Task 2 Completion report

Steps involved into completion of task:

Step 1: Take my question and have ChatGPT analyse it to explain to me what i'm trying to make

Step 2: Go to Claude AI and ask it to develop code based on the prompt provided.

Step 3: Had to download VsCode and Sublime to be able to run and analyse the code

Step 4: Take the generated code and then place it in VsCode and attempted to run the code

Step 5: Had to download libraries, update pips, update transformers and force it to be a particular version

Step 6: Run the code and found any issues, took them to ChatGPT and then ask it to correct

Step 7: Repeat the process until it either works or ChatGPT can no longer provide positive progression

Step 8: After successfully running, I was asked to implement a false positive tracker to the semantic constructor.

Step 9: ChatGPT failed to produce any progress as it started to go down a rabbit hole which led to it rewriting code and the code being significantly worse than before

Step 10: Took the original code to Claude AI and then had success instantly which when implemented and then I ran the code it ran successfully which led to the Semantic Constructor being able to train itself to identify, mark and fix false positives of some mock information.

Step 11: Check formatting, save code as to avoid corrupt data being lost

Step 12: Submit code and see for feedback

```
import torch
```

```
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
from transformers import BertModel, BertTokenizer, AdamW
print("Imports successful!")
import pandas as pd
import numpy as np
import json
from typing import List, Dict, Tuple, Any
from sklearn.model_selection import train_test_split
import logging
from collections import defaultdict

# Set up logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s -
%(message)s')
logger = logging.getLogger(__name__)

class FalsePositiveTracker:
   """
   Tracks false positive predictions in the semantic destructor to improve model
performance
   """
   def __init__(self, taxonomy):
       self.taxonomy = taxonomy
       self.false_positives = {
           "domain": defaultdict(list),  # Domain false positives
           "entity": defaultdict(list),  # Entity false positives
           "concept": defaultdict(list),  # Concept false positives
           "relation": defaultdict(list)  # Relation false positives
       }
       self.feedback_history = []
       self.learning_samples = []

   def record_false_positive(self, category, predicted, correct, text_sample,
confidence=None):
       """
       Record a false positive instance

       Args:
           category: Type of false positive ('domain', 'entity', 'concept',
'relation')
           predicted: What the model predicted
           correct: The correct classification
           text_sample: The text that was misclassified
```

```
            confidence: Confidence score of the false prediction (optional)
        """
        entry = {
            "text": text_sample,
            "predicted": predicted,
            "correct": correct,
            "confidence": confidence
        }

        self.false_positives[category][predicted].append(entry)

        # Add to feedback history
        self.feedback_history.append({
            "timestamp": pd.Timestamp.now().isoformat(),
            "category": category,
            "false_positive": entry
        })

        logger.info(f"Recorded {category} false positive: '{predicted}' should be
'{correct}'")

    def get_false_positive_stats(self):
        """Get statistics about recorded false positives"""
        stats = {}

        for category, fp_dict in self.false_positives.items():
            category_stats = {
                "total": sum(len(items) for items in fp_dict.values()),
                "by_prediction": {pred: len(items) for pred, items in
fp_dict.items()}
            }
            stats[category] = category_stats

        return stats

    def generate_learning_samples(self, limit=5):
        """Generate learning samples from false positives for model retraining"""
        samples = []

        # Focus on the most common false positives
        for category, fp_dict in self.false_positives.items():
            for predicted, entries in fp_dict.items():
                for entry in entries[:limit]:  # Take up to 'limit' samples for each
prediction
                    sample = {
```

```python
                    "category": category,
                    "text": entry["text"],
                    "correct_label": entry["correct"],
                    "previous_label": predicted
                }
                samples.append(sample)

        self.learning_samples = samples
        return samples

    def export_feedback(self, file_path):
        """Export all feedback to a JSON file"""
        data = {
            "false_positives": dict(self.false_positives),  # Convert defaultdict to
dict
            "feedback_history": self.feedback_history,
            "stats": self.get_false_positive_stats(),
            "learning_samples": self.learning_samples
        }

        with open(file_path, 'w') as f:
            json.dump(data, f, indent=2)

        logger.info(f"Exported false positive data to {file_path}")

    def import_feedback(self, file_path):
        """Import feedback data from a JSON file"""
        try:
            with open(file_path, 'r') as f:
                data = json.load(f)

            # Convert dict back to defaultdict
            for category in data["false_positives"]:
