Working with RNNs in Keras* [Online]. Retrieved from https://www.tensorflow.org/guide/keras/working_with_rnns
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gómez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). *Attention is all you need. Advances in Neural Information Processing Systems (Vol. 30)* [Online]. https://papers.nips.cc/paper/7181-attention-is-all-you-need
- Zahra. (2021, June 24). *How to calculate the mean and the std of cifar10 data* [Online]. Stack overflow. https://stackoverflow.com/questions/66678052/how-to-calculate-the-mean-and-the-std-of-cifar10-data?