3 Assignment 3: Streaming Analytics on Text Data

3.1 Introduction

In this assignment, we develop a predictive pipeline using Spark Structured Streaming to process and classify live textual data streamed from arXiv.org. The goal is to automatically categorize scientific articles based on their titles and abstracts. From the outset, we chose a multiclass classification approach, assigning a single fine-grained category (e.g., cs.AI, math.AG) to each article. Importantly, we did not collapse labels into broader main categories (like cs or math), thereby maintaining the original granularity of the dataset and increasing the task complexity. The project was carried out in three main stages. First, we ingested a historical dataset by tapping into a live streaming source provided by arXiv.org. Next, we preprocessed the text data and trained a classification model capable of assigning categories to scientific articles based on their content. Finally, we deployed the model within a real-time Spark Structured Streaming environment to demonstrate live predictions on incoming data.

3.2 Collecting historical data

To retrieve a representative dataset of scholarly publications from Arxiv, we made use of the official Arxiv API, which exposes metadata in ATOM XML format. We implemented a Python script that iteratively queries the API for each day of the year 2025, using a date-based filtering approach on the `submittedDate` field:

```
from_str = start_date.strftime("%Y%m%d%H%M")
to_str = (start_date + day).strftime("%Y%m%d%H%M")
url = f"https://export.arxiv.org/api/query?search_query=submittedDate:[{fro
response = requests.get(url)
```

Each request fetches up to 2000 articles per day, ensuring we remain within the API limits while covering the entire year. For each retrieved article, we parsed key metadata fields including the identifier, title, abstract, publication date, authors, and subject categories:

```
item = {
    'id': aid_raw,
    'title': entry.find('atom:title', ns).text.strip(),
    'summary': entry.find('atom:summary', ns).text.strip(),
    'published': entry.find('atom:published', ns).text,
    'updated': entry.find('atom:updated', ns).text,
    'authors': [author.find('atom:name', ns).text for author in entry.finda
    'categories': [cat.attrib['term'] for cat in entry.findall('atom:catego
}
```

Each article was then stored as an individual JSON file, using the article's unique identifier as the filename:

```
filename = f"{aid}.json"
filepath = os.path.join(base_dir, filename)
with open(filepath, "w") as f:
    json.dump(item, f, indent=2)
```

This structure facilitates easy downstream processing, deduplication, and analysis. The resulting dataset was stored locally in a designated folder, forming a clean and structured archive of Arxiv submissions throughout the year 2025. As such, we were able to collect around 64 thousand articles, providing us with a rich dataset to train our models on.

3.3 Predictive Model

Before training the classification model, we prepared the input data by combining each article's title and abstract into a single text field. This was done to maximize the semantic information available for each prediction. We applied minimal but essential preprocessing: converting text to lowercase and trimming whitespace. For labels, we extracted the first listed category from each article's metadata, maintaining a fine-grained multiclass setup. These cleaned text-label pairs were then passed into downstream vectorization and modeling stages.
Note that, in our code, we refer to the multi-class category as 'main' category, signifying it was the first categories in the list of categories for an article. This does not, however, refer to the main category as intended by the assignment instructions. As previously detailed in the assignment introduction, we utilize multi-class with all categories (so no aggregation into a broader category).

3.3.1 TF-IDF Vectorizer

In our first iteration in building the predictive model, we applied Term Frequency-Inverse Document Frequency (TF-IDF) vectorization as a text preprocessing step to convert raw textual data into a numerical format suitable for machine learning models. TF-IDF is a statistical measure that evaluates how important a word is to a document in a collection.
Term Frequency (TF) measures how frequently a word occurs, while Inverse Document Frequency (IDF) measures how important a word is by scaling down the weights of terms that appear frequently across many documents. As such, common but less informative words (such as "the", "and", "is") receive lower weights, while more content-specific words receive higher weights. This helps enhance the model's ability to differentiate between classes based on meaningful textual features.
The method particularly helps reducing the dimensionality and sparsity of the text data. This transformation allowed us to apply traditional machine learning models more effectively, such as logistic regression and random forest.

```
tokenizer = Tokenizer(inputCol="text", outputCol="words")
stopwords = StopWordsRemover(inputCol="words", outputCol="filtered")
tf = HashingTF(inputCol="filtered", outputCol="rawFeatures",
                numFeatures=10000)
idf = IDF(inputCol="rawFeatures", outputCol="features")
```

As the code shows, transforming preprocessed text into numerical feature vectors is done with HashingTF, which computes term frequencies (TF) using a fixed size hashing function. This converts the list of filtered tokens into a sparse vector where each position corresponds to the hash of a term. The result is that each article is transformed into a 10,000-dimensional sparse vector representing term frequencies, which is then passed through the IDF step to obtain TF-IDF weights.