                for pred, entries in data["false_positives"][category].items():
                    self.false_positives[category][pred].extend(entries)

            self.feedback_history = data["feedback_history"]
            self.learning_samples = data.get("learning_samples", [])

            logger.info(f"Imported false positive data from {file_path}")
            return True
        except Exception as e:
            logger.error(f"Error importing feedback data: {e}")
            return False
```

```python
class DomainTaxonomy:
   """
   Represents the taxonomy of domains and sub-domains for semantic destruction
   """
   def __init__(self):
       self.taxonomy = {
           "regulatory": ["legitimacy", "framework"],
           "domain": ["political", "economic", "social_culture", "technological",
"legal"],
           "strategic": ["fake_news", "strategies"],
           "organisational_culture": ["optimisation", "reduction"],
           "organisation": ["capabilities", "innovation", "leadership"],
           "organisation_strategy": ["social_media", "facts"],
           "individual": ["demographic", "personality", "self_efficacy", "beliefs",
"trust"],
           "information_source": ["information_ecosystem", "risk", "propagation"]
       }

       # Flatten taxonomy for classification
       self.all_domains = []
       for major, minors in self.taxonomy.items():
           self.all_domains.append(major)
           self.all_domains.extend(minors)

       # Create id mappings
       self.domain_to_id = {domain: idx for idx, domain in
enumerate(self.all_domains)}
       self.id_to_domain = {idx: domain for domain, idx in
self.domain_to_id.items()}

       # Track major-minor relationships
       self.minor_to_major = {}
       for major, minors in self.taxonomy.items():
           for minor in minors:
               self.minor_to_major[minor] = major

   def get_domain_id(self, domain):
       """Get the ID for a given domain"""
       return self.domain_to_id.get(domain, -1)

   def get_domain_name(self, domain_id):
       """Get the domain name for a given ID"""
       return self.id_to_domain.get(domain_id, "unknown")
```

```python
    def get_major_domains(self):
        """Get all major domains"""
        return list(self.taxonomy.keys())

    def get_minor_domains(self, major_domain):
        """Get all minor domains for a major domain"""
        return self.taxonomy.get(major_domain, [])

    def get_major_for_minor(self, minor_domain):
        """Get the major domain for a given minor domain"""
        return self.minor_to_major.get(minor_domain)

    def is_major_domain(self, domain):
        """Check if a domain is a major domain"""
        return domain in self.taxonomy

    def is_minor_domain(self, domain):
        """Check if a domain is a minor domain"""
        return domain in self.minor_to_major

    def get_all_domains(self):
        """Get all domains (major and minor)"""
        return self.all_domains

    def num_domains(self):
        """Get the total number of domains"""
        return len(self.all_domains)


class SemanticDestructor:
    """
    A system that breaks down text into semantic components using transformer-based
NLP,
    with specific focus on regulatory, organizational, and information domains.
    """
    def __init__(self, model_name="bert-base-uncased", device=None):
        self.model_name = model_name
        self.device = device if device else torch.device("cuda" if
torch.cuda.is_available() else "cpu")
        logger.info(f"Using device: {self.device}")

        # Initialize taxonomy
        self.taxonomy = DomainTaxonomy()

        # Initialize false positive tracker
```

```
        self.fp_tracker = FalsePositiveTracker(self.taxonomy)

        # Initialize tokenizer and base model
        self.tokenizer = BertTokenizer.from_pretrained(model_name)
        self.model = SemanticDestructorModel(model_name,
self.taxonomy.num_domains()).to(self.device)

        # Components for semantic destruction
        self.components = {
            "domain_classifier": DomainClassifier(self.model, self.taxonomy),
            "entity_extractor": EntityExtractor(self.model),
            "concept_analyzer": ConceptAnalyzer(self.model, self.taxonomy),
            "relation_mapper": RelationMapper(self.model, self.taxonomy)
        }

    def fine_tune(self, corpus_path, batch_size=8, epochs=3, learning_rate=2e-5,
                  include_fp_samples=True):
        """Fine-tune the model on domain-specific data"""
        # Load and prepare domain data
        dataset = DomainDataset(corpus_path, self.tokenizer, self.taxonomy)

        # Add false positive samples to training data if available and requested
        if include_fp_samples and self.fp_tracker.learning_samples:
            logger.info(f"Adding {len(self.fp_tracker.learning_samples)} false
positive samples to training data")
            fp_samples = self._prepare_fp_samples(self.fp_tracker.learning_samples)
            dataset.add_samples(fp_samples)

        train_size = int(0.9 * len(dataset))
        val_size = len(dataset) - train_size
        train_dataset, val_dataset = torch.utils.data.random_split(dataset,
[train_size, val_size])

        train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)
        val_loader = DataLoader(val_dataset, batch_size=batch_size)

        # Set up optimizer
        optimizer = AdamW(self.model.parameters(), lr=learning_rate)

        # Training loop
        logger.info(f"Starting fine-tuning on {len(train_dataset)} samples...")
