CA6000 Advanced Data Analytics Assignment

Priority Classification of Customer Support Tickets Using Neural Networks

A Comparative Analysis using Logistic Regression, TextCNN, and BERT

Su Zhiyuan Tuo Xueting

1. Introduction

1.1 Background & Motivation

We chose customer support ticket classification for this project because it's something we can relate to - we've all waited too long for a response to an urgent issue. Companies receive thousands of support tickets daily, and manually sorting them by priority is time-consuming and inconsistent. The idea here is to use machine learning to automatically classify tickets as high, medium, or low priority, which could help support teams respond to critical issues faster.

We were curious whether modern NLP models would work well on this kind of text. Support tickets are tricky - they're often informal, sometimes vague, and the line between "medium" and "low" priority can be pretty subjective even for humans.

1.2 What We Set Out to Do

Our goal was to compare different approaches to see what works best for this problem:

- Start with a simple baseline (TF-IDF + Logistic Regression) to establish a reference point
- Try a neural network approach with TextCNN
- Use a pre-trained transformer model (DistilBERT) to see if transfer learning helps

We wanted to understand the trade-offs between these approaches - not just accuracy, but also training time and practicality.

---

2. Problem Definition

2.1 Task Description

This is a 3-class text classification problem. Given a support ticket (text), the model needs to predict whether it's high, medium, or low priority. We're treating it as a standard supervised learning task with the dataset labels as ground truth.

For evaluation, we're using both accuracy and macro-F1 score. Accuracy alone can be misleading if the classes are imbalanced, so macro-F1 gives a better picture of how well the model handles each class.

2.2 Constraints and Assumptions

A few things to keep in mind about this setup:

- The dataset is synthetic, so it might not capture all the messiness of real support tickets
- All tickets are in English
- We used DistilBERT instead of full BERT because of GPU memory constraints on our machine
- The class distribution isn't perfectly balanced (more on this later), which is why we're reporting per-class metrics

---

3. Dataset Description

3.1 Where the Data Comes From

We used the "customer-support-tickets" dataset from Hugging Face (uploaded by Tobi-Bueck). It contains synthetic customer support tickets covering domains like software, healthcare, financial services, and infrastructure. Each ticket has a subject, body text, and priority label.

We filtered to keep only English tickets and combined subject + body into a single text field.

3.2 Dataset Size and Split

After cleaning, we ended up with 28,261 samples total. We split them 70/15/15:

Split	Samples
Training	19,782
Validation	4,239
Test	4,240

We used stratified splitting to maintain similar class distributions across all splits.

3.3 Data Format

The processed data is stored as CSV files with these columns:

- `text`: combined subject and body
- `priority`: the label (high/medium/low)
- `cleaned_text`: preprocessed version for modeling

3.4 How We Cleaned the Data

We wrote a simple cleaning function that:

- Converts everything to lowercase
- Removes URLs and email addresses (they don't help with classification)
- Normalizes whitespace

```
def basic_clean(text):
    text = text.lower()
    text = re.sub(r'http\S+|www\S+', '', text)
    text = re.sub(r'\S+@\S+', '', text)
    text = ' '.join(text.split())
    return text.strip()
```

Nothing fancy, but it works. We applied the same cleaning to all data used across all models to keep things fair.

Each model needs a different input format though:

Model	How text is encoded

Model	How text is encoded
Logistic Regression	TF-IDF vectors (max 10k features, unigrams + bigrams)
TextCNN	Integer sequences using a vocabulary we built from training data
DistilBERT	WordPiece tokens using HuggingFace's tokenizer

3.5 Looking at the Data

Here's the class distribution in the training set:

Priority	Count	Percentage
High	7,698	38.9%
Medium	8,041	40.6%
Low	4,043	20.4%

So there's some imbalance - "low" priority has about half the samples of the other classes. This is actually realistic for support tickets (people tend to think their issues are more urgent than they are).

Text length stats:

- Average: ~411 characters / ~60 words
- Range: 16 to 1,715 characters
- Most tickets are between 200-600 characters

One thing we noticed: text length doesn't really correlate with priority. We initially thought maybe longer tickets would be lower priority (more detail = less urgent?), but that's not the case here.

---

4. Why We Chose These Models

We went with a "start simple, then add complexity" approach.

4.1 Baseline: TF-IDF + Logistic Regression

We started here because it's fast and gives you a reasonable baseline. TF-IDF turns text into numerical features based on word frequencies, and logistic regression finds a linear decision boundary.

