

Entire Implementation Approach of Automating News Article Classification with Machine Learning Based on Document Classification Aspects



School of Computing, Engineering & Digital Technologies

Teesside University

Middlesbrough TS1 3BA

Author: Michael Zats (18361063/2)

Supervisor: PhD Bohus Ziskal

Contents

List of Figures	iii
1 Entire Literature Review	1
1.1 Entire Implementation Approach	1
1.1.1 Data Uploading and Data Preprocessing	2
1.1.2 ML Models Creation and Execution of the Experiment	4
1.1.3 Models' Evaluation	27
1.1.4 Online App Deployment	32

List of Figures

1	Entire Literature Review	1
1.1	Project Execution Model	2
1.2	Deployed Online Classification App Based on SVM	40

1 Entire Literature Review

1.1 Entire Implementation Approach

The project is executed inside the paid version of Google Colab environment, Google Colab Pro, as in order to run the project within various datasets it was needed to have a higher RAM (Random-access memory) as the free version gave not sufficient enough for such size project size of 12 GB. The datasets were chosen for the project execution were the secondary data from the website Kaggle, (10)Dataset Text Document Classification, and BBC Full Text Document Classification. Another point to mention, is that due to the amount of data difference, the computation of BBC Full Text Document Classification took twice as much time as from (10)Dataset Text Document Classification.

The project is ran with Python 3 and various of libraries mentioned before in Tools Used for the Product. The models selected for the project execution were Support vector machine, Random Forest, Decision Tree, Naive Bayes and Deep Neural Network. So to evaluate the efficiency of the models, we choose the following evaluation methods: Precision, Recall, Accuracy and F1-Score, and for our task, the F1-Score emerged as the most crucial metric. Project Execution Model in Figure 4.1 leads through all the steps taken.

1 Entire Literature Review

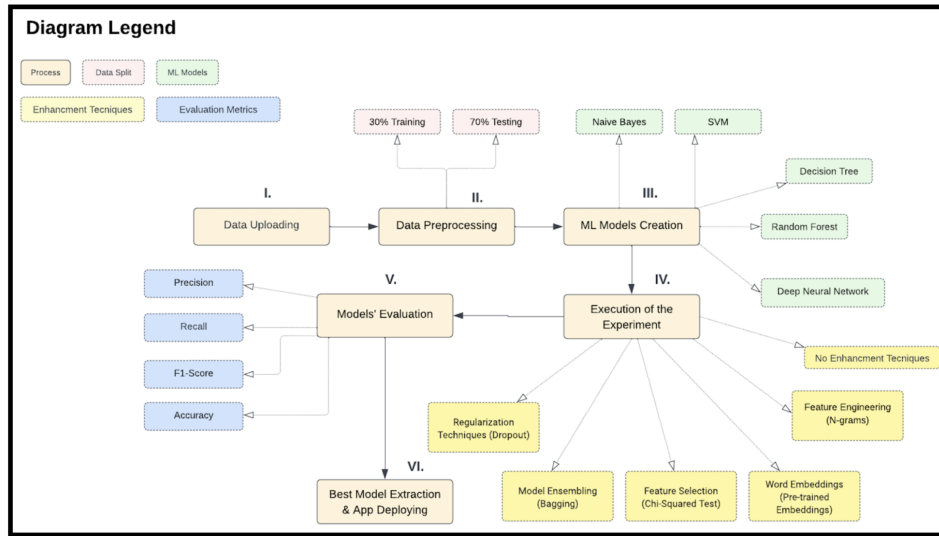


Figure 1.1: Project Execution Model

1.1.1 Data Uploading and Data Preprocessing

Regarding the project execution, firstly inside of the google colab environment it was necessary to install all the required libraries, so we used the following command to install them:

```
!pip install numpy pandas statsmodels scikit-learn matplotlib  
seaborn nltk gensim tensorflow keras==2.12.0
```

following the initial step, the required libraries and API were imported, the list of the imported libraries could be seen before in Librarbies and APIs

After importing the libraries, the project started its experiment part, and so, as we had two datasets to explore, one with 1.8 MB and the other with 3.7MB of data. The same approach was used as for the first as well as for the second dataset. At the beginning of the experiment part, it was needed to upload the zipped files where the data is hold, then to unzip it and extract each category. Meanwhile, for the purpose of the ML project execution, the data was split to 30% for training and 70% for testing. The following code

represents the mentioned before:

```
# Define the base path where the 'bbc' folder is located
# Make sure this path points directly to the folder containing the
  category folders.
base_path = Path('DATASET_10') # Replace with the correct path to
  your 'bbc' folder
# Dynamically list categories based on the directories in the 'bbc
  ' folder
categories = [f.name for f in base_path.iterdir() if f.is_dir()]
texts = []
labels = []
print(f"Found categories: {categories}")
# Iterate over the categories and read the files
for category in categories:
    category_path = base_path / category
    # Print the current category path for debugging
    print(f"Reading files from category: {category}")

    for file_path in category_path.iterdir():
        if file_path.is_file() and not file_path.name.startswith('.')
            .'):
            # Print each file path being read for debugging
            print(f"Reading file: {file_path}")

            with open(file_path, 'r', encoding='utf-8', errors='
                ignore') as file:
                texts.append(file.read())
                labels.append(category)
# Check if we have collected any texts and labels
if not texts or not labels:
```

1 Entire Literature Review

```
raise ValueError("No texts or labels have been collected.  
Please check your directory paths and file permissions.")  
# Splitting the dataset into training and testing sets  
X_train, X_test, y_train, y_test = train_test_split(texts, labels,  
                                                    test_size=0.3, random_state=42)
```

1.1.2 ML Models Creation and Execution of the Experiment

After the data splitting, we started from running the project without enhancement techniques to make sure that we have already benchmarks to compare the models with the enhancement techniques afterwards. A methodology that integrates both conventional machine learning algorithms and a bespoke deep learning model to undertake text classification was employed.

This section begins with the preprocessing of text data through Term Frequency-Inverse Document Frequency (TF-IDF) vectorization, specifically tailored to extract unigram features. By employing the `TfidfVectorizer` from the `sklearn.feature_extraction.text` module, configured to exclude English stop words and focus on unigrams (`ngram_range=(1,1)`), we transform our textual data into a numerical format that is conducive to machine learning. This process is applied to both our training and testing datasets, converting the resulting sparse matrices into dense arrays to facilitate compatibility with the Keras neural network framework and other machine learning models.

```
tfidf_vectorizer_ngrams = TfidfVectorizer(stop_words='english',  
                                           ngram_range=(1,1))  
X_train_tfidf_ngrams = tfidf_vectorizer_ngrams.fit_transform(  
    X_train).toarray()  
X_test_tfidf_ngrams = tfidf_vectorizer_ngrams.transform(X_test).  
    toarray()
```

Following the data preparation, a simple feed-forward neural network is defined using

1.1 Entire Implementation Approach

Keras to explore the potential of deep learning in text classification. This network comprises an input layer, two hidden layers with ReLU activation, dropout layers for regularization, and a softmax output layer, which is dimensioned according to the number of unique classes in our dataset. This design is aimed at mitigating overfitting while ensuring the network is capable of learning complex patterns in the data.