        for epoch in range(epochs):
            self.model.train()
            total_loss = 0
```

```
            for batch in train_loader:
                optimizer.zero_grad()

                inputs = {k: v.to(self.device) for k, v in batch.items() if k !=
'labels'}
                labels = batch['labels'].to(self.device) if 'labels' in batch else
None

                outputs = self.model(**inputs, labels=labels)
                loss = outputs.loss

                loss.backward()
                optimizer.step()

                total_loss += loss.item()

            # Validation
            val_loss = self._validate(val_loader)

            logger.info(f"Epoch {epoch+1}/{epochs} - Train loss:
{total_loss/len(train_loader):.4f} - Val loss: {val_loss:.4f}")

        logger.info("Fine-tuning complete")

    def _prepare_fp_samples(self, fp_samples):
        """Convert false positive samples to training format"""
        prepared_samples = []

        for sample in fp_samples:
            if sample["category"] == "domain":
                prepared_samples.append({
                    "text": sample["text"],
                    "domain": sample["correct_label"],
                    "domain_id":
self.taxonomy.get_domain_id(sample["correct_label"])
                })

        return prepared_samples

    def _validate(self, val_loader):
        """Run validation and return validation loss"""
        self.model.eval()
        val_loss = 0
```

```python
        with torch.no_grad():
            for batch in val_loader:
                inputs = {k: v.to(self.device) for k, v in batch.items() if k != 'labels'}
                labels = batch['labels'].to(self.device) if 'labels' in batch else None

                outputs = self.model(**inputs, labels=labels)
                loss = outputs.loss

                val_loss += loss.item()

        return val_loss / len(val_loader)

    def destruct(self, text):
        """Break down input text into semantic components based on the domain taxonomy"""
        # Tokenize input
        inputs = self.tokenizer(text, return_tensors="pt", padding=True, truncation=True, max_length=512)
        inputs = {k: v.to(self.device) for k, v in inputs.items()}

        # Process text
        with torch.no_grad():
            outputs = self.model(**inputs)

        # Extract results from components
        results = {}
        for name, component in self.components.items():
            results[name] = component.extract(text, inputs, outputs)

        # Format final result
        destruction_result = SemanticDestructionResult(text, results, self.taxonomy)
        return destruction_result

    def provide_feedback(self, result, category, predicted, correct):
        """
        Provide feedback on a prediction to improve model

        Args:
            result: A SemanticDestructionResult object
            category: Type of prediction ('domain', 'entity', 'concept', 'relation')
            predicted: What the model predicted
            correct: The correct classification
        """
```

```
        # Extract confidence if available
        confidence = None

        if category == "domain":
            for domain in result.components["domain_classifier"]:
                if domain["domain"] == predicted:
                    confidence = domain["confidence"]
                    break

        # Record the false positive
        self.fp_tracker.record_false_positive(
            category=category,
            predicted=predicted,
            correct=correct,
            text_sample=result.original_text,
            confidence=confidence
        )

        # Generate learning samples after accumulating data
        if sum(len(fps) for fps in self.fp_tracker.false_positives.values()) % 10 ==
0:
            self.fp_tracker.generate_learning_samples()

    def export_false_positives(self, file_path):
        """Export false positive data"""
        self.fp_tracker.export_feedback(file_path)

    def import_false_positives(self, file_path):
        """Import false positive data"""
        return self.fp_tracker.import_feedback(file_path)

    def save(self, path):
        """Save the model and tokenizer"""
        torch.save({
            'model_state_dict': self.model.state_dict(),
            'model_name': self.model_name
        }, path)
        logger.info(f"Model saved to {path}")

    @classmethod
    def load(cls, path, device=None):
        """Load the model from a saved file"""
        checkpoint = torch.load(path, map_location=device)
        model_name = checkpoint['model_name']
```

```
        destructor = cls(model_name=model_name, device=device)
        destructor.model.load_state_dict(checkpoint['model_state_dict'])

        logger.info(f"Model loaded from {path}")
        return destructor


class SemanticDestructorModel(nn.Module):
   """Neural network model for domain-specific semantic destruction tasks"""
   def __init__(self, model_name, num_domains):
        super(SemanticDestructorModel, self).__init__()
        self.base_model = BertModel.from_pretrained(model_name)
        self.hidden_size = self.base_model.config.hidden_size
        self.num_domains = num_domains

        # Task-specific layers
        self.domain_classifier = nn.Linear(self.hidden_size, num_domains)
        self.entity_extractor = nn.Linear(self.hidden_size, 9)  # Entity types
        self.concept_analyzer = nn.Linear(self.hidden_size, 128)  # Concept space
        self.relation_classifier = nn.Linear(self.hidden_size * 2, 16)  # Relation
types

        # Additional layers for domain-specific detection
        self.regulatory_detector = nn.Linear(self.hidden_size, 2)
        self.organizational_detector = nn.Linear(self.hidden_size, 3)
        self.individual_detector = nn.Linear(self.hidden_size, 5)
        self.information_detector = nn.Linear(self.hidden_size, 3)

   def forward(self, input_ids, attention_mask=None, token_type_ids=None,
labels=None):
        # Get base model outputs
        base_outputs = self.base_model(