The main limitation is that it treats text as a "bag of words" - word order doesn't matter. "Not good" and "good not" would look the same to this model.

Settings we used:

- TF-IDF with max 10,000 features, including bigrams
- Logistic regression with balanced class weights

4.2 TextCNN

This was our first neural network attempt. TextCNN uses convolutional filters to capture local patterns (like n-grams) in the text. The idea is that certain word combinations might signal priority level.

Architecture:

- Embedding layer (128 dimensions)
- Three parallel conv layers with filter sizes 3, 4, 5 (100 filters each)
- Max pooling, then concatenation
- Dropout (0.5) and final classification layer

We trained it for 5 epochs with Adam optimizer. One thing we learned: TextCNN really benefits from pre-trained embeddings, which we didn't use here. That probably hurt its performance.

4.3 DistilBERT

Finally, we tried fine-tuning DistilBERT. It's a smaller version of BERT that's been pre-trained on a huge amount of text, so it already "understands" language to some degree.

We just added a classification head on top and fine-tuned the whole thing for 3 epochs. The learning rate is much smaller (2e-5) because you don't want to destroy the pre-trained knowledge.

Training took about 30 minutes on our GPU, compared to under a minute for logistic regression.

4.4 Other Things We Could Have Tried

Looking back, there are some alternatives we didn't explore:

- SVM or Naive Bayes as additional baselines
- Using pre-trained word embeddings (GloVe, Word2Vec) for TextCNN
- Larger models like BERT-base or RoBERTa
- Handling class imbalance more explicitly (oversampling, focal loss)

Maybe for future work.

---

5. Methodology

5.1 Overall Pipeline

Our workflow was pretty standard:

1. Load raw data, filter to English, clean text
2. Split into train/val/test (70/15/15, stratified)
3. For each model: prepare inputs, train, evaluate
4. Compare results on the held-out test set

All three models used the exact same data splits and cleaned text, so the comparison is fair.

5.2 Training Details

Logistic Regression:

- Straightforward sklearn fit/predict
- Used class_weight='balanced' to handle imbalance
- Training takes less than a minute

TextCNN:

- Built vocabulary from training data (all words that appear at least once)
- Padded/truncated sequences to length 256
- Batch size 64, 5 epochs
- Used weighted cross-entropy loss

DistilBERT:

- Max sequence length 128 (tickets are usually short enough)
- Batch size 32, 3 epochs
- AdamW optimizer with learning rate 2e-5
- Linear learning rate warmup

5.3 What We Measured

For evaluation, we computed:

- Accuracy (overall)
- Precision, recall, F1 for each class
- Macro-averaged F1 (treats all classes equally)
- Confusion matrix (to see where mistakes happen)

We focused on macro-F1 as the main metric because accuracy can be inflated by just predicting the majority class.

---

6. Implementation Notes

6.1 Code Organization

We structured the project with separate notebooks for each model:

- `01_eda.ipynb` - data exploration
- `02_baseline_ml.ipynb` - logistic regression
- `03_cnn_model.ipynb` - TextCNN
- `04_bert_model.ipynb` - DistilBERT fine-tuning

Shared code (data loading, preprocessing, evaluation) is in the `src/` folder.

6.2 Some Issues We Ran Into

CUDA memory errors with BERT: At first we tried batch size 64, but our GPU kept running out of memory. Had to reduce to 32.

TextCNN underfitting: Our initial TextCNN wasn't learning much. Turned out we had the embedding dimension too small (was 50, changed to 128) and needed more filters.

Class imbalance: The "low" class kept getting worse predictions. Adding class weights to the loss function helped somewhat.

6.3 Reproducibility

We set random seeds where possible and saved all trained models. The notebooks should be runnable end-to-end, assuming you have the data in the right place.

---

7. Results

7.1 Test Set Performance

Here's how the three models compared on the held-out test set:

Model	Accuracy	Macro F1	Training Time
Logistic Regression	64.4%	63.6%	<1 min
TextCNN	60.4%	59.3%	~10 min
DistilBERT	**73.2%**	**72.4%**	~30 min

So DistilBERT wins pretty clearly. What surprised us was that TextCNN actually did worse than logistic regression on this dataset - we'll discuss why below.

