

TF-IDF with n-grams and Machine Learning Models For NLP

Lecture 08

Prepared by

Dr. Hager Saleh

Import necessary libraries

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix, accuracy_score,
precision_score, recall_score, f1_score, classification_report
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
```

Reading dataset

```
df = pd.read_csv("clean_dataset.csv", encoding="ISO-8859-1")
```

```

df = df.groupby('sentiment', group_keys=False).apply(lambda x:
x.sample(n=1000, random_state=42))
df = df.reset_index(drop=True)

C:\Users\compumarts\AppData\Local\Temp\
ipykernel_15024\2195919672.py:1: DeprecationWarning:
DataFrameGroupBy.apply operated on the grouping columns. This behavior
is deprecated, and in a future version of pandas the grouping columns
will be excluded from the operation. Either pass
`include_groups=False` to exclude the groupings or explicitly select
the grouping columns after groupby to silence this warning.
  df = df.groupby('sentiment', group_keys=False).apply(lambda x:
x.sample(n=1000, random_state=42))

df.head()

  sentiment clean_text
0  negative      say bad trip angri
1  negative      sound forebod
2  negative  kat ive beg mum lt get attic wont let waaa ye ...
3  negative      realli realli bore guess go bed
4  negative      stuck finger throat bunch bump tongu throat

df=df.dropna()

df['sentiment'].value_counts()

sentiment
negative      999
positive      999
neutral       994
Name: count, dtype: int64

X = df['clean_text'].values
y = df['sentiment'].values

```

TF-IDF with Unigrams, Bigrams, and Trigrams

Three types of n-grams

- Unigrams: "machine", "learning", "fun"
- Bigrams: "machine learning", "deep learning"
- Trigrams: "machine learning is", "deep learning is"
- ngram_range=(min_n, max_n)
 - Unigrams(1,1)
 - Bigrams(2,2)

- Unigrams + Bigrams (1,2)
- Unigrams + Bigrams + Trigrams(1,3)

```
vectorizer = TfidfVectorizer(ngram_range=(1, 2)) # (1,2) = unigrams + bigrams
```

```
X = vectorizer.fit_transform(X)
```

Splitting data

```
# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split( X, y,
test_size=0.30, random_state=42, stratify=y)
```

Evaluation models

```
def evaluate_model(model_name, y_true, y_pred):

    # Calculate metrics
    accuracy = accuracy_score(y_true, y_pred)
    precision = precision_score(y_true, y_pred, average='weighted')
    recall = recall_score(y_true, y_pred, average='weighted')
    f1 = f1_score(y_true, y_pred, average='weighted')
    cm = confusion_matrix(y_true, y_pred)

    # Create a report
    report = classification_report(y_true, y_pred)

    # Output results
    metrics = {
        'Model Name': model_name,
        'Accuracy': accuracy,
        'Precision': precision,
        'Recall': recall,
        'F1 Score': f1,

        'Classification Report': report
    }

    # Plot Confusion Matrix
    plt.figure(figsize=(4, 4))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False,
                xticklabels=np.unique(y_true),
yticklabels=np.unique(y_true))
    plt.title(f'Confusion Matrix for {model_name}')
    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')
    plt.show()

    return metrics
```

Random Forest Classifier

```
# Define the Random Forest Classifier
rf_classifier = RandomForestClassifier(random_state=42)

# Define the hyperparameters for grid search
param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [None, 10, 20, 30],
}

# Initialize GridSearchCV
grid_search = GridSearchCV(estimator=rf_classifier,
    param_grid=param_grid, cv=5, scoring='accuracy', n_jobs=-1)