            input_ids=input_ids,
            attention_mask=attention_mask,
            token_type_ids=token_type_ids
        )

        sequence_output = base_outputs.last_hidden_state  # [batch_size, seq_len,
hidden_size]
        pooled_output = base_outputs.pooler_output  # [batch_size, hidden_size]

        # Domain classification
        domain_logits = self.domain_classifier(pooled_output)  # [batch_size,
num_domains]
```

```
        # Entity extraction
        entity_logits = self.entity_extractor(sequence_output)  # [batch_size, seq_len, 9]

        # Concept analysis
        concept_embeddings = self.concept_analyzer(pooled_output)  # [batch_size, 128]

        # Mock relation classification (would use entity pairs in practice)
        batch_size = pooled_output.size(0)
        relation_inputs = torch.cat([pooled_output, pooled_output], dim=-1)  # [batch_size, 2*hidden_size]
        relation_logits = self.relation_classifier(relation_inputs)  # [batch_size, 16]

        # Domain-specific detectors
        regulatory_logits = self.regulatory_detector(pooled_output)  # [batch_size, 2]
        org_logits = self.organizational_detector(pooled_output)  # [batch_size, 3]
        individual_logits = self.individual_detector(pooled_output)  # [batch_size, 5]
        info_logits = self.information_detector(pooled_output)  # [batch_size, 3]

        # Calculate loss if labels provided
        loss = None
        if labels is not None:
            loss_fct = nn.CrossEntropyLoss()
            loss = loss_fct(domain_logits, labels)

        # Bundle outputs
        outputs = base_outputs

        # Add custom outputs
        outputs.domain_logits = domain_logits
        outputs.entity_logits = entity_logits
        outputs.concept_embeddings = concept_embeddings
        outputs.relation_logits = relation_logits
        outputs.regulatory_logits = regulatory_logits
        outputs.org_logits = org_logits
        outputs.individual_logits = individual_logits
        outputs.info_logits = info_logits
        outputs.loss = loss

        return outputs
```

```
class DomainClassifier:
   """Classifies text into the domain taxonomy"""
   def __init__(self, model, taxonomy):
       self.model = model
       self.taxonomy = taxonomy

   def extract(self, text, inputs, outputs):
       # Get domain logits
       domain_logits = outputs.domain_logits  # [batch_size, num_domains]
       domain_probs = torch.softmax(domain_logits, dim=-1).cpu().numpy()[0]

       # Get top domains
       top_domain_ids = np.argsort(-domain_probs)[:5]  # Top 5 domains

       # Format results
       domain_results = []
       for domain_id in top_domain_ids:
           domain_name = self.taxonomy.get_domain_name(domain_id)
           confidence = float(domain_probs[domain_id])

           # Skip low confidence predictions
           if confidence < 0.05:
               continue

           domain_type = "major" if self.taxonomy.is_major_domain(domain_name) else
"minor"
           major_domain = domain_name if domain_type == "major" else
self.taxonomy.get_major_for_minor(domain_name)

           domain_results.append({
               "domain": domain_name,
               "type": domain_type,
               "major_domain": major_domain,
               "confidence": round(confidence, 4)
           })

       return domain_results


class EntityExtractor:
   """Extracts domain-specific entities from text"""
   def __init__(self, model):
       self.model = model
       self.entity_types = [
```

```
            "O",  # Outside any entity
            "B-ORG", "I-ORG",  # Organization
            "B-PERS", "I-PERS",  # Person
            "B-REG", "I-REG",  # Regulation
            "B-TECH", "I-TECH"  # Technology
        ]

    def extract(self, text, inputs, outputs):
        # Get entity logits
        entity_logits = outputs.entity_logits  # [batch_size, seq_len, 9]
        entity_preds = torch.argmax(entity_logits, dim=-1).cpu().numpy()[0]

        # Tokenize text for alignment
        tokens = text.split()  # Simplified tokenization

        # Extract entities
        entities = []
        current_entity = None

        # Map predictions to tokens (simplified)
        for i, token_pred in enumerate(entity_preds[:len(tokens)]):
            tag = self.entity_types[token_pred]

            if tag.startswith("B-"):
                if current_entity:
                    entities.append(current_entity)
                entity_type = tag[2:]
                current_entity = {"type": entity_type, "text": tokens[i], "start":
i}
            elif tag.startswith("I-") and current_entity and current_entity["type"]
== tag[2:]:
                current_entity["text"] += " " + tokens[i]
            elif tag == "O":
                if current_entity:
                    entities.append(current_entity)
                    current_entity = None

        if current_entity:
            entities.append(current_entity)

        return entities


class ConceptAnalyzer:
    """Analyzes domain-specific concepts"""
```

```
    def __init__(self, model, taxonomy):
        self.model = model
        self.taxonomy = taxonomy

        # Domain-specific concepts
        self.domain_concepts = {
            "regulatory": ["compliance", "legislation", "policy", "authority",
"governance"],
            "domain": ["politics", "economy", "society", "technology", "law"],
            "strategic": ["misinformation", "disinformation", "tactics",
"objectives"],
            "organisational_culture": ["efficiency", "streamlining", "values",
"norms"],
            "organisation": ["resources", "creativity", "management", "adaptation"],
            "organisation_strategy": ["platforms", "marketing", "communication",
"evidence"],
            "individual": ["age", "gender", "traits", "confidence", "perception",
"attitudes"],
            "information_source": ["media", "channels", "hazard", "spread",
"virality"]
        }

    def extract(self, text, inputs, outputs):
        # Get concept embeddings
