arXiv Paper Categorization Report

This report outlines the Jupyter Notebook, which focuses on building a machine learning model to categorize scientific papers from the arXiv dataset based on their titles and abstracts. The notebook demonstrates data acquisition from Kaggle, text preprocessing, model training using a Support Vector Classifier, and a simple prediction mechanism.

1. Project Objective

The primary objective of this project is to develop a machine learning model that can accurately classify arXiv papers into their respective scientific categories. This involves processing large volumes of text data (titles and abstracts) and training a classifier to infer the category of a given paper.

2. Setup and Data Acquisition

The notebook begins by setting up the Kaggle API and downloading the arXiv dataset.

- **Kaggle Setup**: The Kaggle library is installed, and environment variables for Kaggle username and API key are set. This allows programmatic access to Kaggle datasets.

  ```
  !pip install kaggle
  import os
  os.environ["KAGGLE_USERNAME"] = "your_kaggle_username" # Placeholder
  os.environ["KAGGLE_KEY"] = "your_kaggle_key" # Placeholder
  ```

  (Note: The actual username and key are placeholders and should be replaced with valid credentials by the user.)

- **Dataset Download**: The arxiv dataset from Cornell University is downloaded using the Kaggle API. This dataset contains metadata for arXiv papers in JSON format.

  ```
  !kaggle datasets download Cornell-University/arxiv
  ```

- **Dataset Extraction**: The downloaded arxiv.zip file is unzipped, revealing the

arxiv-metadata-oai-snapshot.json file, which is the primary data source for this project.

```
!unzip /content/arxiv.zip
```

3. Library Imports and Data Loading

After data acquisition, necessary libraries are imported, and a subset of the data is loaded for processing.

- **NLTK Downloads**: NLTK data packages (punkt_tab, stopwords, wordnet) are downloaded, which are crucial for various text preprocessing steps.

```
import nltk
nltk.download('punkt_tab')
nltk.download('stopwords')
nltk.download('wordnet')
```

- **Core Libraries**: Essential libraries for data handling, text processing, and machine learning are imported:

```
import pandas as pd
import json
import re
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score, classification_report
from nltk.stem import PorterStemmer
from nltk.corpus import stopwords
import joblib
```

- **Data Loading**: The arxiv-metadata-oai-snapshot.json file is read line by line, and the first 10,000 entries are loaded into a pandas DataFrame. This is a common practice to handle very large JSON files that might not fit entirely into memory.

```
file_path = "/content/arxiv-metadata-oai-snapshot.json"
data = []
```

```
with open(file_path, "r") as f:
    for _ in range(10000): # Limiting to 10,000 entries for demonstration
        line = f.readline()
        if not line:
            break
        data.append(json.loads(line))
df = pd.DataFrame(data)
```

4. Text Preprocessing

A preprocess_text function is defined to clean and normalize the paper abstracts.

- **Preprocessing Steps**:
    1. **Lowercase Conversion**: Converts all text to lowercase.
    2. **Remove Digits**: Removes numerical digits from the text.
    3. **Remove Non-Alphanumeric Characters**: Keeps only word characters and spaces, removing punctuation and special symbols.
    4. **Tokenization and Stop Word Removal**: Splits the text into words and removes common English stop words.
    5. **Stemming**: Applies Porter Stemmer to reduce words to their root form (e.g., "running" to "run").
    6. **Join Tokens**: Reconstructs the cleaned tokens into a single string.

```
def preprocess_text(text):
    """Text cleaning and preprocessing"""
    text = text.lower()
    text = re.sub(r'\d+', '', text)
    text = re.sub(r'[^\w\s]', '', text)
    text = text.split()
    ps = PorterStemmer()
    stop_words = set(stopwords.words('english'))
    text = [ps.stem(word) for word in text if word not in stop_words]
    return ' '.join(text)
```

- **Feature Combination and Cleaning**: A new 'text' column is created by concatenating 'title' and 'abstract'. The preprocess_text function is then applied to the 'abstract' column (after filling any NaN values with an empty string) to create the clean_text column, which will be used for feature extraction.

```
df["text"] = df["title"] + " " + df["abstract"]
df['clean_text'] = df['abstract'].fillna("").apply(preprocess_text)
```

5. Feature Extraction

The cleaned text data is transformed into numerical features using TF-IDF.