Logistic Regression After transforming the textual data into numerical format using TF-IDF vectorization, logistic regression is applied as a supervised learning method to classify documents.Logistic regression uses these weighted term vectors to learn a linear decision boundary between different target classes (e.g., sentiment labels or document categories). The categorical target label is encoded into a numeric label using a StringIndexer, which is necessary for training:

```
label_indexer = StringIndexer(inputCol="main_category", outputCol="label")
```

Next, we define the classifier. We use logistic regression with L2 regularization (regParam=0.01) and limit the number of iterations to 10:

```
lr = LogisticRegression(maxIter=10, regParam=0.01)
```

These into a single pipeline, allowing consistent preprocessing during both training and inference:

```
pipeline_LR = Pipeline(stages=[tokenizer, stopwords, tf, idf,
                                label_indexer, lr])
```

Before applying this pipeline, we split the data into training and test sets.; After which the model is trained on the training data, and predictions are generated on the held-out test set:

```
train_data, test_data = df.randomSplit([0.8, 0.2], seed=42)
model = pipeline_LR.fit(train_data)
predictions = model.transform(test_data)
```

To assess the quality of the logistic regression model, we use Spark's MulticlassClassificationEvaluator with accuracy as the evaluation metric. This evaluator compares the predicted labels with true labels in the test set and computes the proportion of correct predictions. This provides us with a baseline indicator of how well the model generalizes to unseen data. For our logistic regression, we achieve an accuracy of 0.5969, which is already quite a strong performance.

Random Forest We next explored whether ensemble methods could improve classification performance by implementing a random forest classifier. To do this, we implemented the same pipeline structure as with the logistic regression: tokenization, stopword removal, TF-IDF transformation, label indexing, and then fitting the model. However, due to limited computing resources on a local machine, we were constrained in the size and complexity of the model. Specifically, we reduced the TF vector dimensionality from 10,000 to 1,000 features and used a relatively small forest of only 10 trees with a maximum depth of 5. These settings were chosen to avoid memory crashes during training and prediction. This limited configuration didn't allow us to fully explore the potential of the random forest classifier, but it did provide initial results as to what results from RF might provide. Tthe accuracy, as assessed by Spark's MulticlassClassificationEvaluator, is 0.1784, which is a drastic drop from the almost 60 percent accuracy attained with logistic regression.

```
paramGrid_rf = ParamGridBuilder() \
    .addGrid(rf.numTrees, [10]) \
    .addGrid(rf.maxDepth, [5, 10]) \
    .build()

cv_rf = CrossValidator(
    estimator=pipeline_rf,
    estimatorParamMaps=paramGrid_rf,
    evaluator=evaluator,
    numFolds=2
)
```

To improve the performance of the random forest classifier, we applied cross-validation to tune hyperparameters. This was done using Spark's CrossValidator in combination with a small parameter grid, due to the limited memory and processing power available on a local machine. The tuning grid explored only a single value for the number of trees (numTrees = 10) and two options for maximum tree depth (maxDepth = 5 and 10). Additionally, the number of folds was reduced to 2 to keep the training time manageable.The best configuration resulted in a tuned accuracy of 0.2225 on the test set. This suggests that either the model capacity was too limited under the constraints, or that the random forest classifier may require denser or more informative features to perform well in this specific multiclass text classification task.
These poor results are in line with our expectations, however, as TF-IDF produces high-dimensional sparse vectors, which are less ideal for tree-based models.

3.3.2 Word2Vec

To address the limitations of sparse, high-dimensional TF-IDF features in tree-based models, we replaced the feature extraction method with Word2Vec. Unlike TF-IDF, which treats words as independent tokens, Word2Vec embeds each word into a low-dimensional, dense vector space based on its usage context. This embedding is learned such that words with similar meanings or usage patterns end up with similar vector representations. In our case, Spark's Word2Vec was applied with a reduced vector size (10 for random forest, 100 for logistic regression) due to runtime constraints. The resulting document vectors, derived by averaging word vectors, provide a more compact and semantically informative input for our random forest classifier. The same pipeline logic is used as with TF-IDF, where the hashing and IDF are replaced with:

```
word2vec = Word2Vec(inputCol="filtered", outputCol="features",
                    vectorSize=10, minCount=5)
```

For accurate comparison, we apply both the random forest and logistic regression to these embeddings, though we expect this to mainly improve the accuracy of the random forest as the data becomes more compatible with the kind of hierarchical decision making that RF's use. By feeding the model dense, semantically meaningful vectors rather than sparse token frequencies, we expect the trees to identify clearer and more robust decision boundaries, especially when data is limited or noisy.