# Fit the grid search to the data
grid_search.fit(X_train, y_train)

# Get the best estimator from grid search
best_rf_classifier = grid_search.best_estimator_

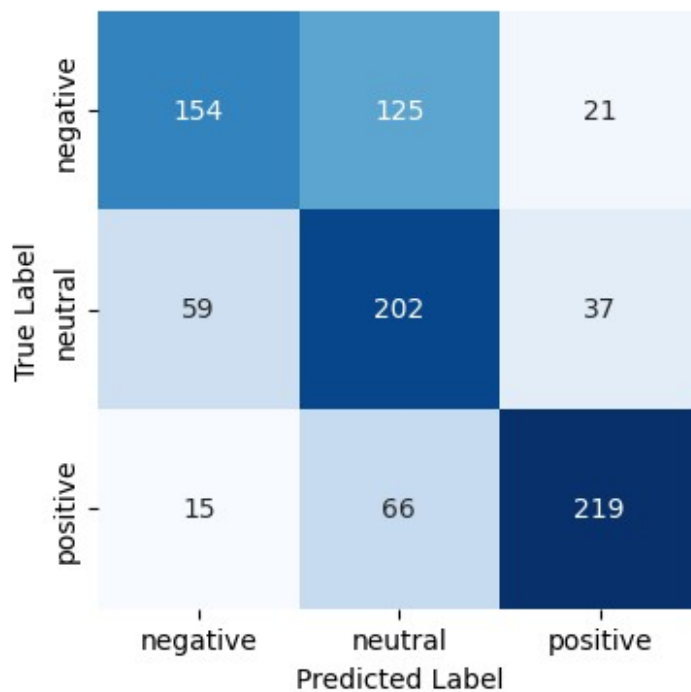
# Make predictions using the best model
y_pred = best_rf_classifier.predict(X_test)

# Evaluate the model
evaluation_results = evaluate_model('RandomForestClassifier', y_test,
    y_pred)

# Print the evaluation results
for key, value in evaluation_results.items():
    if key == 'Classification Report':
        print(value) # Print report separately for better readability
    else:
        print(f"{key}: {value:.4f}" if isinstance(value, float) else
            f"{key}: \n{value}")

# Print the best parameters found by grid search
print("\nBest hyperparameters found by GridSearchCV:")
print(grid_search.best_params_)
```

Confusion Matrix for RandomForestClassifier



Model Name:
RandomForestClassifier
Accuracy: 0.6403
Precision: 0.6603
Recall: 0.6403
F1 Score: 0.6425

	precision	recall	f1-score	support
negative	0.68	0.51	0.58	300
neutral	0.51	0.68	0.58	298
positive	0.79	0.73	0.76	300
accuracy			0.64	898
macro avg	0.66	0.64	0.64	898
weighted avg	0.66	0.64	0.64	898

Best hyperparameters found by GridSearchCV:
{'max_depth': 30, 'n_estimators': 300}

LogisticRegression

```
# Define the Logistic Regression model
LR = LogisticRegression(random_state=42)

# Define the hyperparameters for grid search
```

```

param_grid = {
    'solver': ['liblinear', 'lbfgs'],      # Solver type
}

# Initialize GridSearchCV
grid_search = GridSearchCV(estimator=LR, param_grid=param_grid, cv=5,
scoring='accuracy', n_jobs=-1)

# Fit the grid search to the data
grid_search.fit(X_train, y_train)

# Get the best estimator from grid search
best_LR = grid_search.best_estimator_

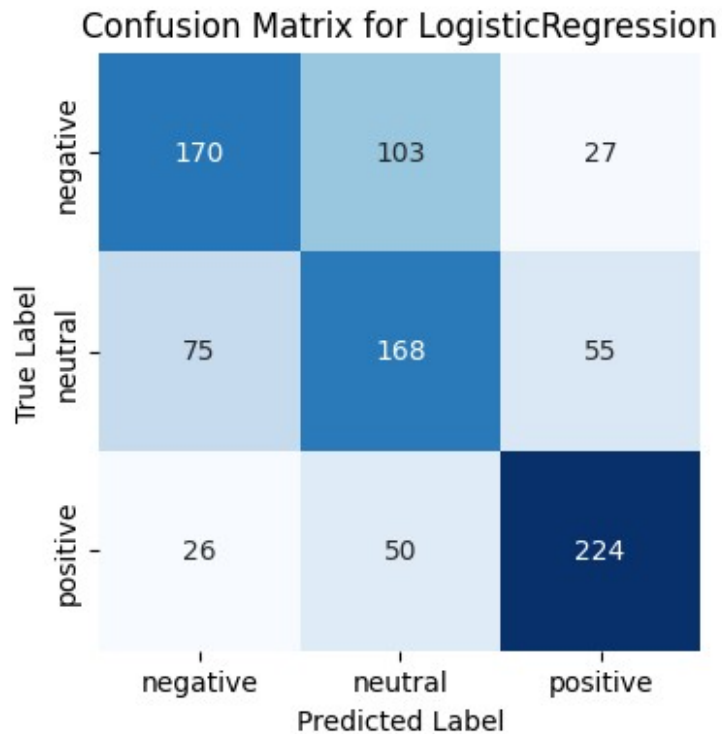
# Make predictions using the best model
y_pred = best_LR.predict(X_test)

