Clinical Valence Behavioral Testing: Paper Structure and Code Review

Document Overview

This document addresses two key questions:

1. What would a research paper look like for this behavior testing work?
2. A comprehensive code review of the testing pipeline

Part 1: Research Paper Structure for Behavioral Testing

1.1 Abstract Template

Title: Behavioral Testing of Clinical NLP Models: Measuring the Impact of Valence-Laden Language on ICD Diagnosis Predictions

Abstract Structure:

> Background: Clinical decision support systems increasingly rely on NLP models trained on electronic health records. However, these models may exhibit biases based on subjective language used to describe patients.

>

> Objective: To systematically evaluate how valence-laden descriptors (pejorative, laudatory, neutral) affect ICD diagnosis predictions from a clinical NLP model.

>

> Methods: We implement a behavioral testing framework that applies controlled linguistic perturbations to clinical notes from MIMIC-III, measuring prediction shifts across 1,266 ICD-9 diagnosis codes using the DATEXIS/CORe clinical diagnosis prediction model.

>

> Results: [To be filled with actual results showing significant prediction shifts for specific diagnosis codes]

>

> Conclusions: [Summary of findings regarding model sensitivity to valence language]

1.2 Introduction Section

Background Context

The introduction should establish:

1. **The Problem:** Clinical NLP models are increasingly used for automated diagnosis coding, triage, and clinical decision support. However, clinical notes often contain subjective language describing patient behavior.

2. **Prior Work:**
• van Aken et al. (2021) introduced behavioral testing for clinical NLP
• Studies on stigmatizing language in medical records (Beach et al., Park et al.)
• CheckList methodology for NLP model testing (Ribeiro et al., 2020)

3. **Research Gap:** Limited systematic evaluation of how valence-laden patient descriptors affect model predictions

4. **Study Objectives:**
• Quantify prediction shifts when pejorative terms are introduced
• Compare effects of laudatory vs. pejorative language
• Identify which diagnosis codes are most sensitive to valence shifts

1.3 Methods Section

1.3.1 Dataset Description

Attribute	Value
Source	MIMIC-III Clinical Database
Sample Type	Discharge summaries
Format	CSV with id, text, short_codes
Code System	ICD-9-CM (3-digit groups)
Minimum Code Frequency	100 occurrences

1.3.2 Model Description

Component	Specification
Base Model	DATEXIS/CORe-clinical-diagnosis-prediction

Architecture	BioBERT-based transformer
Task	Multi-label ICD code classification
Output	1,266 diagnosis code probabilities
Attention Analysis	Layer 11, Head 11

1.3.3 Shift Transformations

The paper should describe each shift type:

Neutralize Shift (Baseline):

- Removes ALL valence terms from text
- Creates objective baseline for comparison
- Terms removed: all pejorative, laudatory, and neutral descriptors

Pejorative Shift (Treatment 1):

- Inserts negative patient descriptors
- Four severity levels tested:
- NON_COMPLIANT: "non-compliant", "negligent", "careless"
- UNCOOPERATIVE: "uncooperative", "difficult", "argumentative"
- RESISTANT: "resistant", "hostile", "aggressive"
- DIFFICULT: "manipulative", "malingering", "drug-seeking"

Laudatory Shift (Treatment 2):

- Inserts positive patient descriptors
- Four levels: COMPLIANT, COOPERATIVE, PLEASANT, RESPECTFUL
- Terms: "compliant", "cooperative", "courteous", "considerate"

Neutral Valence Shift (Treatment 3):

- Inserts objective descriptors
- Terms: "typical", "average", "regular", "presenting"

1.3.4 Insertion Strategy

Text modification follows a systematic approach:

1. Identify patient characteristic position (age, gender, patient type)
2. Insert descriptor after identified position
3. Remove conflicting valence terms
4. Preserve medical content and clinical facts

1.3.5 Statistical Analysis

Test	Purpose	Reference

Paired t-test	Parametric comparison	Standard
Wilcoxon signed-rank	Non-parametric alternative	Standard
Permutation test	Approximate randomization	Yeh (2000)
Cohen's d	Effect size	Cohen (1988)
Hedges' g	Bias-corrected effect size	Hedges (1981)
FDR correction	Multiple comparisons	Benjamini-Hochberg

1.4 Results Section Structure

Expected Tables and Figures

Table 1: Dataset characteristics

- Number of samples per shift
- Code frequency distribution

Table 2: Overall prediction shift summary

- Mean probability change per shift type
- Number of significantly affected diagnosis codes

Table 3: Top 10 most affected diagnosis codes

- Diagnosis code
- Mean shift magnitude
- Effect size (Cohen's d)
- Corrected p-value