Logistic Regression We evaluated logistic regression on Word2Vec embeddings using the same pipeline as in previous setups. As mentioned in the introduction paragraph of Word2Vec, our logistic regression pipeline had Word2Vec transform the text into dense 100-dimensional vectors by learning semantic relationships between the words. When trained on these embeddings, logistic regression achieved an accuracy of 0.5788 on the test set, which is slightly lower than the 0.5969 obtained with TF-IDF. This drop is expected, as averaging word vectors may lose some discriminative details captured by sparse TF-IDF weights, especially when document structure or specific keywords are crucial for classification.

Random Forest We next applied the random forest classification on the Word2Vec embeddings. The pipeline again follows the same structure, but with a reduced vector size for the Word2Vec of 1O for performance reasons. The resulting model achieved an accuracy of 0.2888, which is quite a lot higher than with TF-IDF features, but still far below logistic regression. While Word2Vec's dense embeddings are generally more compatible with tree-based models, the limited vector dimensionality, shallow tree depth, and small forest size constrained the model's ability to capture complex patterns in the data. These results reflect the trade-offs made to accommodate resource limitations during local computation.

3.3.3 LLM: MiniLM

Given the hardware limitations of the local machine, especially when working with embeddings and large models, we transitioned to Google Colab, which provides access to powerful resources including GPU's from Google Services. To keep the modeling efficient yet effective, we opted for MiniLM (Minimal Language Model), a lightweight transformer that produces high-quality sentence embeddings with a fraction of the computational cost required by larger models like BERT. Despite its compact size, MiniLM is designed to retain strong performance on semantic tasks, making it suitable for resource-constrained environments.
In our implementation, we used the all-MiniLM-L6-v2 model from the SentenceTransformers library to generate dense and contextual embeddings for each document. These embeddings encapsulate semantic relationships between words and phrases, moving beyond traditional token-based approaches

like TF-IDF or Word2Vec. MiniLM thus represents a shift toward contextualized language understanding, offering a more nuanced and effective feature representation—especially valuable when sufficient compute power is available.

```
# Embed text with MiniLM
from sentence_transformers import SentenceTransformer
model = SentenceTransformer('all-MiniLM-L6-v2')
texts = df['text'].tolist()
labels = df['main_category'].tolist()
embeddings = model.encode(texts, show_progress_bar=True)
```

This code loads the MiniLM model and applies it to the dataset's combined title and summary text. Each document is encoded into a fixed-size vector, enabling downstream classifiers to work with semantically rich, dense input features.

Logistic Regression After generating document embeddings using MiniLM, we trained a logistic regression classifier to predict the main category of each document. The dataset was split into training and test sets using an 80/20 split. A higher max iteration was specified to ensure convergence given the dense, high-dimensional embedding inputs. The model achieved an overall accuracy of 66 percent, with particularly strong performance on well-represented categories such as cs.CV, cs.CL, and astro-ph.GA.
The classification report provided precision, recall and F1-scores for each category. It's found that while the model performs well on more frequently present categories, it struggles for underrepresented labels where zero precision is yielded due to no predicted samples. This emphasizes the main limitation in predictive modeling for this assignment. The overall results are found in Table 2.

```
# Split
X_train, X_test, y_train, y_test = train_test_split(embeddings,
labels, test_size=0.2, random_state=42)
# Train
clf = LogisticRegression(max_iter=1000)
clf.fit(X_train, y_train)
# Evaluate
print(classification_report(y_test, clf.predict(X_test)))
```

Metric	Precision	Recall	F1-score
Accuracy			0.66
Macro Avg	0.49	0.38	0.40
Weighted Avg	0.64	0.66	0.64

Table 2: Summary of classification performance using MiniLM + Logistic Regression

Splitting the data for this was after the embedding, which is considered acceptable as MiniLM is a pre-trained model that does not learn or adapt based on the dataset. The embedder itself isn't trained, so there is no risk of leakage from test data influencing the learning.