The exploration extends beyond deep learning to include a variety of traditional machine learning models such as Naive Bayes, SVM with a linear kernel, Random Forest, and Decision Tree. Each model, including the custom deep neural network, is trained on the TF-IDF transformed training data and subsequently evaluated on the test set. The performance of each model is meticulously analyzed using metrics such as precision, recall, F1-score, and accuracy, facilitated by the `classification_report` function from `sklearn.metrics`. This comprehensive evaluation strategy enables a nuanced comparison of the models' effectiveness in classifying text data.

```
df_results = train_and_evaluate_models(X_train_tfidf_ngrams ,
                                       X_test_tfidf_ngrams)
print(df_results)
```

Concluding this phase, the results of the model evaluations are collated into a pandas DataFrame, offering a succinct and comparative view of the performance metrics across all models. This structured presentation not only underscores the relative strengths and weaknesses of each classification strategy but also provides a foundational basis for further analysis and discussion within the thesis. As a result, the entire chunk of the code is represented below.

```
# Using only unigrams
tfidf_vectorizer_ngrams = TfidfVectorizer(stop_words='english',
                                           ngram_range=(1,1))
X_train_tfidf_ngrams = tfidf_vectorizer_ngrams.fit_transform(
    X_train).toarray() # Convert to array for Keras
```

1 Entire Literature Review

```
X_test_tfidf_ngrams = tfidf_vectorizer_ngrams.transform(X_test).
    toarray()

# Define a simple feed-forward neural network for Keras
def create_nn_model():
    model = Sequential()
    model.add(Dense(128, input_dim=X_train_tfidf_ngrams.shape[1],
        activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(64, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(len(set(y_train)), activation='softmax')) #
        Number of classes
    model.compile(loss='sparse_categorical_crossentropy',
        optimizer='adam', metrics=['accuracy'])
    return model

def train_and_evaluate_models(X_train, X_test):
    # Define the models
    models = {
        'Naive Bayes': MultinomialNB(),
        'SVM': SVC(kernel='linear'),
        'Random Forest': RandomForestClassifier(random_state=42),
        'Decision Tree': DecisionTreeClassifier(random_state=42),
        'Deep Neural Network': KerasClassifier(build_fn=
            create_nn_model, epochs=10, batch_size=32, verbose=0)
    }

    # Train and evaluate the models
    results = {}
    for model_name, model in models.items():
        # Train
        model.fit(X_train, y_train)
```

```

# Predict
y_pred = model.predict(X_test)

# Evaluate
report = classification_report(y_test, y_pred, output_dict
                              =True)
results[model_name] = {
    'Precision': report['weighted avg']['precision'],
    'Recall': report['weighted avg']['recall'],
    'F1-Score': report['weighted avg']['f1-score'],
    'Accuracy': report['accuracy'],
    'Support': report['weighted avg']['support']}
}

# Convert results to DataFrame for display
df_results = pd.DataFrame(results).transpose()
return df_results

# Train and evaluate models with Unigrams
df_results = train_and_evaluate_models(X_train_tfidf_ngrams,
                                       X_test_tfidf_ngrams)
print(df_results)

```

Following the initial evaluation of text classification models using unigrams, the study progressed to incorporate an enhancement technique involving the use of n-grams, specifically bigrams, in the preprocessing stage. This method aimed to capture not only the individual words (unigrams) but also pairs of consecutive words (bigrams), thereby enriching the feature set with contextual information that could potentially improve the models' understanding of the textual data.

To implement this enhancement, the TF-IDF vectorization process was adjusted to consider both unigrams and bigrams. This adjustment was made by modifying the `ngram_range` parameter of the `TfidfVectorizer` from (1,1) to (1,2), allowing the vector-

1 Entire Literature Review

izer to generate a feature matrix that includes both single words and their adjacent pairs. This enriched feature matrix was anticipated to provide a more nuanced representation of the text, capturing syntactic structures and phrases that single words alone might miss.

```
tfidf_vectorizer_ngrams = TfidfVectorizer(stop_words='english',
                                           ngram_range=(1,2))
X_train_tfidf_ngrams = tfidf_vectorizer_ngrams.fit_transform(
    X_train).toarray()
X_test_tfidf_ngrams = tfidf_vectorizer_ngrams.transform(X_test).
    toarray()
```

With the new TF-IDF matrices incorporating bigrams, the neural network architecture remained consistent with the initial approach, emphasizing the flexibility and adaptability of the model to accommodate the expanded input dimensionality. The network continued to feature an input layer that dynamically adjusted to the size of the feature set, intermediate layers with dropout to combat overfitting, and an output layer tailored for multi-class classification.

The models, now prefixed with 'N-grams' to denote their training on the enriched dataset, were subjected to the same rigorous training and evaluation process. This process involved training each model on the bigram-enriched training dataset and subsequently evaluating their performance on the test dataset. The evaluation metrics—Precision, Recall, F1-Score, Accuracy, and Support—were once again computed for each model, allowing for a direct comparison between the outcomes of the original unigram-based approach and the enhanced bigram approach.

The expectation was that the incorporation of bigrams would lead to a discernible improvement in model performance, given the additional linguistic context provided by adjacent word pairs. This hypothesis was systematically tested through the comparison of results, which were meticulously compiled into a DataFrame for clear and concise analysis.

```
df_results_ngrams = train_and_evaluate_models(X_train_tfidf_ngrams
, X_test_tfidf_ngrams)
print(df_results_ngrams)
```

The exploration of bigrams as an enhancement technique underscores the thesis's commitment to leveraging advanced preprocessing methods to improve the efficacy of machine learning models in text classification tasks. Through this iterative approach, the study aimed to uncover the extent to which the inclusion of n-grams can augment the predictive capabilities of both traditional and deep learning models, thereby contributing valuable insights to the field of natural language processing. The representative entire chunk of code is seen below.

```
# Using bigrams
tfidf_vectorizer_ngrams = TfidfVectorizer(stop_words='english',
ngram_range=(1,2))
X_train_tfidf_ngrams = tfidf_vectorizer_ngrams.fit_transform(
X_train).toarray() # Convert to array for Keras
X_test_tfidf_ngrams = tfidf_vectorizer_ngrams.transform(X_test).
toarray()
# Define a simple feed-forward neural network for Keras
def create_nn_model():
    model = Sequential()
    model.add(Dense(128, input_dim=X_train_tfidf_ngrams.shape[1],
activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(64, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(len(set(y_train)), activation='softmax')) #
    Number of classes
    model.compile(loss='sparse_categorical_crossentropy',
optimizer='adam', metrics=['accuracy'])
```