Figure 1: Distribution of prediction shifts across valence types

- Box plots or violin plots comparing shifts

Figure 2: Heatmap of diagnosis probability changes

- Rows: Top affected diagnosis codes
- Columns: Shift types (pejorative, laudatory, neutral)

Figure 3: Attention weight analysis

- Which tokens receive increased/decreased attention after shifts

1.5 Discussion Topics

1. **Clinical Implications:** How do prediction shifts affect potential patient care?
2. **Bias Patterns:** Which diagnosis categories are most sensitive?
3. **Model Robustness:** Should clinical NLP models ignore valence language?
4. **Limitations:** MIMIC-III specific, single model tested
5. **Future Work:** Cross-model validation, mitigation strategies

1.6 Expected Findings Format

A typical findings paragraph would read:

> "We observed that introducing pejorative descriptors significantly altered diagnosis predictions for [N] out of 1,266 ICD codes ($p < 0.05$, FDR-corrected). The largest effect sizes were observed for [specific codes], with Cohen's d ranging from [X] to [Y]. Interestingly, laudatory descriptors showed [similar/opposite] effects, suggesting [interpretation]."

Part 2: Pipeline Code Review

2.1 Architecture Overview

The pipeline follows a clean, modular architecture:

```
main.py (Entry Point)
|
+-- config_loader.py (Configuration)
|
+-- valence_testing.py (Test Orchestration)
| |
| +-- test_shifts/ (Shift Implementations)
| | +-- base_shift.py
| | +-- pejorative_shift.py
| | +-- laudatory_shift.py
| | +-- neutralVal_shift.py
| | +-- neutralize_shift.py
| |
| +-- prediction.py (Model Inference)
|
+-- statistical_analysis.py (Analysis)
|
+-- utils.py (Utilities)
```

2.2 Entry Point Analysis: main.py

Code Quality Assessment

Strengths:

1. **Clean CLI Interface:** Uses Python Fire for argument parsing (line 305)
```
if __name__ == '__main__':
fire.Fire(run)
```
2. **Configuration Hierarchy:** Supports config file + command-line overrides (lines 74-83)
```
test_set_path = parse_argument(test_set_path) if test_set_path else config.data.test_set_path
model_path = parse_argument(model_path) if model_path else config.model.name
```
3. **Reproducibility:** Random seed management (lines 84-90)
```
if random_seed is not None:
utils.set_random_seeds(random_seed)
if config.deterministic:
utils.configure_deterministic_mode()
```
4. **Comprehensive Logging:** Experiment parameters tracked (lines 97-106)

Areas for Improvement:

1. **Error Handling:** Could use more specific exception types
2. **Type Hints:** Some parameters lack full type annotations
3. **Magic Strings:** Shift prefix mapping (lines 161-166) could be configurable

2.3 Core Testing Framework: valence_testing.py

Class Design

The `BehavioralTesting` class follows single responsibility principle:

```
class BehavioralTesting:
def __init__(self, test_dataset_path: str, text_label: str = "text"):
# Load and validate dataset

def run_test(self, shift, predictor, save_path):
# Orchestrate test execution

def _apply_shift(self, shift: BaseShift):
# Apply transformation to all samples

def _validate_inputs(self, shift, predictor, save_path):
# Input validation
```

Strengths:

1. **Dataclass for Results:** Clean data structure (lines 9-19)
```
@dataclass
class TestResults:
shift_statistics: Dict
```

```
timestamp: str = field(default_factory=lambda: datetime.now().isoformat())
```

2. **Custom Exceptions:** BehavioralTestingError for clear error handling
3. **Separation of Concerns:** Validation, shifting, and prediction are separate methods

Code Pattern:

```
def run_test(self, shift, predictor, save_path):
self._validate_inputs(shift, predictor, save_path)
predictor.initialize_for_prediction(save_path)
groups, stats = self._apply_shift(shift)
for group_name, samples in groups.items():
predictor.predict_group(samples, group_name)
predictor.save_results(save_path)
return TestResults(shift_statistics=stats)
```

2.4 Shift Implementation: Base Class Pattern

Design Pattern: Template Method

The `BaseShift` class (base_shift.py) implements the Template Method pattern:

```
class BaseShift:
def make_shift(self, samples, return_stats=False):
# Template method - calls abstract methods
for group in self.get_groups(): # Abstract
shifted = self.get_shift_method(sample, group) # Abstract

def get_groups(self):
raise NotImplementedError

def get_shift_method(self, sample, group):
raise NotImplementedError
```

Critical Implementation Detail (lines 30-33):

```
# CRITICAL FIX: Each group now independently determines which samples to skip.
# A sample that returns None for one group may still be valid for another group.
```

This ensures samples are processed per-group, not globally filtered.

