النموذج اللغوي المبتكر في نظام بصيرة

نظرة عامة

النموذج اللغوي المبتكر هو مكون أساسي في نظام بصيرة يهدف إلى تحليل وفهم اللغة العربية بطريقة فريدة ومبتكرة، مع التركيز على المعاني العميقة للحروف والكلمات والتراكيب. يعتمد هذا النموذج على المعادلات الرياضية التكيفية والنواة الرياضية لنظام بصيرة بدلاً من الشبكات العصبية التقليدية، مما يوفر نهجًا جديدًا لمعالجة اللغة الطبيعية.

الهيكل العام

يتكون النموذج اللغوي المبتكر من المكونات الرئيسية التالية:

1. **محرك دلالات الحروف (LetterSemanticsEngine)**: يحلل المعاني الدلالية والفلسفية والرياضية للحروف العربية.
2. **محرك التحليل الصرفي (MorphologicalAnalysisEngine)**: يحلل بنية الكلمات وأنماطها الصرفية.
3. **محرك التحليل النحوي (SyntacticAnalysisEngine)**: يحلل بنية الجمل والعلاقات النحوية بين الكلمات.
4. **محرك التحليل البلاغي (RhetoricalAnalysisEngine)**: يحلل الأساليب البلاغية والتعبيرات المجازية في النص.
5. **محرك التكامل (IntegrationEngine)**: يدمج نتائج التحليلات المختلفة لتقديم فهم شامل للنص.

المكونات التفصيلية

محرك دلالات الحروف (LetterSemanticsEngine)

هذا المحرك هو قلب النموذج اللغوي المبتكر. يقوم بتحليل كل حرف في النص بناءً على قاعدة بيانات شاملة لدلالات الحروف العربية، والتي تتضمن:

- **الخصائص الصوتية**: مخارج الحروف وصفاتها.
- **دلالات الشكل البصري**: المعاني المرتبطة بشكل الحرف.
- **المحاور الدلالية الأساسية**: المعاني الجوهرية للحرف.
- **الأبعاد الفلسفية**: الارتباطات الفلسفية والروحية.
- **التمثيلات الرياضية**: معادلات رمزية تمثل خصائص الحرف.

```
class LetterSemanticsEngine:
  def __init__(self):
```

```
        self.letter_semantics_db = self._load_letter_semantics_database()
        self.symbolic_engine = AdvancedSymbolicExpression("x") # Placeholder

    def _load_letter_semantics_database(self):
        # Load from symbolic_processing.data.letter_semantics_database
        from symbolic_processing.data.letter_semantics_database import
LETTER_SEMANTICS_DB
        return LETTER_SEMANTICS_DB

    def analyze_letter(self, letter):
        """تحليل دلالات حرف واحد."""
        return self.letter_semantics_db.get(letter, {"error": "Letter not found"})

    def analyze_text(self, text):
        """تحليل دلالات الحروف في نص كامل."""
        analysis = {}
        for char in text:
            if char.isalpha() and char in self.letter_semantics_db:
                analysis[char] = self.analyze_letter(char)
        return analysis

    def get_mathematical_representation(self, letter):
        """الحصول على التمثيل الرياضي للحرف."""
        semantics = self.analyze_letter(letter)
        if "mathematical_representation" in semantics:
            try:
                # Assuming representation is a string that can be parsed
                return
AdvancedSymbolicExpression(semantics["mathematical_representation"])
            except Exception as e:
                print(f"Error creating symbolic expression for letter {letter}: {e}")
                return None
        return None
```

محرك التحليل الصرفي (MorphologicalAnalysisEngine)