1 Entire Literature Review

```
    return model

def train_and_evaluate_models(X_train, X_test):
    # Define the models
    models = {
        'N-grams Naive Bayes': MultinomialNB(),
        'N-grams SVM': SVC(kernel='linear'),
        'N-grams Random Forest': RandomForestClassifier(
            random_state=42),
        'N-grams Decision Tree': DecisionTreeClassifier(
            random_state=42),
        'N-grams Deep Neural Network': KerasClassifier(build_fn=
            create_nn_model, epochs=10, batch_size=32, verbose=0)
    }
    # Train and evaluate the models
    results = {}
    for model_name, model in models.items():
        # Train
        model.fit(X_train, y_train)
        # Predict
        y_pred = model.predict(X_test)
        # Evaluate
        report = classification_report(y_test, y_pred, output_dict
            =True)
        results[model_name] = {
            'Precision': report['weighted avg']['precision'],
            'Recall': report['weighted avg']['recall'],
            'F1-Score': report['weighted avg']['f1-score'],
            'Accuracy': report['accuracy'],
            'Support': report['weighted avg']['support']
        }
    }
```



```

# Convert results to DataFrame for display
df_results = pd.DataFrame(results).transpose()
return df_results

# Train and evaluate models with N-grams
df_results_ngrams = train_and_evaluate_models(X_train_tfidf_ngrams
, X_test_tfidf_ngrams)
print(df_results_ngrams)

```

Following the exploration of text classification using TF-IDF vectorization with unigrams and bigrams, the study further advances by integrating an enhancement technique through the utilization of pre-trained word embeddings, specifically **Word2Vec**, to capture semantic relationships between words more effectively. This approach signifies a pivotal shift towards leveraging distributed representations of text, aiming to encapsulate the contextual nuances that simpler vectorization methods may overlook.

The process begins by transforming the textual data into vectors using the **Word2Vec** model. Each document is tokenized, and words not present in the **Word2Vec** model's vocabulary are filtered out. For words within the model's scope, their corresponding vectors are retrieved and averaged to represent the entire document. This method ensures that each document is encapsulated by a dense vector, capturing the semantic essence of the text based on the collective representation of its words.

```

X_train_word2vec = np.array([document_to_word2vec(doc,
word2vec_model) for doc in X_train]) X_test_word2vec = np.array
([document_to_word2vec(doc, word2vec_model) for doc in X_test])

```

To accommodate the nuanced representations derived from **Word2Vec**, a neural network model is designed, mirroring the structure used in previous experiments but adapted to handle the input dimensionality specific to **Word2Vec** vectors. The network comprises an initial dense layer with 128 neurons, followed by dropout layers for regularization, and culminates in an output layer tailored to the number of classification categories, utilizing

1 Entire Literature Review

the softmax activation function.

This setup is subjected to the same rigorous evaluation framework applied in previous phases, comparing a suite of machine learning models adapted to work with the **Word2Vec** feature representations. These models include variations of Naive Bayes, SVM, Random Forest, Decision Tree, and a Deep Neural Network, each prefixed with 'Pre-trained Embeddings' to denote their adaptation to the **Word2Vec**-enhanced dataset.

The comprehensive evaluation of these models, conducted on the **Word2Vec**-transformed datasets, yields a matrix of performance metrics—precision, recall, F1-score, accuracy, and support—offering a granular view into the efficacy of each algorithm in harnessing the semantic richness of **Word2Vec** embeddings for text classification.

```
df_results_word2vec = train_and_evaluate_models(X_train_word2vec ,
        X_test_word2vec) print(df_results_word2vec)
```

This segment of the study underscores the value of pre-trained word embeddings in text classification, demonstrating their potential to enhance model performance by leveraging deep, contextualized representations of language. The comparative analysis of model performances, facilitated by the adaptation to **Word2Vec** features, contributes to a nuanced understanding of the intersection between traditional machine learning techniques and the advanced semantic capabilities afforded by deep learning embeddings. The entire chunk of the code is represented below.

```
def document_to_word2vec(doc, model):
    # Tokenize the document, filter out words not in the model's
    # vocabulary
    words = [word for word in doc.split() if word in model.
        key_to_index]
    if len(words) == 0:
        return np.zeros(model.vector_size)
    # Convert words to vectors and average them
```

1.1 Entire Implementation Approach

```
    return np.mean([model[word] for word in words], axis=0)
X_train_word2vec = np.array([document_to_word2vec(doc,
    word2vec_model) for doc in X_train])
X_test_word2vec = np.array([document_to_word2vec(doc,
    word2vec_model) for doc in X_test])
# Define a simple feed-forward neural network for Keras
def create_nn_model():
    model = Sequential()
    model.add(Dense(128, input_dim=X_train_word2vec.shape[1],
        activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(64, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(len(set(y_train)), activation='softmax')) #
        Number of classes
    model.compile(loss='sparse_categorical_crossentropy',
        optimizer='adam', metrics=['accuracy'])
    return model
def train_and_evaluate_models(X_train, X_test):
    # Define the models
    models = {
        'Pre-trained Embeddings Naive Bayes': GaussianNB(),
        'Pre-trained Embeddings SVM': SVC(kernel='linear'),
        'Pre-trained Embeddings Random Forest':
            RandomForestClassifier(random_state=42),
        'Pre-trained Embeddings Decision Tree':
            DecisionTreeClassifier(random_state=42),
        'Pre-trained Embeddings Deep Neural Network':
            KerasClassifier(build_fn=create_nn_model, epochs=10,
                batch_size=32, verbose=0)
```

1 Entire Literature Review

```
}

# Train and evaluate the models
results = {}
for model_name, model in models.items():
    # Train
    model.fit(X_train, y_train)
    # Predict
    y_pred = model.predict(X_test)
    # Evaluate
    report = classification_report(y_test, y_pred, output_dict
                                  =True)
    results[model_name] = {
        'Precision': report['weighted avg']['precision'],
        'Recall': report['weighted avg']['recall'],
        'F1-Score': report['weighted avg']['f1-score'],
        'Accuracy': report['accuracy'],
        'Support': report['weighted avg']['support']
    }

# Convert results to DataFrame for display
df_results = pd.DataFrame(results).transpose()
return df_results

# Train and evaluate models with Word2Vec features
df_results_word2vec = train_and_evaluate_models(X_train_word2vec,
                                                X_test_word2vec)
print(df_results_word2vec)
```

Building upon the initial experiments with unigrams, bigrams, and Word2Vec embeddings, the research further explores the optimization of text classification through feature selection. This phase introduces the use of the chi-squared (χ^2) test to identify and retain the most significant features, thereby refining the dataset for enhanced model

1.1 Entire Implementation Approach

performance. The chi-squared test, a statistical method to examine the independence of two events, is employed here to discern the relevance of each feature (word or bigram) to the classification task at hand.

