1. Importing Necessary Dependencies

```
!pip install scikit-learn-intelex
```

- **scikit-learn-intelex** is installed, which is an extension to scikit-learn to accelerate machine learning algorithms using Intel's hardware.

Python code

```
import os
import re
import zipfile
import warnings
```

- **os**: Provides a way of using operating system-dependent functionality like file operations.
- **re**: Provides regular expression matching operations to process and filter text.
- **zipfile**: Module to work with ZIP archives.
- **warnings**: Module to manage warnings and prevent displaying them during execution.

---

2. Data Science and Machine Learning Libraries

Python code

```
import dpnp as np
import modin.pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from wordcloud import WordCloud
```

- **dpnp**: Data-parallel version of NumPy, optimized for Intel architectures.
- **modin.pandas**: A parallelized version of pandas for faster data manipulation.
- **seaborn**: A library for creating visually appealing statistical plots.
- **matplotlib.pyplot**: Used for plotting graphs and visualizations.
- **WordCloud**: For generating word clouds from text data.

---

3. Scikit-Learn Patching for Intel Hardware Optimization

Python code

```
from sklearnex import patch_sklearn
patch_sklearn()
```

- **patch_sklearn**: Applies Intel's optimizations to scikit-learn, making the machine learning algorithms run faster on Intel hardware.

---

4. Text Processing Libraries

Python code

```
import string
import nltk
from nltk.stem import PorterStemmer
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
```

- **nltk**: Natural Language Toolkit used for text processing.
- **PorterStemmer**: A stemming algorithm used to reduce words to their root form (e.g., "running" to "run").
- **stopwords**: Common words that are removed to avoid noise in text processing (e.g., "the", "and").
- **word_tokenize**: Splits a string into individual words or tokens.

---

5. NLTK Downloads

Python code

```
nltk.download('punkt')
nltk.download('stopwords')
```

- **'punkt'**: Tokenizer model for splitting text into words or sentences.
- **'stopwords'**: Downloads the English stopwords dataset.

---

6. TensorFlow, DeBERTa, and Transformers

Python code

```
import intel_tensorflow as tf
```

```
from transformers import TFAutoModelForSequenceClassification,
AutoTokenizer
```

- **intel_tensorflow**: Intel-optimized version of TensorFlow for improved performance on Intel hardware.
- **transformers**: Hugging Face library used for NLP tasks like sequence classification with pre-trained models.
- **TFAutoModelForSequenceClassification**: TensorFlow model for sequence classification tasks.
- **AutoTokenizer**: Automatically loads the appropriate tokenizer for a given model.

---

7. Loading Dataset

Python code

```
data =
pd.read_csv('/kaggle/input/pheme-dataset-for-rumour-detection/dataset.
csv')
```

- **data**: Loads the dataset using pandas' `read_csv` method from the specified file path.

---

8. Data Exploration

Python code

```
data.describe().style.background_gradient(cmap='Blues').set_properties
(**{'font-family':'Segoe UI'})
```

- **describe()**: Provides summary statistics of the dataset.
- **background_gradient()**: Adds color to the summary statistics table for better visualization.

Python code

```
data.head()
```

- Displays the first five rows of the dataset.

---

9. Word Count Calculation

Python code

```
data['num_words'] = data['text'].apply(lambda x: len(x.split()))
avg_words = round(data['num_words'].mean())
max_words = round(data['num_words'].max())
```

- **apply()**: Applies a lambda function to each row of the text column.
- **x.split()**: Splits the text into words and calculates the number of words in each row.
- **mean()**: Calculates the average number of words.
- **max()**: Finds the longest entry (in terms of words) in the dataset.

---

10. Duplicates and Missing Data

Python code

```
df = data.drop_duplicates()
df.drop(df[df.isnull().any(axis=1)].index, inplace=True)
```

- **drop_duplicates()**: Removes duplicate rows.
- **isnull()**: Identifies rows with missing data, and **drop()** removes them.

---

11. Data Visualization

Python code

```
sns.countplot(data=df, x='is_rumor', palette='viridis')
```

- **countplot()**: Visualizes the distribution of the target variable `is_rumor`.

Python code

```
plt.figure(figsize=(6, 6))
df['topic'].value_counts().plot.pie(autopct='%1.1f%%')
```

- **value_counts()**: Counts the occurrences of each category.
- **pie()**: Creates a pie chart representing the distribution of topics.

---

12. Text Preprocessing

- **Lowercase and Remove Special Characters**

Python code
```
df['text'] = df['text'].str.lower().str.replace('[^\w\s]', '',
regex=True)
```

- **str.lower()**: Converts text to lowercase.
- **str.replace()**: Removes special characters using a regular expression.
- **Tokenization and Stop Words Removal**

Python code
```
df['text'] = df['text'].apply(data_processing)
```

- **data_processing()**: Custom function that tokenizes, removes stop words, and performs text preprocessing.
- **Stemming**

Python code
```
stemmer = PorterStemmer()
df['text'] = df['text'].apply(stemming)
```

- **stemming()**: Function that applies stemming to each word in the text.