يقوم هذا المحرك بتحليل بنية الكلمات وتحديد جذورها وأنماطها الصرفية (الأوزان) والزوائد (السوابق واللواحق). يستخدم قاعدة بيانات للأنماط الصرفية وقواعد التحويل.

```
class MorphologicalAnalysisEngine:
    def __init__(self):
        self.patterns_db = self._load_patterns_database()
        self.roots_db = self._load_roots_database() # Assumed DB
        self.affixes_db = self._load_affixes_database() # Assumed DB

    def _load_patterns_database(self):
        # Load from symbolic_processing.data.morphological_patterns_database
        from symbolic_processing.data.morphological_patterns_database import
MORPHOLOGICAL_PATTERNS_DB
```

```
        return MORPHOLOGICAL_PATTERNS_DB

    def _load_roots_database(self):
        # Placeholder for loading roots database
        return {"كتب": {"meaning": "writing"}, "علم": {"meaning": "knowledge"}}

    def _load_affixes_database(self):
        # Placeholder for loading affixes database
        return {"ال": {"type": "prefix", "meaning": "definite article"}, "ون": {"type":
"suffix", "meaning": "plural masculine"}}

    def analyze_word(self, word):
        """تحليل كلمة واحدة صرفيًا."""
        analysis = {
            "word": word,
            "root": None,
            "pattern": None,
            "prefixes": [],
            "suffixes": [],
            "meaning_components": []
        }

        # Basic prefix/suffix stripping (example)
        processed_word = word
        for prefix, data in self.affixes_db.items():
            if data["type"] == "prefix" and processed_word.startswith(prefix):
                analysis["prefixes"].append(prefix)
                processed_word = processed_word[len(prefix):]
                analysis["meaning_components"].append(data["meaning"])

        for suffix, data in self.affixes_db.items():
            if data["type"] == "suffix" and processed_word.endswith(suffix):
                analysis["suffixes"].append(suffix)
                processed_word = processed_word[:-len(suffix)]
                analysis["meaning_components"].append(data["meaning"])

        # Root and pattern matching (simplified example)
        # This needs a much more sophisticated algorithm
        possible_root = self._guess_root(processed_word)
        if possible_root in self.roots_db:
            analysis["root"] = possible_root
            analysis["meaning_components"].append(self.roots_db[possible_root]
["meaning"])
            analysis["pattern"] = self._match_pattern(processed_word, possible_root)
            if analysis["pattern"] in self.patterns_db:

analysis["meaning_components"].append(self.patterns_db[analysis["pattern"]]
["meaning"])

        return analysis
```

```
    def _guess_root(self, word_part):
        # Highly simplified root guessing - needs proper linguistic rules
        if len(word_part) == 3:
            return word_part # Assume 3-letter roots for simplicity
        # Add more complex logic here
        if word_part == "كاتب": return "كتب"
        if word_part == "معلوم": return "علم"
        return None

    def _match_pattern(self, word_part, root):
        # Simplified pattern matching
        if root == "كتب" and word_part == "كاتب": return "فاعل"
        if root == "علم" and word_part == "معلوم": return "مفعول"
        # Add more patterns
        return None

    def analyze_text(self, text):
        """تحليل النص صرفيًا كلمة بكلمة."""
        words = text.split()
        return [self.analyze_word(word) for word in words]
```

محرك التحليل النحوي (SyntacticAnalysisEngine)

يحلل هذا المحرك بنية الجمل والعلاقات النحوية بين الكلمات. يقوم بتحديد أنواع الكلمات (اسم، فعل، حرف)، الأدوار النحوية (فاعل، مفعول به، مبتدأ، خبر)، وأنواع الجمل (اسمية، فعلية).

```
class SyntacticAnalysisEngine:
    def __init__(self):
        # Initialize necessary resources (e.g., grammar rules)
        self.grammar_rules = self._load_grammar_rules()

    def _load_grammar_rules(self):
        # Placeholder for loading grammar rules
        return {
            "S -> NP VP": 0.9,
            "NP -> Det N": 0.7,
            "NP -> N": 0.3,
            "VP -> V NP": 0.6,
            "VP -> V": 0.4,
            # Add more complex Arabic grammar rules
        }

    def analyze_sentence(self, sentence):
        """تحليل جملة واحدة نحويًا."""
        words = sentence.split()
        tagged_words = self._tag_words(words) # Part-of-speech tagging
        parse_tree = self._parse_sentence(tagged_words) # Parsing
```

```
        return {
            "sentence": sentence,
            "tagged_words": tagged_words,
            "parse_tree": parse_tree,
            "sentence_type": self._determine_sentence_type(parse_tree)
        }

    def _tag_words(self, words):
        # Simplified POS tagging - needs a proper Arabic POS tagger
        tags = []
        for word in words:
            if word in ["ذهب", "كتب", "يقرأ"]: tags.append((word, "Verb"))
            elif word in ["الرجل", "الكتاب", "العلم"]: tags.append((word, "Noun"))
            elif word in ["إلى", "في", "على"]: tags.append((word, "Preposition"))
            elif word == "ال": tags.append((word, "Determiner"))
            else: tags.append((word, "Unknown"))
        return tags

    def _parse_sentence(self, tagged_words):
        # Simplified parsing - needs a proper Arabic parser (e.g., CFG, dependency)
        # This example just returns the tagged words as a flat structure
        return {"type": "FlatStructure", "nodes": tagged_words}
        # A real implementation would build a tree based on grammar rules

    def _determine_sentence_type(self, parse_tree):
        # Simplified sentence type determination
        if parse_tree and parse_tree["nodes"]:
            first_word_tag = parse_tree["nodes"][0][1]
            if first_word_tag == "Verb": return "فعلية"
            if first_word_tag == "Noun": return "اسمية"
        return "غير محدد"

    def analyze_text(self, text):
        """تحليل النص نحويًا جملة بجملة."""
        # Basic sentence splitting (needs improvement)
        sentences = text.split(".")
        return [self.analyze_sentence(s.strip()) for s in sentences if s.strip()]
```