Logistic Regression with balanced class weights To improve the performance of our classifier on underrepresented categories, we retrained logistic regression using the class weight='balanced' parameter. This option automatically adjusts the weights inversely proportional to class frequencies, thereby placing more emphasis on minority classes during training. The intent is to mitigate the bias toward majority classes that is common in imbalanced datasets, like ours, which spans over 100 different categories with uneven distribution.

Metric	Precision	Recall	F1-score
Accuracy			0.57
Macro Avg	0.41	0.54	0.45
Weighted Avg	0.66	0.57	0.59

Table 3: Summary of classification performance using Logistic Regression with `class_weight=balanced`

The model achieved an accuracy of 57 percent, slightly lower than the 66 percent observed with unweighted logistic regression. However, this trade-off is expected as the model penalizes mistakes on minority classes more heavily. Accuracy alone can be misleading in imbalanced settings; the balanced model significantly improved recall and F1-scores for many low-frequency categories. While precision dropped for some dominant classes, the overall macro-average F1-score improved, suggesting a more equitable performance across the entire label space. This makes the balanced model valuable when coverage of all categories, not just the most common ones, is important.

XGBoost Lastly, we made initial explorations with XGBoost on MiniLM Embeddings. XGBoost is a gradient-boosted tree ensemble model with high accuracy (see assignment 1). We applied it to the same dense document embeddings generated by MiniLM, leveraging their semantic richness. MiniLM was not retrained or altered for this step—its pretrained sentence embeddings served as input features to the XGBoost model. Because XGBoost requires numeric class labels, we used a LabelEncoder to convert our string-based category labels into integers.

Metric	Precision	Recall	F1-score
Accuracy			0.56
Macro Avg	0.42	0.33	0.36
Weighted Avg	0.54	0.56	0.54

Table 4: Summary of classification performance using XGBoost Classifier

This early XGBoost run achieved an accuracy of 56 percent, placing it just below the class-balanced logistic regression model. While some categories performed well, many minority classes still suffered from low recall and zero precision. This was likely due to the high number of classes and the class imbalance. Additionally, since XGBoost is designed to work best with structured tabular data, it may not fully capitalize on the dense, contextual properties of MiniLM embeddings without further tuning.

Fine-tuning MiniLM To further enhance performance, we experimented with fine-tuning the MiniLM model using the Hugging Face Trainer API. The goal was to adapt the pretrained sentence encoder to our specific classification task and dataset, given that the categories are not natural language. We trained the model for 3 epochs and observed a gradual decrease in training and validation loss. However, the final evaluation accuracy plateaued at 39.4 percent, notably lower than the 66 percent achieved with logistic regression on frozen MiniLM embeddings.

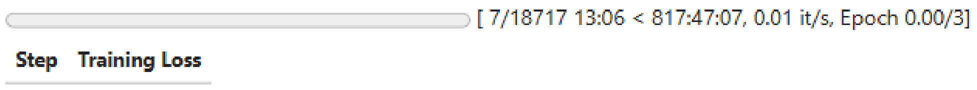
Epoch	Training Loss	Validation Loss	Accuracy
1	3.5684	2.9925	0.3102
2	2.7386	2.5902	0.3821
3	2.4546	2.4729	0.3939

Table 5: Training progress during `MiniLM` fine-tuning

This result suggests that without substantial hyperparameter tuning, data balancing, or training regularization, fine-tuning a pretrained transformer like MiniLM may lead to underfitting (low accuracy) or overfitting (poor generalization). In our case, the default training setup (3 epochs, batch size of 16, learning rate of 2e-5) led to only modest improvements in accuracy over time, reaching a peak of 39.4 percent. While the model architecture was correctly adapted for multiclass classification (AutoModelForSequenceClassification with numlabels set), the relatively small dataset and long-tail class distribution made optimization challenging. Moreover, because tokenization and model training occurred after the train/test split, the setup correctly avoids data leakage, unlike embedding-based pipelines where the split must come before feature computation.
Given these constraints and results, we chose not to further pursue fine-tuning and instead focused on using frozen MiniLM embeddings with logistic regression, which achieved better performance and greater training stability.