# Evaluate the model
evaluation_results = evaluate_model('LogisticRegression', y_test,
y_pred)

# Print the evaluation results
for key, value in evaluation_results.items():
    if key == 'Classification Report':
        print(value) # Print report separately for better readability
    else:
        print(f"{key}: {value:.4f}" if isinstance(value, float) else
f"{key}: \n{value}")

# Print the best hyperparameters found by grid search
print("\nBest hyperparameters found by GridSearchCV:")
print(grid_search.best_params_)

```



Model Name:
LogisticRegression
Accuracy: 0.6258
Precision: 0.6278
Recall: 0.6258
F1 Score: 0.6260

	precision	recall	f1-score	support
negative	0.63	0.57	0.60	300
neutral	0.52	0.56	0.54	298
positive	0.73	0.75	0.74	300
accuracy			0.63	898
macro avg	0.63	0.63	0.63	898
weighted avg	0.63	0.63	0.63	898

Best hyperparameters found by GridSearchCV:
{'solver': 'liblinear'}

DecisionTreeClassifier

```
# Define the Decision Tree model
DT = DecisionTreeClassifier(random_state=42)

# Define the hyperparameters for grid search
param_grid = {
```

```

        'criterion': ['gini', 'entropy'],           # The function to
        measure the quality of a split
        'max_depth': [None, 10, 20, 30],          # The maximum depth
        of the tree
    }

    # Initialize GridSearchCV
    grid_search = GridSearchCV(estimator=DT, param_grid=param_grid, cv=5,
                               scoring='accuracy', n_jobs=-1)

    # Fit the grid search to the data
    grid_search.fit(X_train, y_train)

    # Get the best estimator from grid search
    best_DT = grid_search.best_estimator_

    # Make predictions using the best model
    y_pred = best_DT.predict(X_test)

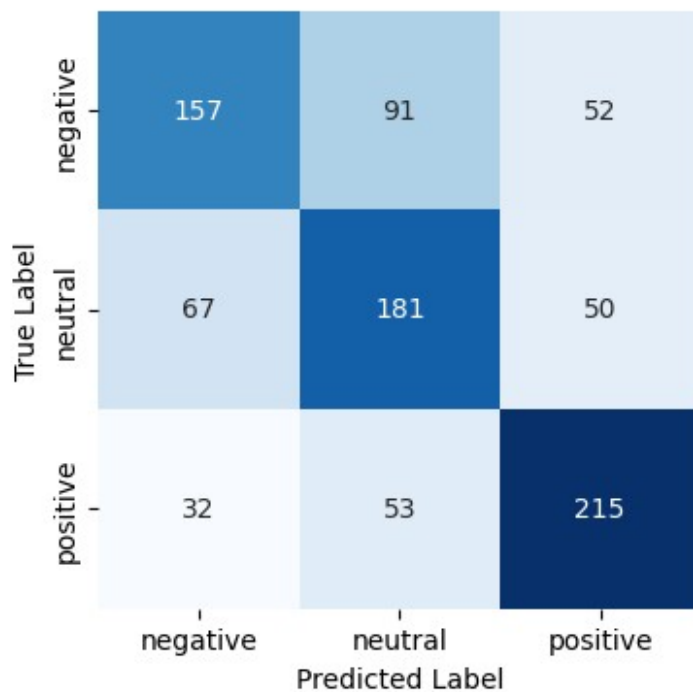
    # Evaluate the model
    evaluation_results = evaluate_model('DecisionTreeClassifier', y_test,
                                        y_pred)

    # Print the evaluation results
    for key, value in evaluation_results.items():
        if key == 'Classification Report':
            print(value) # Print report separately for better readability
        else:
            print(f"{key}: {value:.4f}" if isinstance(value, float) else
                  f"{key}: \n{value}")

    # Print the best hyperparameters found by grid search
    print("\nBest hyperparameters found by GridSearchCV:")
    print(grid_search.best_params_)

```

Confusion Matrix for DecisionTreeClassifier



Model Name:
DecisionTreeClassifier
Accuracy: 0.6158
Precision: 0.6163
Recall: 0.6158
F1 Score: 0.6143

	precision	recall	f1-score	support
negative	0.61	0.52	0.56	300
neutral	0.56	0.61	0.58	298
positive	0.68	0.72	0.70	300
accuracy			0.62	898
macro avg	0.62	0.62	0.61	898
weighted avg	0.62	0.62	0.61	898

Best hyperparameters found by GridSearchCV:
{'criterion': 'entropy', 'max_depth': None}

Support Vector Machine