After the application of TF-IDF vectorization with both unigrams and bigrams to the text data, thereby expanding the feature space to capture more complex linguistic patterns, the dimensionality of the feature set is substantially increased. To manage this complexity and focus on the most informative aspects of the data, the chi-squared test is applied, selecting the top 10,000 features deemed most relevant to the classification outcomes. This feature selection process not only aims to improve model accuracy by concentrating on significant features but also seeks to enhance computational efficiency by reducing the feature space.

```
k_best = 10000 ch2 = SelectKBest(chi2, k=k_best)
X_train_chi2_selected = ch2.fit_transform(X_train_tfidf_ngrams,
y_train) X_test_chi2_selected = ch2.transform(
X_test_tfidf_ngrams)
```

With the optimized feature set obtained through chi-squared selection, the study progresses to model training and evaluation, maintaining a consistent approach with previous phases. A simple feed-forward neural network is adapted to accommodate the revised input dimensionality, ensuring the model is tailored to the characteristics of the chi-squared selected features. The evaluation framework extends to include a variety of machine learning models, each prefixed with 'Chi-Squared Test' to signify their adaptation to the refined feature set. This suite of models encompasses Naive Bayes, SVM, Random Forest, Decision Tree, and a Deep Neural Network, mirroring the diverse methodologies explored in earlier stages of the research.

```
k_best = 10000 ch2 = SelectKBest(chi2, k=k_best)
X_train_chi2_selected = ch2.fit_transform(X_train_tfidf_ngrams,
y_train) X_test_chi2_selected = ch2.transform(
```

1 Entire Literature Review

```
X_test_tfidf_ngrams)
```

With the optimized feature set obtained through chi-squared selection, the study progresses to model training and evaluation, maintaining a consistent approach with previous phases. A simple feed-forward neural network is adapted to accommodate the revised input dimensionality, ensuring the model is tailored to the characteristics of the chi-squared selected features. The evaluation framework extends to include a variety of machine learning models, each prefixed with 'Chi-Squared Test' to signify their adaptation to the refined feature set. This suite of models encompasses Naive Bayes, SVM, Random Forest, Decision Tree, and a Deep Neural Network, mirroring the diverse methodologies explored in earlier stages of the research.

```
df_results_chi2 = train_and_evaluate_models(X_train_chi2_selected ,  
      X_test_chi2_selected) print(df_results_chi2)
```

The comprehensive assessment of these models, facilitated by the chi-squared feature selection technique, provides a deep dive into the impact of focused feature selection on text classification performance. By comparing the results against those obtained with the full feature set, the study elucidates the balance between feature richness and model efficiency, offering insights into optimal strategies for text classification in varying computational and data contexts. This phase of the research underscores the significance of targeted feature selection, particularly in high-dimensional data scenarios, as a means to bolster model performance while managing computational resources effectively. The entire chunk of the code is represented below.

```
# Using bigrams  
tfidf_vectorizer_ngrams = TfidfVectorizer(stop_words='english',  
      ngram_range=(1,2))  
X_train_tfidf_ngrams = tfidf_vectorizer_ngrams.fit_transform(  
      X_train).toarray() # Convert to array for Keras
```

1.1 Entire Implementation Approach

```
X_test_tfidf_ngrams = tfidf_vectorizer_ngrams.transform(X_test).
    toarray()
# Select top 10,000 features based on the chi-squared test
k_best = 10000
ch2 = SelectKBest(chi2, k=k_best)
X_train_chi2_selected = ch2.fit_transform(X_train_tfidf_ngrams,
    y_train)
X_test_chi2_selected = ch2.transform(X_test_tfidf_ngrams)
# Define a simple feed-forward neural network for Keras
def create_nn_model():
    model = Sequential()
    model.add(Dense(128, input_dim=X_train_chi2_selected.shape[1],
        activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(64, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(len(set(y_train)), activation='softmax')) #
        Number of classes
    model.compile(loss='sparse_categorical_crossentropy',
        optimizer='adam', metrics=['accuracy'])
    return model
def train_and_evaluate_models(X_train, X_test):
    # Define the models
    models = {
        'Chi-Squared Test Naive Bayes': MultinomialNB(),
        'Chi-Squared Test SVM': SVC(kernel='linear'),
        'Chi-Squared Test Random Forest': RandomForestClassifier(
            random_state=42),
        'Chi-Squared Test Decision Tree': DecisionTreeClassifier(
            random_state=42),
```

1 Entire Literature Review

```
'Chi-Squared Test Deep Neural Network': KerasClassifier(
    build_fn=create_nn_model, epochs=10, batch_size=32,
    verbose=0)
}

# Train and evaluate the models
results = {}
for model_name, model in models.items():
    # Train
    model.fit(X_train, y_train)
    # Predict
    y_pred = model.predict(X_test)
    # Evaluate
    report = classification_report(y_test, y_pred, output_dict
                                  =True)
    results[model_name] = {
        'Precision': report['weighted avg']['precision'],
        'Recall': report['weighted avg']['recall'],
        'F1-Score': report['weighted avg']['f1-score'],
        'Accuracy': report['accuracy'],
        'Support': report['weighted avg']['support']
    }

# Convert results to DataFrame for display
df_results = pd.DataFrame(results).transpose()
return df_results

# Train and evaluate models with chi-squared selected features
df_results_chi2 = train_and_evaluate_models(X_train_chi2_selected,
                                             X_test_chi2_selected)
print(df_results_chi2)
```

In the subsequent phase of the study, the focus shifts toward the implementation

of ensemble techniques, particularly the Bagging (Bootstrap Aggregating) method, to further refine the predictive performance of the models. This approach is predicated on the principle of combining the predictions from multiple models to reduce variance, enhance stability, and improve accuracy over single models. The Bagging strategy is particularly notable for its effectiveness in mitigating overfitting, a common challenge in complex classification tasks.

The Bagging method involves training a series of base estimators on random subsets of the original dataset, with replacement, and then aggregating their individual predictions to form a final verdict. This technique leverages diversity among the base models to achieve a more robust and reliable classification performance.

For the practical application of this technique, a modified neural network model is defined to dynamically adjust to varying input dimensions and number of classes, ensuring flexibility across different datasets. This model retains the architecture of previously utilized neural networks, featuring dense layers with dropout regularization to prevent overfitting, and culminates in a softmax layer tailored to the classification task at hand.

```
def create_nn_model(input_dim, num_classes):
    model = Sequential()
    model.add(Dense(128, input_dim=input_dim, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(64, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(num_classes, activation='softmax'))
    model.compile(loss='sparse_categorical_crossentropy',
                  optimizer='adam', metrics=['accuracy'])
    return model
```

The ensemble's training and evaluation are conducted using the `BaggingClassifier` from `Scikit-learn`, with a configuration that allows for the integration of both traditional

1 Entire Literature Review

machine learning models and a Keras-based deep neural network. This ensemble model is instantiated with a specified base estimator and the number of models to aggregate, set to 10 in this case, to form the Bagging ensemble. The models considered include SVM, Random Forest, Decision Tree, Naive Bayes, and a Deep Neural Network, each trained and evaluated on the chi-squared feature-selected dataset.

```
results_bagging = {
    'Bagging SVM': train_bagging_classifier(SVC(kernel='linear'),
        X_train_chi2_selected, X_test_chi2_selected, y_train,
        y_test),
    'Bagging Random Forest': train_bagging_classifier(
        RandomForestClassifier(random_state=42),
        X_train_chi2_selected, X_test_chi2_selected, y_train,
        y_test),
    'Bagging Decision Tree': train_bagging_classifier(
        DecisionTreeClassifier(random_state=42),
        X_train_chi2_selected, X_test_chi2_selected, y_train,
        y_test),
    'Bagging Naive Bayes': train_bagging_classifier(MultinomialNB
        (), X_train_chi2_selected, X_test_chi2_selected, y_train,
        y_test),
    'Bagging Deep Neural Network': train_bagging_classifier(
        KerasClassifier(build_fn=lambda: create_nn_model(
            X_train_chi2_selected.shape[1], len(set(y_train))), epochs
            =10, batch_size=32, verbose=0), X_train_chi2_selected,
            X_test_chi2_selected, y_train, y_test)
}
```