        concept_embeddings = outputs.concept_embeddings.cpu().numpy()[0]

        # Extract domain-specific concepts (simplified)
        results = {}

        # Extract concepts for top domains
        domain_logits = outputs.domain_logits
        domain_probs = torch.softmax(domain_logits, dim=-1).cpu().numpy()[0]
        top_domain_ids = np.argsort(-domain_probs)[:3]  # Top 3 domains

        for domain_id in top_domain_ids:
            domain_name = self.taxonomy.get_domain_name(domain_id)
            if domain_name in self.domain_concepts:
                # Check for concept mentions in text
                domain_concepts = []
                for concept in self.domain_concepts[domain_name]:
                    if concept.lower() in text.lower():
                        # In reality, would use more sophisticated detection
                        salience = np.random.uniform(0.6, 0.9)  # Mock salience
                        domain_concepts.append({
                            "name": concept,
```

```python
                            "salience": round(float(salience), 4)
                        })

                if domain_concepts:
                    results[domain_name] = domain_concepts

        return results


class RelationMapper:
    """Maps relations between entities and concepts across domains"""
    def __init__(self, model, taxonomy):
        self.model = model
        self.taxonomy = taxonomy

        # Relation types
        self.relation_types = [
            "affects", "regulates", "part_of", "implements",
            "influences", "composed_of", "depends_on", "enables",
            "contradicts", "reinforces", "reduces", "increases",
            "prohibits", "evaluates", "measures", "categorizes"
        ]

    def extract(self, text, inputs, outputs):
        # Get relation logits
        relation_logits = outputs.relation_logits
        relation_probs = torch.softmax(relation_logits, dim=-1).cpu().numpy()[0]

        # Get top relations
        top_relation_ids = np.argsort(-relation_probs)[:3]  # Top 3 relations

        # Mock relations between domains
        relations = []

        # Extract domain information
        domain_logits = outputs.domain_logits
        domain_probs = torch.softmax(domain_logits, dim=-1).cpu().numpy()[0]
        top_domain_ids = np.argsort(-domain_probs)[:5]  # Top 5 domains
        top_domains = [self.taxonomy.get_domain_name(idx) for idx in top_domain_ids]

        # Create a few relations between top domains
        if len(top_domains) >= 2:
            for i in range(min(3, len(top_relation_ids))):
                relation_type = self.relation_types[top_relation_ids[i]]
                confidence = float(relation_probs[top_relation_ids[i]])
```

```python
                # Only include meaningful relations
                if confidence > 0.2:
                    # Choose source and target domain
                    source_idx = i % len(top_domains)
                    target_idx = (i + 1) % len(top_domains)

                    relations.append({
                        "source": top_domains[source_idx],
                        "target": top_domains[target_idx],
                        "relation": relation_type,
                        "confidence": round(confidence, 4)
                    })

        return relations


class DomainDataset(Dataset):
    """Dataset for domain-specific fine-tuning"""
    def __init__(self, corpus_path, tokenizer, taxonomy, max_length=512):
        self.tokenizer = tokenizer
        self.taxonomy = taxonomy
        self.max_length = max_length

        # Load domain corpus
        try:
            self.data = pd.read_csv(corpus_path)
        except:
            # Mock data if file doesn't exist
            logger.warning(f"Couldn't load corpus from {corpus_path}. Using mock data.")
            self.data = self._create_mock_data()

    def add_samples(self, samples):
        """Add additional samples to the dataset"""
        if not samples:
            return

        # Convert to DataFrame
        new_samples = pd.DataFrame(samples)

        # Ensure domain_id exists
        if "domain_id" not in new_samples.columns:
            new_samples["domain_id"] = new_samples["domain"].apply(
                lambda x: self.taxonomy.get_domain_id(x)
```

```
            )

        # Append to existing data
        self.data = pd.concat([self.data, new_samples], ignore_index=True)
        logger.info(f"Added {len(samples)} samples to dataset. New size:
{len(self.data)}")

    def _create_mock_data(self):
        """Create mock training data"""
        mock_samples = [
            # Regulatory domain examples
            {
                "text": "New regulatory frameworks are essential for ensuring
legitimacy in emerging markets.",
                "domain": "regulatory"
            },
            {
                "text": "The legitimacy of the regulatory body has been questioned
by industry leaders.",
                "domain": "legitimacy"
            },
            # Domain examples
            {
                "text": "Political and economic factors influence technological
adaptation in regulated industries.",
                "domain": "domain"
            },
            {
                "text": "Legal frameworks must account for technological innovation
while protecting social culture.",
                "domain": "legal"
            },
            # Strategic examples
            {
                "text": "The spread of fake news requires strategic countermeasures
from media organizations.",
                "domain": "strategic"
            },
            {
                "text": "New strategies are needed to combat disinformation in the
digital landscape.",
                "domain": "strategies"
            },
            # Organizational culture examples
            {
```

```
                "text": "Optimisation of workflows has transformed the
organizational culture.",
                "domain": "optimisation"
            },
            {