- **TF-IDF Vectorization**: TfidfVectorizer is used to convert the clean_text into TF-IDF (Term Frequency-Inverse Document Frequency) vectors. max_features is set to 10,000 to consider the most relevant terms. The categories column is selected as the target variable y.

```
tfidf_vectorizer = TfidfVectorizer(max_features=10000)
X = tfidf_vectorizer.fit_transform(df["clean_text"])
y = df["categories"]
```

6. Model Training and Evaluation

A Support Vector Classifier (SVC) with a linear kernel is chosen for the multi-class paper categorization.

- **Train-Test Split**: The TF-IDF features (X) and target categories (y) are split into training and testing sets (80% training, 20% testing).

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

- **Model Initialization and Training**: A SVC model with a linear kernel and probability=True (to enable probability estimates) is initialized and trained on the training data.

```
svm_model = SVC(kernel='linear', probability=True)
svm_model.fit(X_train, y_train)
```

- **Prediction and Evaluation**: Predictions are made on the test set, and the model's performance is evaluated using accuracy score and a classification report.

```
y_pred = svm_model.predict(X_test)
print("Accuracy:", accuracy_score(y_test, y_pred))
print("Classification Report:\n", classification_report(y_test, y_pred))
```

The evaluation results show:

Accuracy: 0.447

The overall accuracy is approximately **44.7%**. The classification report indicates that the model performs well on highly represented categories like 'astro-ph' but struggles significantly with many other categories, especially those with very few samples in the test set (resulting in 0.00 precision, recall, and f1-score). This is typical for datasets with a large number of classes and significant class imbalance.

7. Prediction Example

The notebook includes a simple interactive example to demonstrate how the trained model can predict the category of a new paper.

- **User Input**: The user is prompted to enter a paper title and abstract.
- **Preprocessing and Prediction**: The combined input text is preprocessed, transformed into TF-IDF features, and then fed to the svm_model for prediction.

```
title = input("Enter Paper Title: ")
abstract = input("Enter Paper Abstract: ")
user_input = preprocess_text(title + " " + abstract)
X_input = tfidf_vectorizer.transform([user_input])
predicted_category = svm_model.predict(X_input)[0]
print(f"\n🔍 Predicted Paper Category: {predicted_category}")
```

For example, for the input title "Lifetime of doubly charmed baryons" and a corresponding abstract, the model predicts the category "hep-ph".

8. Conclusion and Future Work

The notebook successfully demonstrates a pipeline for categorizing arXiv papers using a Support Vector Classifier. While the overall accuracy is moderate (around 44.7%), the model shows capability in identifying well-represented categories. The low performance on many categories is likely due to the vast number of unique categories and the inherent sparsity of the dataset.

Potential areas for future improvements include:

- **Handling Multi-label Classification**: The categories column often contains multiple labels (e.g., 'astro-ph gr-qc'). The current approach treats each unique combination as a single class, which might not be optimal. Implementing a true

multi-label classification strategy (e.g., using BinaryClassifier or LabelPowerset) would be beneficial.

- **Addressing Class Imbalance**: Many categories have very few samples, leading to poor performance. Techniques like oversampling minority classes, undersampling majority classes, or using weighted loss functions could help.
- **Advanced Text Embeddings**: Exploring more sophisticated text representations beyond TF-IDF, such as Word2Vec, GloVe, FastText, or contextual embeddings like BERT, which can capture semantic relationships more effectively.
- **Deep Learning Models**: Implementing neural network architectures (e.g., CNNs, LSTMs, or transformer-based models) specifically designed for text classification, which might be better suited for the complexity of scientific abstracts.
- **Hyperparameter Tuning**: Optimizing the parameters of TfidfVectorizer (e.g., ngram_range, min_df, max_df) and SVC (e.g., C, gamma) to improve model performance.
- **Larger Dataset Utilization**: The notebook currently processes only 10,000 entries. Utilizing the full dataset (which is much larger) would likely improve model generalization, though it would require more computational resources.
- **Category Grouping/Hierarchy**: Investigating if related categories can be grouped or if a hierarchical classification approach would be more appropriate.