The aggregation of models through Bagging is meticulously evaluated, with performance metrics such as Precision, Recall, F1-Score, Accuracy, and Support being calculated for each ensemble. This evaluation not only highlights the efficacy of Bagging in improving

model performance but also offers comparative insights into how different base models contribute to the ensemble's overall accuracy and reliability.

```
df_results_bagging = pd.DataFrame(results_bagging).transpose()
print(df_results_bagging)
```

This exploration of Bagging ensembles represents a crucial advancement in the study, illustrating the potential of ensemble methods to elevate the performance of text classification models. By harnessing the collective power of multiple classifiers, the research demonstrates the ability to achieve superior predictive accuracy, showcasing the value of ensemble techniques in complex machine learning tasks. The findings from this phase underscore the importance of diversity among base models and the strategic application of ensemble methods as a means to navigate the challenges inherent in text classification. The entire chunk of the code can be seen below as followed:

```
# Model Definitions
def create_nn_model(input_dim, num_classes):
    model = Sequential()
    model.add(Dense(128, input_dim=input_dim, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(64, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(num_classes, activation='softmax'))
    model.compile(loss='sparse_categorical_crossentropy',
                  optimizer='adam', metrics=['accuracy'])
    return model

# Model Training and Evaluation
def train_bagging_classifier(base_estimator, X_train, X_test,
                             y_train, y_test):
    bagging_classifier = BaggingClassifier(base_estimator=
        base_estimator, n_estimators=10, random_state=42)
```

1 Entire Literature Review

```
bagging_classifier.fit(X_train, y_train)
y_pred = bagging_classifier.predict(X_test)
report = classification_report(y_test, y_pred, output_dict=
    True)
return {
    'Precision': report['weighted avg']['precision'],
    'Recall': report['weighted avg']['recall'],
    'F1-Score': report['weighted avg']['f1-score'],
    'Accuracy': report['accuracy'],
    'Support': report['weighted avg']['support']
}

# Main Execution
results_bagging = {
    'Bagging SVM': train_bagging_classifier(SVC(kernel='linear'),
        X_train_chi2_selected, X_test_chi2_selected, y_train,
        y_test),
    'Bagging Random Forest': train_bagging_classifier(
        RandomForestClassifier(random_state=42),
        X_train_chi2_selected, X_test_chi2_selected, y_train,
        y_test),
    'Bagging Decision Tree': train_bagging_classifier(
        DecisionTreeClassifier(random_state=42),
        X_train_chi2_selected, X_test_chi2_selected, y_train,
        y_test),
    'Bagging Naive Bayes': train_bagging_classifier(MultinomialNB
        (), X_train_chi2_selected, X_test_chi2_selected, y_train,
        y_test),
    'Bagging Deep Neural Network': train_bagging_classifier(
        KerasClassifier(build_fn=lambda: create_nn_model(
            X_train_chi2_selected.shape[1], len(set(y_train))), epochs
```

1.1 Entire Implementation Approach

```
=10, batch_size=32, verbose=0), X_train_chi2_selected,
X_test_chi2_selected, y_train, y_test)
}
# Convert results to DataFrame for display
df_results_bagging = pd.DataFrame(results_bagging).transpose()
print(df_results_bagging)
```

Continuing the exploration of machine learning techniques for text classification, the study introduces a nuanced approach focusing on regularization strategies to mitigate overfitting and improve model generalization. This phase examines the impact of incorporating dropout in deep neural networks and regularization in traditional machine learning models on classification performance.

The methodology begins with the definition of a neural network model, which is structurally designed to include dropout layers. These layers act as a form of regularization, randomly disabling a fraction of neurons during the training phase to prevent the model from becoming too reliant on any single feature, thereby enhancing its ability to generalize to unseen data. The network comprises an initial dense layer with 128 neurons, followed by a dropout layer, a second dense layer with 64 neurons and another dropout layer, culminating in an output layer tailored to the number of classification categories.

```
def create_nn_model(input_dim, num_classes):
    model = Sequential()
    model.add(Dense(128, input_dim=input_dim, activation='relu'))
    model.add(Dropout(0.5)) # Dropout layer for regularization
    model.add(Dense(64, activation='relu'))
    model.add(Dropout(0.5)) # Dropout layer for regularization
    model.add(Dense(num_classes, activation='softmax'))
    model.compile(loss='sparse_categorical_crossentropy',
                  optimizer='adam', metrics=['accuracy'])
    return model
```

1 Entire Literature Review

The study extends the concept of regularization beyond neural networks to traditional machine learning models by adjusting their parameters. For instance, SVMs are evaluated with both L1 and L2 regularization, Naive Bayes with a tweaked smoothing parameter, and Random Forest with constraints on the depth and number of features to consider at each split. These adjustments aim to examine how different forms of regularization affect model performance across various classifiers.

The main execution of this approach involves training and evaluating each classifier, including SVMs with different regularization norms, a regularized version of Naive Bayes, a Random Forest with depth and feature limitations, and the regularized deep neural network. The performance of each model is assessed using metrics such as precision, recall, F1-score, and accuracy, providing a comprehensive view of how regularization influences the effectiveness of text classification models.

```
results_regularized = {'Dropout SVM L1':
    train_and_evaluate_classifier(LinearSVC(penalty='l1', dual
    =False, C=1.0), X_train_chi2_selected, X_test_chi2_selected,
    y_train, y_test),
    'Dropout SVM L2': train_and_evaluate_classifier(SVC(kernel='
    linear', C=1.0), X_train_chi2_selected,
    X_test_chi2_selected, y_train, y_test),
    'Dropout Naive Bayes Regularized':
    train_and_evaluate_classifier(MultinomialNB(alpha=0.5),
    X_train_chi2_selected, X_test_chi2_selected, y_train,
    y_test),
    'Dropout Random Forest Regularized':
    train_and_evaluate_classifier(RandomForestClassifier(
    n_estimators=100, max_depth=10, max_features='sqrt',
    random_state=42), X_train_chi2_selected,
    X_test_chi2_selected, y_train, y_test),
```

```

'Dropout Decision Tree': train_and_evaluate_classifier(
    DecisionTreeClassifier(random_state=42),
    X_train_chi2_selected, X_test_chi2_selected, y_train,
    y_test),
'Dropout Deep Neural Network': train_and_evaluate_classifier(
    KerasClassifier(build_fn=lambda: create_nn_model(
        X_train_chi2_selected.shape[1], len(set(y_train)))), epochs
    =10, batch_size=32, verbose=0), X_train_chi2_selected,
    X_test_chi2_selected, y_train, y_test)
}

df_results_regularized = pd.DataFrame(results_regularized).
    transpose() print(df_results_regularized)