                "text": "Cost reduction initiatives have impacted employee morale
within the organization.",
                "domain": "reduction"
            },
            # Organization examples
            {
                "text": "Innovation capabilities are essential for organizational
survival in disruptive markets.",
                "domain": "innovation"
            },
            {
                "text": "Leadership plays a critical role in developing
organizational capabilities.",
                "domain": "leadership"
            },
            # Organization strategy examples
            {
                "text": "Social media campaigns must be backed by verifiable facts
to maintain credibility.",
                "domain": "organisation_strategy"
            },
            {
                "text": "Factual accuracy in social media advertising is
increasingly scrutinized by regulators.",
                "domain": "facts"
            },
            # Individual examples
            {
                "text": "Demographic factors and personality traits influence trust
in information sources.",
                "domain": "individual"
            },
            {
                "text": "Self-efficacy beliefs impact how individuals evaluate and
share information online.",
                "domain": "self_efficacy"
            },
            # Information source examples
            {
```

```
                "text": "The information ecosystem faces increasing risks from
automated propagation of falsehoods.",
                "domain": "information_source"
            },
            {
                "text": "Understanding risk factors in information propagation is
crucial for platform governance.",
                "domain": "risk"
            }
        ]

        # Add examples with potential for false positives
        false_positive_prone_samples = [
            # Examples that might be misclassified between domains
            {
                "text": "Corporate leadership must consider regulatory frameworks
when implementing new policies.",
                "domain": "leadership"  # Could be confused with regulatory
            },
            {
                "text": "Innovation in policy implementation helps organizations
comply with complex regulations.",
                "domain": "innovation"  # Could be confused with regulatory
            },
            {
                "text": "Trust in regulatory bodies influences individual compliance
with legal requirements.",
                "domain": "trust"  # Could be confused with regulatory
            },
            {
                "text": "Social media platforms develop strategies to combat fake
news while maintaining user trust.",
                "domain": "social_media"  # Could be confused with strategic
            },
            {
                "text": "Information ecosystems are shaped by technological
innovation and legal frameworks.",
                "domain": "information_ecosystem"  # Could be confused with
technological
            },
            {
                "text": "Political discourse on digital platforms affects the
propagation of factual information.",
                "domain": "political"  # Could be confused with information_source
            },
```

```
            {
                "text": "Economic factors influence organizational strategies for
information management.",
                "domain": "economic"  # Could be confused with organisation
            },
            {
                "text": "Self-efficacy in leadership positions correlates with
organizational innovation capabilities.",
                "domain": "self_efficacy"  # Could be confused with leadership
            },
            {
                "text": "Legal considerations in information sharing affect
strategic communication plans.",
                "domain": "legal"  # Could be confused with information_source
            },
            {
                "text": "Digital platforms must optimize information delivery while
maintaining factual accuracy.",
                "domain": "facts"  # Could be confused with optimisation
            },
            {
                "text": "Demographic analysis of user engagement helps shape social
media strategies.",
                "domain": "demographic"  # Could be confused with social_media
            }
        ]

        # Combine all examples
        mock_samples.extend(false_positive_prone_samples)

        # Convert to DataFrame
        df = pd.DataFrame(mock_samples)

        # Add domain labels
        df["domain_id"] = df["domain"].apply(lambda x:
self.taxonomy.get_domain_id(x))

        return df

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        text = self.data.iloc[idx]["text"]
        domain_id = self.data.iloc[idx]["domain_id"]
```

```python
        # Tokenize
        encoding = self.tokenizer(
            text,
            max_length=self.max_length,
            padding="max_length",
            truncation=True,
            return_tensors="pt"
        )

        # Remove batch dimension
        encoding = {k: v.squeeze(0) for k, v in encoding.items()}

        # Add label
        encoding["labels"] = torch.tensor(domain_id, dtype=torch.long)

        return encoding


class SemanticDestructionResult:
    """Container for domain-specific semantic destruction results"""
    def __init__(self, original_text, components, taxonomy):
        self.original_text = original_text
        self.components = components
        self.taxonomy = taxonomy

    def to_dict(self):
        """Convert results to dictionary"""
        return {
            "original_text": self.original_text,
            "domains": self.components["domain_classifier"],
            "entities": self.components["entity_extractor"],
            "concepts": self.components["concept_analyzer"],
            "relations": self.components["relation_mapper"]
        }

    def to_json(self, indent=2):
        """Convert results to JSON string"""
        return json.dumps(self.to_dict(), indent=indent)