```
SVM = SVC(random_state=42)

# Define the hyperparameters for grid search
param_grid = {
```

```

        'kernel': ['linear', 'rbf', 'poly'],      # Kernel type
    }

# Initialize GridSearchCV
grid_search = GridSearchCV(estimator=SVM, param_grid=param_grid, cv=5,
scoring='accuracy', n_jobs=-1)

# Fit the grid search to the data
grid_search.fit(X_train, y_train)

# Get the best estimator from grid search
best_SVM = grid_search.best_estimator_

# Make predictions using the best model
y_pred = best_SVM.predict(X_test)

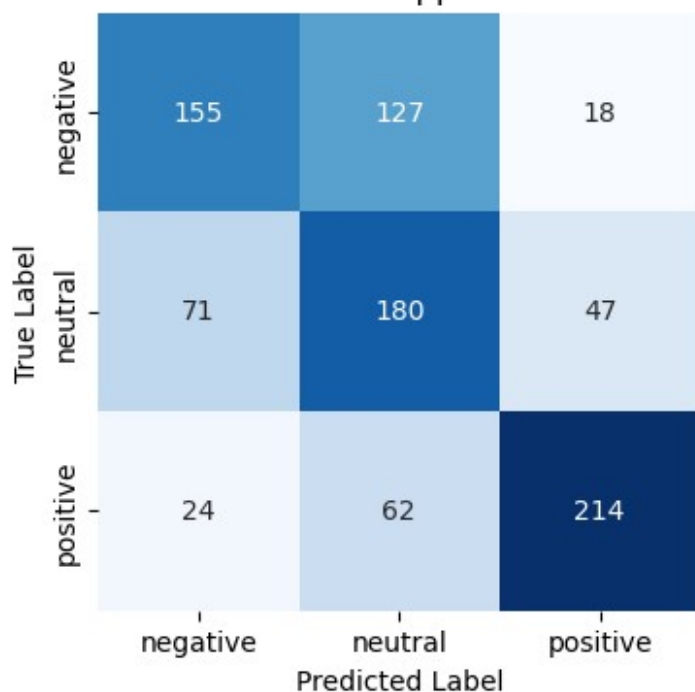
# Evaluate the model
evaluation_results = evaluate_model('Support Vector Machine', y_test,
y_pred)

# Print the evaluation results
for key, value in evaluation_results.items():
    if key == 'Classification Report':
        print(value) # Print report separately for better readability
    else:
        print(f"{key}: {value:.4f}" if isinstance(value, float) else
f"{key}: \n{value}")

# Print the best hyperparameters found by grid search
print("\nBest hyperparameters found by GridSearchCV:")
print(grid_search.best_params_)

```

Confusion Matrix for Support Vector Machine



Model Name:

Support Vector Machine

Accuracy: 0.6114

Precision: 0.6252

Recall: 0.6114

F1 Score: 0.6144

	precision	recall	f1-score	support
negative	0.62	0.52	0.56	300
neutral	0.49	0.60	0.54	298
positive	0.77	0.71	0.74	300
accuracy			0.61	898
macro avg	0.62	0.61	0.61	898
weighted avg	0.63	0.61	0.61	898

Best hyperparameters found by GridSearchCV:

```
{'kernel': 'linear'}
```

KNeighborsClassifier

```
# Define the K-Nearest Neighbors model
knn = KNeighborsClassifier()
```

```

# Define the hyperparameters for grid search
param_grid = {
    'n_neighbors': [1, 3, 5, 7, 9],          # Number of neighbors
    'weights': ['uniform', 'distance'],      # Weight function used
    'metric': ['euclidean', 'manhattan'],    # Distance metric
    'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute'] #
    # Algorithm to compute the nearest neighbors
}

# Initialize GridSearchCV
grid_search = GridSearchCV(estimator=knn, param_grid=param_grid, cv=5,
scoring='accuracy', n_jobs=-1)