محرك التحليل البلاغي (RhetoricalAnalysisEngine)

يقوم هذا المحرك بتحليل الأساليب البلاغية والتعبيرات المجازية في النص، مثل التشبيه، الاستعارة، الكناية، والجناس. يهدف إلى فهم المعاني الأعمق والنوايا الكامنة وراء استخدام هذه الأساليب.

```
class RhetoricalAnalysisEngine:
    def __init__(self):
        self.rhetorical_figures_db = self._load_rhetorical_figures()

    def _load_rhetorical_figures(self):
```

```
        # Placeholder for loading rhetorical figures database with patterns/rules
        return {
            "تشبيه": {"patterns": ["مثل", "كأن", "يشبه"]},
            "استعارة": {"patterns": []}, # Needs more complex detection
            "كناية": {"patterns": []}, # Needs contextual understanding
            "جناس": {"patterns": []} # Needs phonetic/morphological analysis
        }

    def analyze_sentence(self, sentence, syntactic_analysis=None):
        """تحليل جملة واحدة بلاغيًا."""
        figures = []

        # Detect Tashbih (Simile)
        for keyword in self.rhetorical_figures_db["تشبيه"]["patterns"]:
            if keyword in sentence:
                figures.append({
                    "type": "تشبيه",
                    "keyword": keyword,
                    "position": sentence.find(keyword)
                })

        # Add detection logic for other figures (Istiaara, Kinaya, Jinas)
        # This requires more advanced NLP techniques

        return {
            "sentence": sentence,
            "rhetorical_figures": figures
        }

    def analyze_text(self, text, syntactic_analyses=None):
        """تحليل النص بلاغيًا جملة بجملة."""
        sentences = text.split(".")
        analyses = []
        for i, sentence in enumerate(sentences):
            if sentence.strip():
                syntactic_info = syntactic_analyses[i] if syntactic_analyses and i <
len(syntactic_analyses) else None
                analyses.append(self.analyze_sentence(sentence.strip(), syntactic_info))
        return analyses
```

محرك التكامل (IntegrationEngine)

يدمج هذا المحرك نتائج التحليلات المختلفة (دلالات الحروف، الصرف، النحو، البلاغة) لتقديم فهم شامل ومتكامل للنص. يقوم بربط المعلومات من المستويات المختلفة واستخلاص المعاني العميقة.

```
class IntegrationEngine:
    def __init__(self):
        self.letter_semantics_engine = LetterSemanticsEngine()
```

```
        self.morphological_engine = MorphologicalAnalysisEngine()
        self.syntactic_engine = SyntacticAnalysisEngine()
        self.rhetorical_engine = RhetoricalAnalysisEngine()
        # Integration with mathematical core (SymbolicEngine, ExpertSystem, etc.)
        self.symbolic_engine = AdvancedSymbolicExpression("x") # Placeholder
        self.expert_system = AdvancedExpertSystem() # Placeholder

    def analyze_text_integrated(self, text):
        """إجراء تحليل لغوي متكامل للنص."""
        # 1. Letter Semantics Analysis
        letter_analysis = self.letter_semantics_engine.analyze_text(text)

        # 2. Morphological Analysis
        morphological_analysis = self.morphological_engine.analyze_text(text)

        # 3. Syntactic Analysis
        syntactic_analysis = self.syntactic_engine.analyze_text(text)

        # 4. Rhetorical Analysis
        rhetorical_analysis = self.rhetorical_engine.analyze_text(text,
syntactic_analysis)

        # 5. Integration and Deep Meaning Extraction
        integrated_analysis = self._integrate_analyses(
            text,
            letter_analysis,
            morphological_analysis,
            syntactic_analysis,
            rhetorical_analysis
        )

        # 6. Apply Expert System Rules (Example)
        expert_insights = self.expert_system.apply_rules(integrated_analysis)

        integrated_analysis["expert_insights"] = expert_insights

        return integrated_analysis

    def _integrate_analyses(self, text, letter_analysis, morph_analysis, synt_analysis,
rhet_analysis):
        """دمج نتائج التحليلات المختلفة."""
        integration = {
            "original_text": text,
            "letter_semantics": letter_analysis,
            "morphology": morph_analysis,
            "syntax": synt_analysis,
            "rhetoric": rhet_analysis,
            "overall_meaning_vector": None, # Placeholder for mathematical
representation
            "key_themes": [],
            "sentiment": None # Placeholder
        }
```

```
        # Example Integration Logic:
        # - Combine letter semantics for key words
        # - Link morphological roots to syntactic roles
        # - Analyze rhetorical figures in context of sentence structure
        # - Generate a mathematical vector representing the text's meaning

        # Placeholder for theme extraction
        all_words = [item["word"] for item in morph_analysis]
        from collections import Counter
        word_counts = Counter(all_words)
        integration["key_themes"] = [word for word, count in
word_counts.most_common(3)]

        return integration
```