    def get_primary_domain(self):
        """Get the primary domain of the text"""
        if not self.components["domain_classifier"]:
            return None
        return self.components["domain_classifier"][0]["domain"]
```

```python
    def get_domain_hierarchy(self):
        """Get the domain hierarchy of the text"""
        domains = self.components["domain_classifier"]

        hierarchy = {}
        for domain in domains:
            if domain["type"] == "major":
                major = domain["domain"]
                if major not in hierarchy:
                    hierarchy[major] = []
            else:
                major = domain["major_domain"]
                if major not in hierarchy:
                    hierarchy[major] = []
                hierarchy[major].append({
                    "minor": domain["domain"],
                    "confidence": domain["confidence"]
                })

        return hierarchy

    def __str__(self):
        primary_domain = self.get_primary_domain() if hasattr(self, 'get_primary_domain') else 'Unknown'
        entities = self.components.get("entity_extractor", [])
        return f"SemanticDestructionResult: Primary domain '{primary_domain}' with {len(entities)} entities"


class FalsePositiveSample:
    """Container for a sample with known false positive potential"""
    def __init__(self, text, correct_domain, likely_misclassification=None, difficulty="medium"):
        self.text = text
        self.correct_domain = correct_domain
        self.likely_misclassification = likely_misclassification
        self.difficulty = difficulty  # 'easy', 'medium', 'hard'

    def to_dict(self):
        return {
            "text": self.text,
            "correct_domain": self.correct_domain,
            "likely_misclassification": self.likely_misclassification,
            "difficulty": self.difficulty
```

```python
        }



# Sample creation for false positive learning
def create_false_positive_samples():
    """Create samples explicitly designed to test false positive detection"""
    return [
        # Border cases between regulatory and organizational domains
        FalsePositiveSample(
            text="The company implemented new governance policies following
regulatory changes.",
            correct_domain="organisation",
            likely_misclassification="regulatory",
            difficulty="hard"
        ),
        FalsePositiveSample(
            text="Leadership decisions must balance compliance requirements with
operational efficiency.",
            correct_domain="leadership",
            likely_misclassification="regulatory",
            difficulty="hard"
        ),

        # Border cases between strategic and information source domains
        FalsePositiveSample(
            text="Media channels strategically propagate information to targeted
demographics.",
            correct_domain="information_source",
            likely_misclassification="strategic",
            difficulty="medium"
        ),
        FalsePositiveSample(
            text="The spread of misinformation requires strategic countermeasures
from platforms.",
            correct_domain="strategic",
            likely_misclassification="information_source",
            difficulty="medium"
        ),

        # Border cases between individual and organizational culture
        FalsePositiveSample(
            text="Employee self-efficacy contributes to organizational optimization
efforts.",
            correct_domain="self_efficacy",
            likely_misclassification="optimisation",
```

```
            difficulty="hard"
        ),
        FalsePositiveSample(
            text="Cost reduction initiatives must consider impacts on employee trust and confidence.",
            correct_domain="reduction",
            likely_misclassification="trust",
            difficulty="medium"
        ),

        # Ambiguous cases with multiple domain indicators
        FalsePositiveSample(
            text="Legal frameworks for social media companies must balance innovation with protection of individual beliefs.",
            correct_domain="legal",
            likely_misclassification="social_media",
            difficulty="hard"
        ),
        FalsePositiveSample(
            text="Political discourse on technological platforms shapes public trust in information sources.",
            correct_domain="political",
            likely_misclassification="information_ecosystem",
            difficulty="hard"
        ),

        # Clear cases that should be easy to classify
        FalsePositiveSample(
            text="Regulatory frameworks provide legitimacy to new market entrants.",
            correct_domain="regulatory",
            likely_misclassification=None,
            difficulty="easy"
        ),
        FalsePositiveSample(
            text="Personality traits influence how individuals process and share news content.",
            correct_domain="personality",
            likely_misclassification=None,
            difficulty="easy"
        )
    ]


# Usage example with false positive tracking
def main():
```

```python
    # Initialize the domain-specific semantic destructor
    destructor = SemanticDestructor()

    # Create false positive test samples
    fp_samples = create_false_positive_samples()

    # Fine-tune on domain data
    destructor.fine_tune("domain_corpus.csv")

    # Process test samples and check for false positives
    print("\nTesting with false positive samples:")
    for idx, sample in enumerate(fp_samples):
        print(f"\nSample {idx+1}: {sample.text}")
        print(f"Expected domain: {sample.correct_domain}")

        # Process text
        result = destructor.destruct(sample.text)
        primary_domain = result.get_primary_domain()

        print(f"Predicted primary domain: {primary_domain}")

        # Check if prediction is correct
        if primary_domain != sample.correct_domain:
            print(f"FALSE POSITIVE DETECTED! Recording feedback...")