# Fit the grid search to the data
grid_search.fit(X_train, y_train)

# Get the best estimator from grid search
best_knn = grid_search.best_estimator_

# Make predictions using the best model
y_pred = best_knn.predict(X_test)

# Evaluate the model
evaluation_results = evaluate_model('K-Nearest Neighbors', y_test,
y_pred)

# Print the evaluation results
for key, value in evaluation_results.items():
    if key == 'Classification Report':
        print(value) # Print report separately for better readability
    else:
        print(f"{key}: {value:.4f}" if isinstance(value, float) else
f"{key}: \n{value}")

# Print the best hyperparameters found by grid search
print("\nBest hyperparameters found by GridSearchCV:")
print(grid_search.best_params_)

```

```

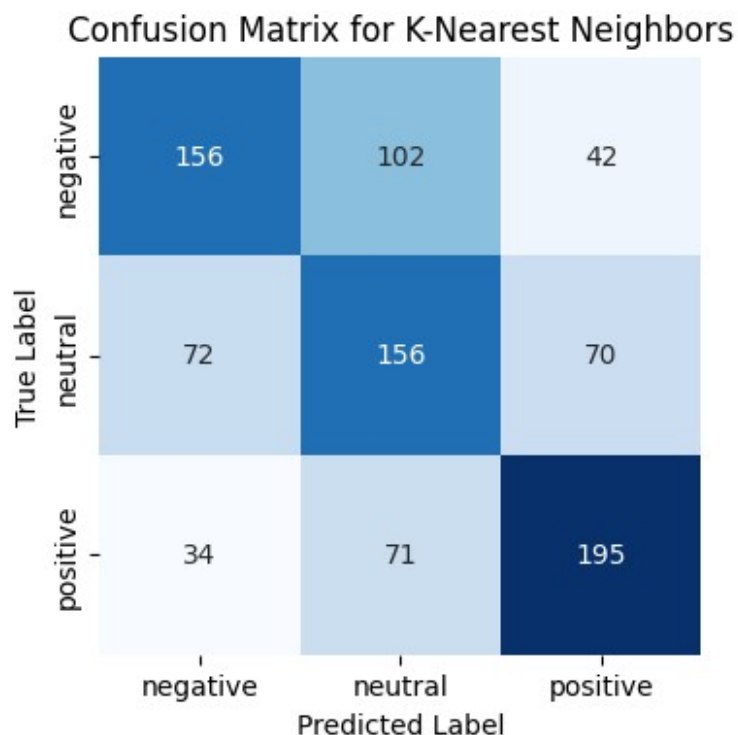
C:\Users\compumarts\anaconda3\envs\tensorflow\Lib\site-packages\
sklearn\model_selection\_search.py:1103: UserWarning: One or more of
the test scores are non-finite: [0.47469139 0.47469139 0.49570177
0.50239234 0.50669971 0.51481312
0.51482797 0.5191182 0.51721574 0.52772265 nan 0.43504699
nan 0.42741318 nan 0.38633109 nan 0.36293636
nan 0.3629375 0.47469139 0.47469139 0.49570177 0.50239234
0.50669971 0.51481312 0.51482797 0.5191182 0.51721574 0.52772265
nan 0.43504699 nan 0.42741318 nan 0.38633109
nan 0.36293636 nan 0.3629375 0.47469139 0.47469139

```

```

0.49570177 0.50239234 0.50669971 0.51481312 0.51482797 0.5191182
0.51721574 0.52772265          nan 0.43504699          nan 0.42741318
          nan 0.38633109          nan 0.36293636          nan 0.3629375
0.47469139 0.47469139 0.49570177 0.50239234 0.50669971 0.51481312
0.51482797 0.5191182 0.51721574 0.52772265          nan 0.43504699
          nan 0.42741318          nan 0.38633109          nan 0.36293636
          nan 0.3629375 ]
warnings.warn(

```



Model Name:
 K-Nearest Neighbors
 Accuracy: 0.5646
 Precision: 0.5685
 Recall: 0.5646
 F1 Score: 0.5652

	precision	recall	f1-score	support
negative	0.60	0.52	0.56	300
neutral	0.47	0.52	0.50	298
positive	0.64	0.65	0.64	300
accuracy			0.56	898
macro avg	0.57	0.56	0.57	898
weighted avg	0.57	0.56	0.57	898

```
Best hyperparameters found by GridSearchCV:  
{'algorithm': 'auto', 'metric': 'euclidean', 'n_neighbors': 9,  
'weights': 'distance'}
```