```

This segment of the research highlights the importance of regularization techniques in enhancing the robustness and generalizability of machine learning models. By comparing the performance of models with and without regularization strategies, the study sheds light on the balance between model complexity and its ability to perform accurately on new, unseen data. This exploration not only contributes to the theoretical understanding of regularization's impact on model performance but also provides practical insights for applying these techniques in text classification tasks. The entire chunk of the code shown below:

```

# Model Definitions
def create_nn_model(input_dim, num_classes):
    model = Sequential()
    model.add(Dense(128, input_dim=input_dim, activation='relu'))
    model.add(Dropout(0.5)) # Dropout layer for regularization
    model.add(Dense(64, activation='relu'))
    model.add(Dropout(0.5)) # Dropout layer for regularization
    model.add(Dense(num_classes, activation='softmax'))

```

1 Entire Literature Review

```
model.compile(loss='sparse_categorical_crossentropy',
              optimizer='adam', metrics=['accuracy'])

return model

# Model Training and Evaluation
def train_and_evaluate_classifier(classifier, X_train, X_test,
                                y_train, y_test):
    classifier.fit(X_train, y_train)
    y_pred = classifier.predict(X_test)
    report = classification_report(y_test, y_pred, output_dict=
        True)
    return {
        'Precision': report['weighted avg']['precision'],
        'Recall': report['weighted avg']['recall'],
        'F1-Score': report['weighted avg']['f1-score'],
        'Accuracy': report['accuracy'],
        'Support': report['weighted avg']['support']
    }

# Main Execution
results_regularized = {
    'Dropout SVM L1': train_and_evaluate_classifier(LinearSVC(
        penalty='l1', dual=False, C=1.0), X_train_chi2_selected,
        X_test_chi2_selected, y_train, y_test),
    'Dropout SVM L2': train_and_evaluate_classifier(SVC(kernel='
        linear', C=1.0), X_train_chi2_selected,
        X_test_chi2_selected, y_train, y_test),
    'Dropout Naive Bayes Regularized':
        train_and_evaluate_classifier(MultinomialNB(alpha=0.5),
        X_train_chi2_selected, X_test_chi2_selected, y_train,
        y_test),
    'Dropout Random Forest Regularized':
```



```

train_and_evaluate_classifier(RandomForestClassifier(
    n_estimators=100, max_depth=10, max_features='sqrt',
    random_state=42), X_train_chi2_selected,
    X_test_chi2_selected, y_train, y_test),
'Dropout Decision Tree': train_and_evaluate_classifier(
    DecisionTreeClassifier(random_state=42),
    X_train_chi2_selected, X_test_chi2_selected, y_train,
    y_test),
'Dropout Deep Neural Network': train_and_evaluate_classifier(
    KerasClassifier(build_fn=lambda: create_nn_model(
        X_train_chi2_selected.shape[1], len(set(y_train)))), epochs
    =10, batch_size=32, verbose=0), X_train_chi2_selected,
    X_test_chi2_selected, y_train, y_test)
}

# Convert results to DataFrame for display
df_results_regularized = pd.DataFrame(results_regularized).
    transpose()
print(df_results_regularized)

```

1.1.3 Models' Evaluation

In the evaluation phase of the study, an exhaustive evaluation process is undertaken to synthesize and analyze the results obtained from the various machine learning models explored across different text classification strategies. This comprehensive analysis is designed to distill insights into the relative performance of each model and technique, ultimately identifying the most effective approach for the dataset in question.

```

all_results = pd.concat([
    df_results, # Basic TF-IDF
    df_results_ngrams, # TF-IDF with n-grams

```

1 Entire Literature Review

```
df_results_word2vec, # Word2Vec embeddings
df_results_chi2, # Chi-squared feature selection
df_results_bagging, # Bagging classifiers
df_results_regularized, # Regularized models
])
```

The culmination of the research involves consolidating the results from all experiments, including those conducted with basic TF-IDF vectorization, n-grams, Word2Vec embeddings, chi-squared feature selection, bagging classifiers, and models employing different regularization strategies. This consolidation is achieved by aggregating the performance metrics—precision, recall, F1-score, accuracy, and support—of each model across the various experiments into a single DataFrame.

```
sorted_results = all_results.sort_values(by='F1-Score', ascending=False)
```

Following the aggregation, the consolidated results are sorted based on the F1-score, a balanced metric that considers both precision and recall, providing a holistic view of each model's performance. This sorting facilitates an at-a-glance comparison of the effectiveness of different text classification approaches, highlighting the models that best navigate the trade-off between identifying relevant instances and maintaining accuracy across varied contexts.

The sorted results are then presented, showcasing the performance of each model and approach in a descending order of F1-scores. This visualization not only allows for the identification of the top-performing models but also offers insights into how different preprocessing, feature selection, and regularization techniques can impact model efficacy.

In a decisive step, the study identifies the best model based on the highest F1-score, underscoring the model that achieved the optimal balance between precision and recall within the dataset. This model is highlighted not only for its superior performance but also as a benchmark for the effectiveness of the machine learning techniques applied

throughout the research.

```
best_model = sorted_results.iloc[0]
print("\nBest Model based on F1-Score:")
print(best_model.name)
print("F1-Score:", best_model['F1-Score'])
```

This evaluative process serves as the cornerstone of the research, offering a nuanced understanding of how various machine learning models and techniques perform in the realm of text classification. By meticulously comparing and contrasting the results, the study provides valuable insights into the most efficacious strategies for text classification, guiding future efforts in the field and offering a roadmap for applying these findings to similar datasets and challenges. The entire chunk of the evaluation code is shown below:

```
# Consolidate all results
all_results = pd.concat([
    df_results,
    df_results_ngrams,
    df_results_word2vec,
    df_results_chi2,
    df_results_bagging,
    df_results_regularized,
])

# Sort based on F1-Score
sorted_results = all_results.sort_values(by='F1-Score', ascending=
    False)

# Display the consolidated results
print(sorted_results)

# Display the best model
best_model = sorted_results.iloc[0]
print("\nBest Model based on F1-Score:")
print(best_model.name)
```

1 Entire Literature Review

```
print("F1-Score:", best_model['F1-Score'])
```

In a further step to distill insights from the comprehensive evaluation of text classification models, the study applies a filtering technique based on the model feature. This method simplifies the analysis by categorizing the accumulated results according to the distinct models utilized throughout the research. By parsing the first unique word from the index of the consolidated results DataFrame, which represents the model name, a new 'Model' column is introduced. This column facilitates a structured way to filter and examine the performance metrics for each model individually. The approach ensures that the analysis can focus on the nuanced differences and comparative effectiveness of each model type across various configurations and preprocessing strategies.

```
all_results['Model'] = all_results.index.to_series().apply(lambda  
    x: x.split()[0])
```

With the 'Model' column established, the dataset is filtered to isolate the results for each unique model name. This segmentation allows for a focused analysis on how each model type—be it Naive Bayes, SVM, Random Forest, Decision Tree, or Deep Neural Network—performs under different experimental conditions. The filtering process underscores the adaptability and strengths of each model, offering a lens through which the efficacy of various machine learning approaches to text classification can be assessed in isolation.