Pejorative Shift Implementation

Term Dictionary Structure (pejorative_shift.py, lines 35-48):

```
pejorative_mentions = {
PejorativeLevel.NON_COMPLIANT: [
"non-compliant", "negligent", "careless", "irresponsible", "unreliable"
],
PejorativeLevel.UNCOOPERATIVE: [
"uncooperative", "difficult", "problematic", "demanding", "argumentative"
```

```
],
# ... more levels
}
```

Transformation Logic (lines 111-177):

1. Check if target term already exists
2. Remove terms from other levels using regex with word boundaries
3. Find insertion position using `utils.find_patient_characteristic_position_in_text()`
4. Insert chosen term at identified position

2.5 Prediction Module: prediction.py

Class Hierarchy

```
Predictor (Abstract Base)
|
+-- TransformerPredictor
|
+-- DiagnosisPredictor
```

Key Methods Analysis

1. Attention Extraction (lines 89-136):

```
@torch.no_grad()
def inference_from_texts(self, text, layer_num, head_num, aggregation):
outputs = self.model(**inputs, output_attentions=True)
attentions = outputs.attentions[layer_num][0][head_num].cpu().numpy()
cls_attention = attentions[0, :][1:-1] # CLS attention, exclude special tokens
```

Sub-token Aggregation (lines 107-134):

Three strategies supported:

- `average`: Mean of sub-token weights (recommended)
- `sum`: Sum of sub-token weights
- `max`: Maximum sub-token weight

2. Batch Processing (lines 225-270):

Thread-safe file handling with context managers:

```
def predict_batch(self, batch_texts, note_ids, group_name):
file_handler = self._get_file_handler(group_name)
for note_id, sample in zip(note_ids, batch_texts):
attention_weights, words, logits = super().inference_from_texts(...)
diagnosis_probs = torch.sigmoid(logits).cpu().numpy()
# Write to diagnosis, attention, and clinical_notes CSVs
```

3. Checkpoint Management (lines 189-209):

State persistence for long-running jobs:

```
@dataclass
class PredictionState:
current_group: str
shift_type: str
processed_samples: int
total_samples: int
last_note_id: int
```

2.6 Statistical Analysis: statistical_analysis.py

Test Suite

Parametric Tests:

- `paired_ttest()`: Standard paired t-test with confidence intervals
- `mann_whitney_test()`: Unpaired non-parametric test

Non-Parametric Tests:

- `wilcoxon_test()`: Signed-rank test for paired samples

Permutation Tests (lines 182-232):

Implementation of Yeh (2000) approximate randomization:

```
def paired_permutation_test(self, baseline, treatment, n_permutations=10000):
observed_diff = np.mean(treatment - baseline)
flip_masks = np.random.choice([-1, 1], size=(n_permutations, n_samples))
permuted_diffs = flip_masks * diffs[np.newaxis, :]
permuted_means = np.mean(permuted_diffs, axis=1)
count_extreme = np.sum(np.abs(permuted_means) >= abs(observed_diff))
p_value = (count_extreme + 1) / (n_permutations + 1)
```

Multiple Comparison Correction (lines 338-358):

```
def correct_multiple_comparisons(self, p_values):
rejected, corrected_p, _, _ = multipletests(
p_values,
alpha=self.significance_level,
method=self.correction_method # 'fdr_bh' by default
)
```

2.7 Utility Functions: utils.py

Patient Characteristic Detection

The `find_patient_characteristic_position_in_text()` function (lines 141-248) is critical for natural text modification:

Pattern Categories:

1. **Age Patterns:** `45 yo`, `[Age over 90]`, `65-year-old`
2. **Gender Patterns:** `male`, `female`, `woman`, `gentleman`
3. **Patient Type:** `patient`, `inpatient`, `outpatient`
4. **Ethnicity:** `African-American`, `Hispanic`, `Asian`
5. **Medical Conditions:** `diabetic`, `hypertensive`, `pregnant`

Performance Optimization:

```
@lru_cache(maxsize=2048)
def find_patient_characteristic_position_in_text(text: str) -> int:
```

LRU caching prevents redundant regex matching.

Reproducibility Functions (lines 518-551)

```
def set_random_seeds(seed: int = 42):
random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)
if torch.cuda.is_available():
torch.cuda.manual_seed_all(seed)

def configure_deterministic_mode(deterministic=True):
torch.use_deterministic_algorithms(True, warn_only=True)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
```

2.8 Code Quality Summary

Overall Assessment

Category	Rating	Notes
Architecture	Excellent	Clean separation of concerns, modular design
Readability	Good	Well-documented, clear naming conventions
Error Handling	Good	Custom exceptions, validation methods
Testing Support	Good	Reproducibility features, checkpointing
Performance	Good	Batch processing, GPU optimization, caching