التكامل مع النواة الرياضية

يتكامل النموذج اللغوي المبتكر بشكل وثيق مع النواة الرياضية لنظام بصيرة:

1. **محرك المعالجة الرمزية (SymbolicEngine)**: يستخدم لتمثيل دلالات الحروف والكلمات والتراكيب كمعادلات رمزية.
2. **نظام الخبير (ExpertSystem)**: يطبق قواعد لغوية وبلاغية لاستخلاص المعاني وتفسير النصوص.
3. **المستكشف التطوري (EvolutionaryExplorer)**: يمكن استخدامه لاكتشاف أنماط لغوية جديدة أو تطور التعبيرات اللغوية.
4. **التكامل الدلالي (SemanticIntegration)**: يربط المفاهيم اللغوية بشبكة دلالية أوسع داخل النظام.

قواعد البيانات اللغوية

يعتمد النموذج على قواعد بيانات متخصصة:

1. **قاعدة بيانات دلالات الحروف (Letter Semantics Database)**: تحتوي على معلومات مفصلة عن كل حرف عربي.
2. **قاعدة بيانات الأنماط الصرفية (Morphological Patterns Database)**: تحتوي على الأوزان الصرفية وقواعدها.
3. **قواعد بيانات الجذور والزوائد (Roots and Affixes Databases)**: (مفترضة) لتحديد مكونات الكلمة.
4. **قواعد النحو (Grammar Rules)**: (مفترضة) لتحليل بنية الجمل.
5. **قاعدة بيانات الأساليب البلاغية (Rhetorical Figures Database)**: (مفترضة) لتحديد الأساليب البلاغية.

الاستخدام

```
from innovative_language_model.integration_engine import IntegrationEngine

# إنشاء محرك التكامل اللغوي
language_engine = IntegrationEngine()

# تحليل نص
text_to_analyze = "العلم نور والجهل ظلام. يرفع العلم بيوتا لا عماد لها."
analysis_result = language_engine.analyze_text_integrated(text_to_analyze)

# عرض بعض النتائج
print("النص الأصلي:", analysis_result["original_text"])
print("\nدلالات بعض الحروف:")
print("ع:", analysis_result["letter_semantics"].get("ع", {}).get("core_meaning", "غير متوفر"))
print("ل:", analysis_result["letter_semantics"].get("ل", {}).get("core_meaning", "غير متوفر"))
print("م:", analysis_result["letter_semantics"].get("م", {}).get("core_meaning", "غير متوفر"))

print("\nالتحليل الصرفي لكلمة 'العلم':")
for item in analysis_result["morphology"]:
    if item["word"] == "العلم":
        print(item)
        break

print("\nالتحليل النحوي للجملة الأولى:")
print(analysis_result["syntax"][0])

print("\nالتحليل البلاغي للجملة الأولى:")
print(analysis_result["rhetoric"][0])

print("\nالمواضيع الرئيسية:", analysis_result["key_themes"])
```

الخلاصة

النموذج اللغوي المبتكر في نظام بصيرة يقدم نهجًا فريدًا وعميقًا لتحليل اللغة العربية، بالاعتماد على دلالات الحروف والتكامل مع النواة الرياضية. يهدف هذا النموذج إلى تجاوز التحليل السطحي للوصول إلى فهم أعمق للمعاني والنوايا الكامنة في النصوص.