For each model identified, the corresponding subset of results is displayed, excluding the now-redundant 'Model' column for clarity. This step ensures that the analysis remains concise and directly relevant to the research questions at hand, highlighting the performance metrics—precision, recall, F1-score, accuracy, and support—pertinent to each model's ability to classify text effectively. This nuanced approach to analyzing the results not only enhances the clarity of the study's findings but also provides a granular understanding of the comparative advantages and limitations of different machine learning

models in the context of text classification. By systematically examining the results for each unique model, the study offers targeted insights that can inform future research and application of machine learning techniques in similar domains.

In the concluding analysis of the study, a statistical examination was conducted to determine if the enhancement techniques applied to the machine learning models resulted in significant differences in performance, specifically measured by the F1-score. This step is critical in understanding the effectiveness of the enhancements and whether they offer a statistically significant improvement over the baseline or other techniques.

The analysis utilized the Analysis of Variance (ANOVA) test, a method used to compare the means of three or more samples to understand if at least one of the sample means differs significantly from the others. This approach is particularly useful in this context, as it allows for the assessment of differences across multiple groups—here, the various models and their enhancement techniques—simultaneously.

```
anova_results = stats.f_oneway(*(all_results[all_results['Model']  
    == model]['F1-Score'] for model in unique_models))
```

The ANOVA test produces two key metrics: the F-statistic and the P-value. The F-statistic indicates the ratio of the variance between the groups to the variance within the groups, providing a measure of the overall variance among the mean F1-scores of the different models. The P-value, on the other hand, offers insight into the statistical significance of the observed variance. Upon reviewing the ANOVA results, the study interprets these metrics to determine the statistical relevance of the enhancements applied to the models. A P-value less than 0.05 would suggest that there is a significant difference in the F1-scores among the models, implying that at least one of the enhancement techniques has a statistically significant impact on model performance.

```
if anova_results.pvalue < 0.05:  
    print("At least one model's F1 score is significantly  
        different.")
```

1 Entire Literature Review

```
else:
    print("No significant difference found between the enhancement
          techniques' F1 scores.")
```

This statistical analysis is essential for validating the effectiveness of the applied enhancements, providing a rigorous basis for the conclusions drawn from the experimental results. By identifying whether the differences in performance metrics are statistically significant, the study ensures that the recommendations and insights derived from the research are grounded in robust statistical evidence, offering a conclusive assessment of the value added by the enhancement techniques.

1.1.4 Online App Deployment

The findings from our investigation reveal that the Support Vector Machine (SVM) machine learning model, even without the application of enhancement techniques, stands out for its computational efficiency and robust average performance metrics. In light of these results, we opted for a merged dataset approach, amalgamating data from DATASET 1 and DATASET 2. This strategy not only enriches our training corpus with a diverse array of texts spanning business, entertainment, politics, sports, and technology but also closely aligns with our client's requirements for the final application deployment. Consequently, this section delineates the process for extracting and externalizing the SVM model, thereby facilitating its integration and application in the deployment of our app.

The foundational stage of our application deployment involves preparing and partitioning the dataset crucial for training our machine learning model, as outlined in the following script:

```
# Define the base path to your dataset
base_path = Path('Data_For_the_App_Training/Data') # Adjust this
to the path of your dataset
```

```
# Dynamically list categories based on directories in the dataset
folder

categories = [f.name for f in base_path.iterdir() if f.is_dir()]
texts = []
labels = []

# Iterate over categories and read files
for category in categories:
    category_path = base_path / category
    for file_path in category_path.iterdir():
        if file_path.is_file() and not file_path.name.startswith('.'):
            with open(file_path, 'r', encoding='utf-8', errors='
                ignore') as file:
                texts.append(file.read())
                labels.append(category)

# Splitting the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(texts, labels,
    test_size=0.3, random_state=42)
```

This script commences by establishing a base path to the dataset, facilitating access to the text data for model training. It then proceeds to automatically categorize the data by scanning the directory structure, streamlining the process of classifying texts into their respective categories. For each category, the script processes eligible files, excluding hidden ones, to compile the texts and their labels into lists. This meticulous collection ensures that each piece of text is associated with its correct category, forming the basis for supervised learning.

The final step involves the division of the dataset into training and testing sets using

1 Entire Literature Review

the `train_test_split` function. This crucial split, allocating 70% of the data for training and 30% for testing, lays the groundwork for model training and evaluation, ensuring the model's effectiveness in classifying new, unseen texts. Through this approach, we establish a systematic and reproducible method for dataset preparation, underpinning the subsequent phases of our application's machine learning pipeline.

After the data preprocessing, code snippet trains and evaluates a Support Vector Machine (SVM) model for text classification. It starts by converting the training and testing text data into TF-IDF features, excluding English stop words and using unigrams. An SVM model with a linear kernel is then initialized and trained on the transformed training data. Finally, the model's performance is evaluated on the test set, with the results displayed through a classification report that includes metrics such as precision, recall, and F1-score.

```
# Training the SVM model
tfidf_vectorizer = TfidfVectorizer(stop_words='english',
                                   ngram_range=(1, 1))
X_train_tfidf = tfidf_vectorizer.fit_transform(X_train)
X_test_tfidf = tfidf_vectorizer.transform(X_test)

# Initialize SVM with probability estimates
svm_model = SVC(kernel='linear', probability=True)
svm_model.fit(X_train_tfidf, y_train)

# Evaluating the model
y_pred = svm_model.predict(X_test_tfidf)
print(classification_report(y_test, y_pred))
```

This code snippet saves the trained SVM model and the TF-IDF vectorizer to disk using Python's pickle module. It serializes the objects into binary format and writes

them to files named `svm_model.pkl` and `tfidf_vectorizer.pkl`, respectively. This allows for the model and vectorizer to be easily reloaded for future predictions without needing to retrain or refit.

```
# Training the SVM model
# Serialize and save the SVM model
with open('svm_model.pkl', 'wb') as file:
    pickle.dump(svm_model, file)

# Serialize and save the TF-IDF vectorizer
with open('tfidf_vectorizer.pkl', 'wb') as file:
    pickle.dump(tfidf_vectorizer, file)
```

This code triggers the download of the serialized TF-IDF vectorizer and SVM model files (`tfidf_vectorizer.pkl` and `svm_model.pkl`) from the server to your local machine, enabling the use of the trained model and vectorizer in the online app.

```
files.download('tfidf_vectorizer.pkl')
files.download('svm_model.pkl')
```

Following the TF-IDF vectorizer and SVM model files extraction, the work goes into Visual Studio Code where the code for the app is written. The code snippet outlines the development of a web-based application utilizing Streamlit, an open-source app framework for Machine Learning and Data Science projects. This application integrates a pre-trained Support Vector Machine (SVM) model for the purpose of classifying English text into predefined categories, demonstrating an advanced application of natural language processing (NLP) techniques within the scope of a Master's thesis project.