3.3.4 LLM: Zero-Shot Classification with BART-large-mnli

We further explored whether more modern generative language models can perform without any task-specific training. This was done by testing zero-shot classification using the facebook/bart-large-mnli model. This approach is based on BART (Bidirectional and Auto-Regressive Transformers), a transformer architecture pretrained for denoising and later fine-tuned on the MNLI (Multi-Genre Natural Language Inference) task. In this setup, the model determines the most likely label among a set of candidate labels by framing classification as a series of textual entailment problems.
Using Hugging Face's pipeline(”zero-shot-classification”), we applied this method to a small sample of our dataset (given the limited computational abilities of locally running this). Each document's text was classified against a list of candidate labels drawn from the unique category values in that sample. However, the performance was poor with only 10 percent accuracy on 10 samples. This is unsurprising for two key reasons. First, the input documents are technical scientific abstracts, not general natural language text, which differs significantly from what BART was pretrained on. Second, the candidate labels were symbolic tags like quant-ph or cs.AR, abbreviations that carry little semantic meaning unless interpreted in context, something BART is not equipped to do without fine-tuning.
While zero-shot learning is powerful in general-purpose NLP tasks with natural labels, it proved ineffective here due to domain mismatch and non-descriptive label formats.
We tried to fine tune a pretrained BART-base model (facebook/bart-base) for the multiclass classification. The input texts were constructed by concatenating each document's title and summary, and labels were encoded numerically. After tokenizing the data, we initialized a classification head on top of BART and configured it for training with 3 epochs. However, as shown in Figure 7, training on a CPU without GPU acceleration resulted in an estimated runtime of over 817 hours (34 days). This renders the process infeasible given available resources. Due to these constraints, we did not pursue BART fine-tuning further and instead focused on more efficient approaches like frozen MiniLM embeddings.



```
[ 7/18717 13:06 < 817:47:07, 0.01 it/s, Epoch 0.00/3]
Step   Training Loss
```

Figure 7: Fine-tuning BART model estimated to take over 800 hours due to CPU-only execution

3.3.5 Conclusion modeling

After exploring a range of modeling strategies, from traditional classifiers like logistic regression and random forests to advanced approaches using pretrained language models such as MiniLM, XGBoost, and BART, we selected logistic regression on frozen MiniLM embeddings as the final model for deployment. This approach achieved the highest overall accuracy (66 percent), while maintaining simplicity and interpretability. Although we tested enhancements such as class weighting, fine-tuning transformer models, and ensemble methods like XGBoost, none of these alternatives provided sufficient performance gains to justify their added complexity or resource requirements. In particular, fine-tuning large language models proved infeasible on local hardware, and tree-based models struggled with the high dimensionality and class imbalance in the dataset. Thus, we chose to deploy the initial logistic regression model without class balancing as the most robust and practical solution. As such, we downloaded the model and embedder from the Colab notebook, and saved it in the Spark folder, to utilize it further in a deployed setting.

```
joblib.dump(clf, 'logreg_model.pkl')
model.save('minilm_model')
```

3.4 Deploying Model on Streamed Data

For deploying the model, we extended the example notebook spark-streaming-example-predictions.ipynb. The goal is to receive live data from a socket, process incoming JSON records, embed the text using the previously saved MiniLM model, and classify it using our trained logistic regression model. We begin by initializing a local Spark Streaming context and defining a custom thread to run it without blocking the notebook interface:

```
ssc = StreamingContext(sc, 10) # 10-second micro-batches
ssc_t = StreamingThread(ssc)
ssc_t.start()
```

The core logic resides in the process() function, which is responsible for reading each RDD of incoming data, performing the necessary transformations, and producing predictions. The complete function is shown below:

```
if not globals()['models_loaded']:
embedder = SentenceTransformer(model_path)
clf = joblib.load(clf_path)
models_loaded = True
```