Extensibility	Excellent	Template pattern for shifts, plugin architecture

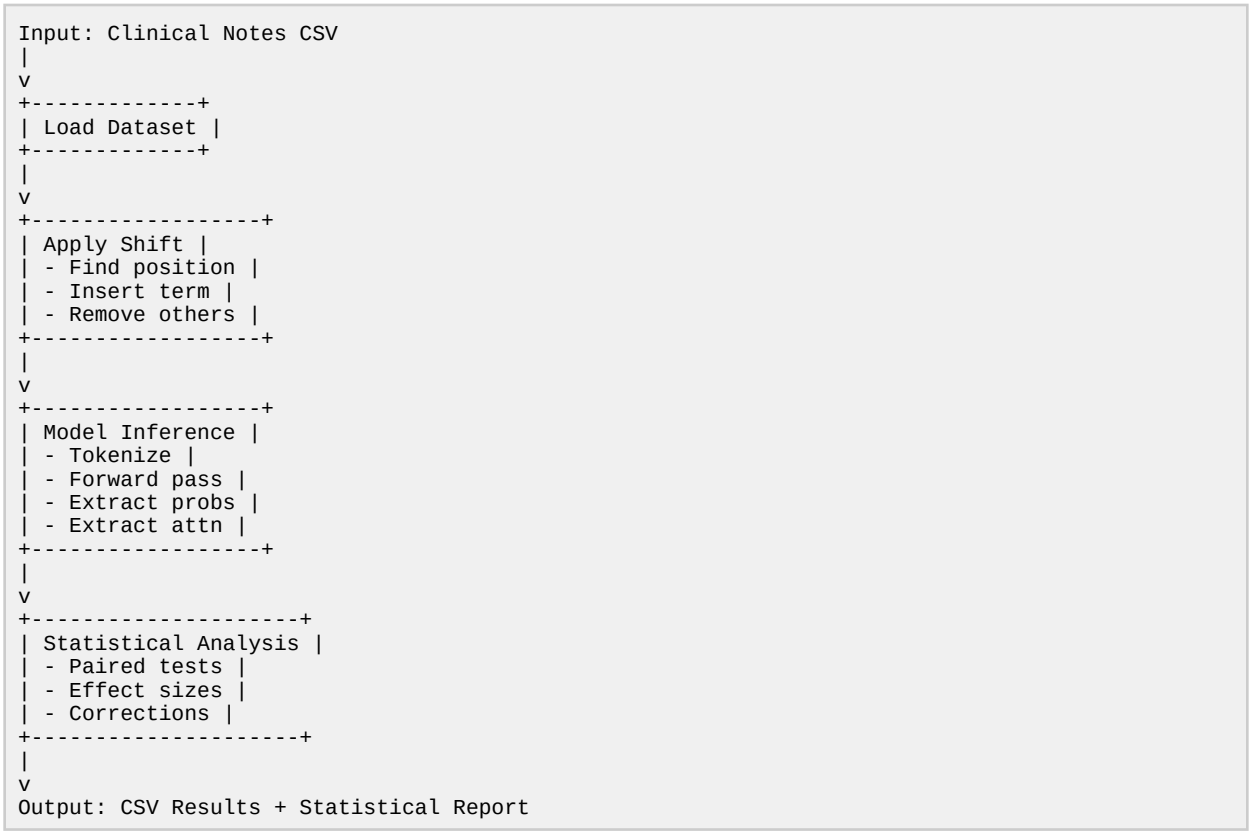
Best Practices Observed

1. **Type Hints:** Used throughout for clarity
2. **Dataclasses:** Clean data structures
3. **Context Managers:** Thread-safe file handling
4. **Logging:** Comprehensive experiment tracking
5. **Configuration:** YAML-based with CLI overrides

Recommendations for Future Development

1. **Unit Tests:** Add pytest test suite for shift transformations
2. **Integration Tests:** End-to-end pipeline validation
3. **Documentation:** API documentation with Sphinx
4. **CI/CD:** Automated testing pipeline
5. **Model Registry:** Support for multiple model backends

2.9 Data Flow Summary



Conclusion

This behavioral testing framework provides a rigorous methodology for evaluating clinical NLP model sensitivity to valence-laden language. The codebase demonstrates professional software engineering practices with clean architecture, comprehensive error handling, and extensive statistical analysis capabilities. The framework is well-suited for producing publishable research on model fairness and robustness in clinical settings.

References

1. van Aken, B., Herrmann, S., & Loser, A. (2021). What Do You See in this Patient? Behavioral Testing of Clinical NLP Models. NeurIPS R2C Workshop.

2. Ribeiro, M. T., Wu, T., Guestrin, C., & Singh, S. (2020). Beyond accuracy: Behavioral testing of NLP models with CheckList. ACL.

3. Yeh, A. (2000). More accurate tests for the statistical significance of result differences. COLING.

4. Benjamini, Y., & Hochberg, Y. (1995). Controlling the false discovery rate. JRSS-B.