At the onset, the application initializes by loading the serialized SVM model and the TF-IDF (Term Frequency-Inverse Document Frequency) vectorizer from their respective files which were downloaded before. These components are fundamental to the app's

1 Entire Literature Review

operation, with the SVM model serving as the core predictive mechanism and the TF-IDF vectorizer transforming raw text inputs into a format amenable to model processing.

Following the setup, the Streamlit framework is employed to construct the user interface of the web application. The interface is intuitively designed, featuring a title and a comprehensive description of the app's functionality, including instructions for users on how to input text for classification. Users are offered three distinct modes for inputting text: direct text entry, uploading a text file, or providing a URL to an article. This flexibility ensures the app's accessibility and usability across various user preferences and scenarios.

For text input via URL, the app implements an additional feature using the requests library to fetch the content of the provided webpage, which is then parsed using BeautifulSoup to extract the textual content. This functionality, however, is subject to limitations imposed by certain websites that restrict web crawling activities.

Upon receiving an input, the application processes the text through the TF-IDF vectorizer, converting it into a numerical vector that reflects the importance of each term within the context of the document corpus. This vector is then fed into the SVM model, which predicts the probability distribution across the available categories. The application subsequently displays the category with the highest probability as the predicted classification.

This Streamlit application exemplifies the practical application of SVM models in text classification tasks, highlighting the process of deploying machine learning models in an accessible and user-friendly manner. The inclusion of detailed instructions, error handling for web crawling, and direct feedback on classification results enhances the user experience, making the application a valuable tool for demonstrating the capabilities of NLP techniques within the context of academic research.

```
import streamlit as st
import pickle
```

1.1 Entire Implementation Approach

```
from sklearn.feature_extraction.text import TfidfVectorizer
import requests
from bs4 import BeautifulSoup

# to run: streamlit run app.py

# Load the serialized SVM model and TF-IDF vectorizer
with open('svm_model.pkl', 'rb') as file:
    svm_model = pickle.load(file)

with open('tfidf_vectorizer.pkl', 'rb') as file:
    tfidf_vectorizer = pickle.load(file)

# Streamlit app interface setup
st.title('Text Classification based on SVM Model')

# Description and links
st.markdown('---')
st.write("""
This app uses a Support Vector Machine (SVM) model for classifying
text from articles in English language into categories. Simply
choose between entering your text, uploading a text file, or
linking the article page in the internet with the URL* in the
box below and press 'Classify' to see the predicted category.
""")
st.write("""
The app is able to identify particular text topics: business,
entertainment, politics, sport, and technology. The app is
developed as part of the Master Thesis work of Michael Zats.
""")
```

1 Entire Literature Review

```
st.markdown("""
Should you want to see all models available or use a different
dataset for training,
please refer to this [Google Colab Notebook](https://colab.
research.google.com/drive/1d_RZR8xhiVBPSOCMJkWhvPtWBXE6nAVQ?usp
=sharing) or this [Github Repository](https://github.com/
Michaelzats/Thesis-PCU).
""")

st.markdown("""
Linking URL Pages option*: sometimes this option may not work as
some particular websites do not let the crawler to go through.
""")
st.markdown('---')

# Choose input mode
mode = st.radio("Choose input mode:", ("Enter text", "Upload text
file", "Enter URL"))

if mode == "Enter text":
    user_input = st.text_area("Enter text here:", "")
elif mode == "Upload text file":
    uploaded_file = st.file_uploader("Choose a text file", type=['
txt'])
    if uploaded_file is not None:
        user_input = str(uploaded_file.read(), "utf-8") # Convert
to string
        st.text_area("File Contents:", user_input, height=300)
elif mode == "Enter URL":
    user_input = st.text_input("Enter URL here:", "")
```

```
if user_input:
    try:
        response = requests.get(user_input)
        soup = BeautifulSoup(response.content, 'html.parser')
        user_input = soup.get_text(separator=' ')
        st.text_area("Fetched text:", user_input, height=300)
    except Exception as e:
        st.error(f"An error occurred while fetching the
                article: {e}")

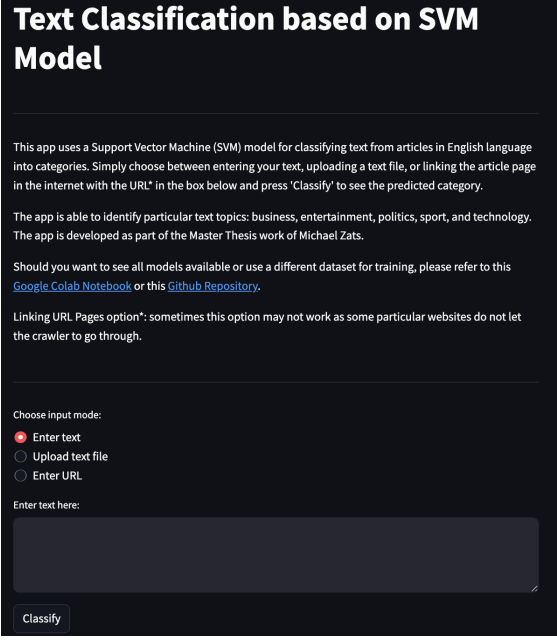
if st.button('Classify'):
    if user_input:
        # Transform and classify the input text
        transformed_input = tfidf_vectorizer.transform([user_input
                                                         ])
        probabilities = svm_model.predict_proba(transformed_input)
        max_prob_index = probabilities[0].argmax()
        prediction = svm_model.classes_[max_prob_index]

        st.subheader('Classification Result')
        st.write(f'Predicted category: **{prediction}**')
    else:
        st.warning("Please enter some text, upload a file, or
                    enter a URL to classify.")
```

Following the code creation inside of Visual Studio Code, the corresponding requirements.txt file is created where the required libraries are shown with the corresponding versions for the software.

```
beautifulsoup4==4.12.3
numpy==1.26.4
```

1 Entire Literature Review



Text Classification based on SVM Model

This app uses a Support Vector Machine (SVM) model for classifying text from articles in English language into categories. Simply choose between entering your text, uploading a text file, or linking the article page in the internet with the URL* in the box below and press 'Classify' to see the predicted category.

The app is able to identify particular text topics: business, entertainment, politics, sport, and technology. The app is developed as part of the Master Thesis work of Michael Zats.

Should you want to see all models available or use a different dataset for training, please refer to this [Google Colab Notebook](#) or this [Github Repository](#).

Linking URL Pages option*: sometimes this option may not work as some particular websites do not let the crawler to go through.

Choose input mode:

- ☒ Enter text
- ☐ Upload text file
- ☐ Enter URL

Enter text here:

Classify

Figure 1.2: Deployed Online Classification App Based on SVM

```
pandas==2.2.1
requests==2.31.0
scikit-learn==1.4.1.post1
streamlit==1.32.2
```

Moving further, so to deploy an app online, the files of **app.py** of the before-mentioned code, **requirements.txt**, **svm_model.pkl** and **tfidf_vectorizer.pkl** deployed in GitHub inside of **Thesis-PCU** to enable streamlit to seek the connection between the files and the server. Streamlit account was created in advance and connected with the GitHub repository of Thesis-PCU. As a result, the software is deployed online in a way of the app and is able to classify text topics of business, entertainment, politics, sport, and technology.