Each document's title and summary are combined into a single input string, then encoded using MiniLM and passed to the classifier. Predictions are merged with the original data and displayed alongside the true category label:

```
def process(time, rdd):
if rdd.isEmpty():
print(f"[{time}]␣—␣Empty␣batch.")
return
print(f"\n=========␣{str(time)}␣=========")
try:
    # Step 1: Read JSON from RDD
    df = spark.read.json(rdd)

    # Step 2: Clean missing title/summary
    df = df.fillna({'title': '', 'summary': ''})

    # Step 3: Combine into a single text column
    df = df.withColumn("text", concat_ws("␣", col("title"), col("summary"))
    # Step 4: Get text to predict
    texts = [row[”text”] for row in df.select(”text”).collect()]

    # Step 5: Load models on first run only
    if not globals()['models_loaded']:
        model_path = ”C:/Users/arthu/Desktop/spark/notebooks/minilm_model”
        clf_path = ”C:/Users/arthu/Desktop/spark/notebooks/logreg_model.pkl
        globals()['embedder'] = SentenceTransformer(model_path)
        globals()['clf'] = joblib.load(clf_path)
        globals()['models_loaded'] = True
        print(”Model␣and␣embedder␣loaded.”)

    # Step 6: Predict
    embeddings = globals()['embedder'].encode(texts)
    predictions = globals()['clf'].predict(embeddings)

    # Step 7: Assemble prediction DataFrame
    pred_df = spark.createDataFrame(zip(texts, [str(p) for p in predictions

    # Step 8: Include true label (first category)
    df = df.withColumnRenamed(”categories”, ”true_label”)

    # Step 9: Merge and show
    result_df = df.join(pred_df, on=”text”, how=”left”)
    result_df.select(”aid”, ”text”, ”true_label”, ”pred”).show(truncate=Fal

    # Step 10: Save results to disk
    timestamp = str(time).replace(”␣”, ”_”).replace(”:”, ”-”)
    output_path = f”./predictions_batch_{timestamp}.json”
    result_df.select(”aid”, ”text”, ”true_label”, ”pred”).write.json(output
    print(f”Saved␣predictions␣to␣{output_path}”)

except Exception as e:
    print(f”Error␣processing␣batch:␣{e}”)
```

The results are output in a human-readable format and also saved to disk for logging and later inspection:

```
result_df.select(”aid”, ”text”, ”true_label”, ”pred”).write.json(output_pat
```

To test the pipeline, we simulate an incoming record using a manually constructed JSON string and pass it to process() via a parallelized RDD. Finally, the actual streaming connection is established as follows:

```
lines = ssc.socketTextStream(”seppe.net”, 7778)
lines.foreachRDD(process)
```

By doing so, we collected eighteen prediction batches (with 20 or so streamed articles and predictions per batch). This is a total of 434 total predictions, which is a sufficient amount for our demonstrative purposes. Table **??** shows the output for one of such batches, where the title and abstract are merged together, and both the true label(s) and the predicted lable from the model are presented.

arXiv ID	Abstract (truncated)	True Labels	Prediction
`2505.19415v1`	*Towards Comprehensive and Explainable Evaluation of Multi-Modal Image Generation Models...* This work introduces MMIG-Bench, a unified benchmark that evaluates multi-modal image generators (like GPT-4o and Gemini) by pairing 4,850 annotated prompts with 1,750 reference images across 380 diverse subjects including humans, animals, and artistic styles.	`cs.CV`	`cs.CV`

Table 6: Example prediction from deployed MiniLM model in streaming context

In a separate notebook, we then loaded all the batches from the streamed environment (in which the predictions are included), using Python's glob and pandas libraries. We then concatenated these JSON files into a single Dataframe and computed classification metrics.

```
#Load all saved predictions from streamed batches
folders = glob.glob("predictions_batch_2025-")
dfs = []
for folder in folders:
parts = glob.glob(os.path.join(folder, "part-.json"))
for file in parts:
df = pd.read_json(file)
dfs.append(df)
#Combine and clean
df = pd.concat(dfs, ignore_index=True)
df["true_label"] = df["true_label"].str.split(",").str[0]
```

We then computed a classification report to assess how well the logistic regression model performed on streamed data using the MiniLM embeddings:

```
from sklearn.metrics import classification_report
print(classification_report(df["true_label"], df["pred"]))
```

The results (Table 7), show a weighted accuracy of 66 percent, consistent with the offline test set. Notably, some high-support categories such as cs.LG, cs.CL and cs.CV achieved strong precision and recall. This shows the model's effectivenss on commonly occuring classes. However, as was the case with the test data, many low-support categories had either poor or undefined metrics due to lack of predictions, which highlights the ongoing issue of class imbalance. There were also cases with no support, where classification was therefore impossible (see table 7).

Label	Precision	Recall	F1-Score	Support
`cs.LG`	0.68	0.70	0.69	64
`cs.CL`	0.69	0.83	0.75	58
`cs.CV`	0.76	0.89	0.82	56
`cs.AI`	0.58	0.25	0.35	28
`stat.ME`	0.00	0.00	0.00	0
`physics.bio-ph`	0.00	0.00	0.00	0
`accuracy`	–	–	0.66	434
`macro avg`	0.49	0.52	0.49	434
`weighted avg`	0.64	0.66	0.64	434

Table 7: Selected classification metrics showing performance on the most and least represented categories, along with overall scores.