7.2 Per-Class Breakdown

Logistic Regression:

```
              precision    recall  f1-score   support
        high       0.69      0.67      0.68      1604
         low       0.54      0.65      0.59       876
      medium       0.67      0.62      0.64      1760
   macro avg       0.63      0.65      0.64      4240
```

TextCNN:

```
              precision    recall  f1-score   support
        high       0.61      0.74      0.67      1604
         low       0.50      0.62      0.55       876
      medium       0.70      0.47      0.56      1760
   macro avg       0.60      0.61      0.59      4240
```

DistilBERT:

```
              precision    recall  f1-score   support
        high       0.75      0.73      0.74      1604
```

```
           low       0.65      0.72      0.68       876
        medium       0.74      0.73      0.73      1760
     macro avg       0.71      0.73      0.72      4240
```

DistilBERT is clearly more balanced - it doesn't sacrifice one class to do well on others.

7.3 Where Models Go Wrong

Looking at the confusion matrices, the main issue across all models is confusion between "medium" and "low" priority. This makes sense - the boundary between these two is fuzzy even for humans.

"High" priority is easier to identify, probably because urgent tickets use more distinctive language (words like "critical", "down", "urgent", "ASAP").

7.4 Why TextCNN Underperformed

We were surprised that TextCNN did worse than the baseline. A few possible reasons:

1. **No pre-trained embeddings:** We trained embeddings from scratch, which doesn't work great with ~20k training samples
2. **Dataset might be too clean:** Synthetic data often lacks the noise and variety of real text, which neural networks need to generalize
3. **Hyperparameter tuning:** We didn't do extensive tuning; there's probably a better configuration

If we had more time, we'd try initializing with GloVe embeddings and doing a proper hyperparameter search.

---

8. Discussion

8.1 What We Learned

The biggest takeaway is that pre-training matters a lot for NLP. DistilBERT, despite being used almost out-of-the-box, beat both other approaches significantly. It's already seen so much text during pre-training that fine-tuning on 20k examples is enough.

On the other hand, the simple TF-IDF + logistic regression baseline held up surprisingly well. If you need something quick and interpretable, it's not a bad choice.

8.2 Practical Considerations

If we were deploying this for a real company:

- **For a quick prototype:** Logistic regression. It's fast, explainable, and decent accuracy.
- **For best accuracy:** DistilBERT. The 30-minute training time is fine if you're not retraining constantly.
- **For edge deployment:** Might need to distill the BERT model further or use something lighter.

8.3 Limitations

Some things that could affect how well these results generalize:

- Synthetic data isn't quite the same as real tickets

- We only tried one random seed for the data split
- Hyperparameter tuning was minimal
- Didn't try any ensemble methods

8.4 What We'd Do Differently

With more time, we would:

- Try pre-trained embeddings for TextCNN
- Do k-fold cross-validation instead of a single split
- Experiment with data augmentation (paraphrasing, back-translation)
- Maybe try an ensemble of logistic regression + BERT

---

9. Conclusion

We built a ticket classification system and compared three approaches. The main findings:

Performance ranking:

1. DistilBERT: 73.2% accuracy, 72.4% macro-F1
2. Logistic Regression: 64.4% accuracy, 63.6% macro-F1
3. TextCNN: 60.4% accuracy, 59.3% macro-F1

Key insights:

- Pre-trained models (BERT) have a big advantage for NLP tasks, even with moderate training data
- Simple baselines can be surprisingly competitive and shouldn't be skipped
- The medium/low priority boundary is inherently difficult - might need better labeling guidelines or additional features

For this particular task, we'd recommend DistilBERT if accuracy is important, or logistic regression if you need speed and interpretability.

---

10. References

Papers:

Devlin et al. (2019). "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding." NAACL 2019.

Kim, Y. (2014). "Convolutional Neural Networks for Sentence Classification." EMNLP 2014.

Sanh et al. (2019). "DistilBERT, a distilled version of BERT." arXiv:1910.01108.

Libraries:

- PyTorch: https://pytorch.org/
- HuggingFace Transformers: https://huggingface.co/transformers/
- scikit-learn: https://scikit-learn.org/

Dataset:

Tobi-Bueck/customer-support-tickets on Hugging Face: https://huggingface.co/datasets/Tobi-Bueck/customer-support-tickets

Disclosure of AI-assisted work: We used Claude as a programming assistant during the project. Specifically, it was used to help with code generation for model classes and training loops, debugging runtime errors, and formatting this report. The dataset selection, preprocessing pipeline design, model training, and evaluation were performed by the authors, and all reported results were generated from our own executed code.