---

13. Count Vectorizer for Word Frequency

Python code
```
cv = CountVectorizer(stop_words='english')
words = cv.fit_transform(df[df['is_rumor'] == 1]['text'])
```

- **CountVectorizer**: Converts text to a matrix of token counts.
- **fit_transform()**: Learns vocabulary from the text data and transforms it into count vectors.

---

14. Word Cloud Visualization

Python code
```
wordcloud = WordCloud(width=2000,
height=2000).generate_from_frequencies(dict(words_freq))
```

- **WordCloud**: Generates a word cloud from word frequencies.

---

15. Initialize DeBERTa Model and Tokenizer

Python code

```
model_name = "microsoft/deberta-v3-base"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model =
TFAutoModelForSequenceClassification.from_pretrained(model_name,
num_labels=2)
```

- **from_pretrained()**: Loads a pre-trained model (DeBERTa v3) for sequence classification with two labels (rumor and non-rumor).

---

16. Tokenizing the Dataset

Python code

```
train_df, test_df = train_test_split(df, test_size=0.2,
random_state=42)
```

- **train_test_split()**: Splits the dataset into training and testing sets (80% train, 20% test).

Python code

```
train_encodings = tokenize_data(train_df, tokenizer, max_length)
```

- **tokenize_data()**: Function to tokenize text into token IDs and attention masks.

---

17. Create TensorFlow Dataset

Python code

```
train_dataset = create_tf_dataset(train_encodings,
train_df['is_rumor']).batch(batch_size)
```

- **create_tf_dataset()**: Function that converts token encodings and labels into a TensorFlow dataset.

---

18. Model Training

Python code
```
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-5),

loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

- **compile()**: Compiles the model with an Adam optimizer, sparse categorical cross-entropy loss, and accuracy metrics.

Python code
```
history = model.fit(train_dataset, validation_data=test_dataset,
epochs=15)
```

- **fit()**: Trains the model on the training dataset for 15 epochs.

---

19. Plot Training and Validation Loss/Accuracy

Python code
```
plt.plot(epochs, train_loss, 'b-o', label='Training loss')
```

- **plt.plot()**: Plots training loss and accuracy over epochs.

---

20. Model Evaluation

Python code
```
test_loss, test_accuracy = loaded_model.evaluate(test_dataset)
```

- **evaluate()**: Evaluates the model's performance on the test dataset.

---

21. Confusion Matrix and Classification Report

Python code
```
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=labels, yticklabels=labels)
```

- **heatmap()**: Visualizes the confusion matrix to compare true and predicted labels.

---

22. Zipping the Model and Tokenizer

Python code
```
with zipfile.ZipFile(zip_file_name, 'w', zipfile.ZIP_DEFLATED) as zipf:
    for root, _, files in os.walk(model_save_path):
        for file in files:
            zipf.write(os.path.join(root, file))
```

- **ZipFile**: Compresses the saved model and tokenizer into a ZIP archive for easier sharing and storage.

Conclusion

The provided code is a comprehensive implementation of a rumor detection system leveraging natural language processing (NLP) and machine learning techniques. By using state-of-the-art models like DeBERTa, optimized with Intel's hardware acceleration, the system effectively classifies social media texts as rumors or non-rumors.

Key Highlights:

1. **Data Preparation**: The code begins with data loading and preprocessing, ensuring that text is cleaned, tokenized, and transformed into a suitable format for machine learning algorithms. The inclusion of techniques like stemming and stopword removal enhances the quality of the input data.
2. **Exploratory Data Analysis (EDA)**: Visualization techniques are employed to understand the distribution of the dataset and the frequency of different topics, which provides insights into the data's characteristics.
3. **Model Training**: The use of DeBERTa for sequence classification is particularly noteworthy. The training process is optimized for performance, leveraging Intel's TensorFlow implementation for faster computation. The model is trained on labeled data, allowing it to learn the nuanced differences between rumors and non-rumors.

4. **Evaluation Metrics**: The system includes rigorous evaluation steps, such as calculating accuracy, generating confusion matrices, and plotting training history, ensuring that the model's performance is well understood.
5. **Deployment Readiness**: Finally, the model and tokenizer are packaged into a ZIP file, making it easy to store or share for further deployment in applications.

In summary, this code exemplifies a well-structured approach to solving a real-world problem through NLP and machine learning. The thorough use of libraries, efficient preprocessing steps, and state-of-the-art model training reflect best practices in the field, paving the way for future enhancements, such as fine-tuning the model with additional data or integrating the system into user-facing applications.