            # Record the false positive
            destructor.provide_feedback(
                result=result,
                category="domain",
                predicted=primary_domain,
                correct=sample.correct_domain
            )

    # Get false positive statistics
    fp_stats = destructor.fp_tracker.get_false_positive_stats()
    print("\nFalse Positive Statistics:")
    print(json.dumps(fp_stats, indent=2))

    # Generate learning samples
    learning_samples = destructor.fp_tracker.generate_learning_samples()
    print(f"\nGenerated {len(learning_samples)} learning samples from false
positives")

    # Export false positive data
    destructor.export_false_positives("false_positives.json")
```

```
    # Fine-tune again with false positive samples
    print("\nFine-tuning with false positive samples...")
    destructor.fine_tune("domain_corpus.csv", include_fp_samples=True)

    # Test again after fine-tuning
    print("\nRe-testing with false positive samples after fine-tuning:")

    improved_count = 0
    for idx, sample in enumerate(fp_samples):
        print(f"\nSample {idx+1}: {sample.text}")
        print(f"Expected domain: {sample.correct_domain}")

        # Process text
        result = destructor.destruct(sample.text)
        primary_domain = result.get_primary_domain()

        print(f"Predicted primary domain: {primary_domain}")

        # Check if prediction is correct
        if primary_domain == sample.correct_domain:
            improved_count += 1

    print(f"\nImprovement after false positive learning:
{improved_count}/{len(fp_samples)} samples correctly classified")

    # Example texts for different domains
    sample_texts = {
        "regulatory": """
            New regulatory frameworks must balance legitimacy concerns with
practical implementation.
            Regulatory bodies need clear guidelines to enforce compliance while
maintaining transparency.
        """,

        "organisation": """
            Organizational capabilities are enhanced through leadership that fosters
innovation.
            Companies with strong innovation capabilities typically outperform their
peers in
            rapidly changing markets.
        """,

        "individual": """
            Demographic factors and personality traits influence how individuals
process information.
```

```
            Self-efficacy beliefs shape trust in various information sources, especially regarding
            complex or controversial topics.
        """,

        "information_source": """
            The information ecosystem faces increasing risks from automated propagation techniques.
            Understanding how misinformation spreads through various channels is essential for
            developing effective countermeasures.
        """
    }

    # Process each sample text
    for domain, text in sample_texts.items():
        print(f"\nProcessing {domain} text:")
        result = destructor.destruct(text)

        # Print detailed results
        print("Domain hierarchy:")
        hierarchy = result.get_domain_hierarchy()
        for major, minors in hierarchy.items():
            print(f"  {major}")
            for minor in minors:
                print(f"    - {minor['minor']} ({minor['confidence']:.4f})")

        print("\nEntities:")
        for entity in result.components["entity_extractor"]:
            print(f"  - {entity['text']} ({entity['type']})")

        print("\n" + "-" * 50)

    # Save model
    destructor.save("domain_semantic_destructor_with_fp.pt")


# Extended example with additional edge cases for testing false positive detection
def extended_test():
    """Test the system with additional challenging examples"""
    destructor = SemanticDestructor()

    # Load a previously trained model if available
    try:
```

```
        destructor =
SemanticDestructor.load("domain_semantic_destructor_with_fp.pt")
        print("Loaded existing model")
    except:
        print("Training new model")
        destructor.fine_tune("domain_corpus.csv")

    # Additional edge cases
    edge_cases = [
        {
            "text": "Media organizations must implement governance frameworks that
ensure factual accuracy while maintaining audience engagement.",
            "expected": "information_source",
            "notes": "Combines media (information source) with governance
(regulatory)"
        },
        {
            "text": "Personal beliefs can impact organizational culture when
leadership fails to establish clear values.",
            "expected": "beliefs",
            "notes": "Combines individual domain (beliefs) with organizational
culture"
        },
        {
            "text": "Technology companies face regulatory scrutiny over how their
innovation strategies impact user privacy.",
            "expected": "technological",
            "notes": "Combines technological domain with regulatory and innovation"
        },
        {
            "text": "Marketing campaigns that leverage demographic data must
maintain trust while optimizing engagement.",
            "expected": "organisation_strategy",
            "notes": "Combines organization strategy with individual (demographic,
trust)"
        },
        {
            "text": "Social media platforms must develop strategies to reduce
misinformation propagation through network effects.",
            "expected": "social_media",
            "notes": "Combines social media with information propagation"
        }
    ]

    # Test edge cases
```

```python
    print("\n=== Testing Edge Cases ===")
    for case in edge_cases:
        print(f"\nText: {case['text']}")
        print(f"Expected domain: {case['expected']}")
        print(f"Notes: {case['notes']}")

        result = destructor.destruct(case['text'])
        primary_domain = result.get_primary_domain()

        print(f"Predicted domain: {primary_domain}")

        # Show top three domains with confidence
        top_domains = result.components["domain_classifier"][:3]
        print("Top domain predictions:")
        for domain in top_domains:
            print(f"  - {domain['domain']} ({domain['confidence']:.4f})")

        # Record feedback if incorrect
        if primary_domain != case['expected']:
            print("Recording false positive...")
            destructor.provide_feedback(
                result=result,
                category="domain",
                predicted=primary_domain,
                correct=case['expected']
            )

    # Export false positives
    destructor.export_false_positives("edge_case_false_positives.json")

    # Analyze the edge case false positives
    fp_stats = destructor.fp_tracker.get_false_positive_stats()
    print("\nEdge Case False Positive Statistics:")
    print(json.dumps(fp_stats, indent=2))


if __name__ == "__main__":
    main()
    print("\n=== Running Extended Test ===")
    extended_test()
```