Automatyczna Klasyfikacja Tematyczna Polskich Artykułów Newsowych

Maciej Biegan Kamil Dziedzic Jan Bobak

Grudzień 2025

Streszczenie

Niniejsza praca przedstawia system automatycznej klasyfikacji tematycznej polskich artykułów newsowych. Zaimplementowano trzy architektury modeli: HerBERT, BERT oraz MLP z wektoryzacją TF-IDF. System agreguje dane z kanałów RSS polskich portali informacyjnych, przetwarza je metodami NLP i klasyfikuje do 6 kategorii. MLP z architekturą [512,256] osiągnął F1-Score=0.8174 przy czasie treningu 1.09s. Analiza statystyczna 12 konfiguracji wykazała, że learning rate jest jedynym parametrem o istotnym wpływie na wydajność ($p<0.0001$, $r=+0.695$).

1 Wstęp

Automatyczne tagowanie artykułów stanowi fundamentalny problem przetwarzania języka naturalnego w erze nadmiaru informacji. Klasyfikacja tematyczna umożliwia efektywną organizację, wyszukiwanie i rekomendację treści w systemach informacyjnych. Dla języka polskiego, z jego bogatą morfologią i złożoną strukturą gramatyczną, zadanie to przedstawia specyficzne wyzwania wymagające dedykowanych rozwiązań.

1.1 Cel pracy

Celem projektu jest opracowanie systemu do automatycznej klasyfikacji polskich artykułów newsowych do kategorii tematycznych: Polska, Świat, Biznes, Technologie, Moto, Sport. System integruje scraping danych, przetwarzanie NLP, trenowanie modeli oraz interfejs webowy do predykcji.

2 Metody Przetwarzania Języka Naturalnego

2.1 Architektura Przetwarzania

System wykorzystuje pipeline składający się z następujących etapów:

1. **Scraping i agregacja**: Ekstrakcja artykułów z kanałów RSS
2. **Preprocessing**: Normalizacja, lematyzacja, filtracja
3. **Reprezentacja cech**: TF-IDF lub embeddingi kontekstowe

4. **Klasyfikacja**: Modele transformerowe lub sieci MLP

5. **Evaluacja**: Metryki F1-Score, Accuracy, Confusion Matrix

2.2 Preprocessing Tekstu

2.2.1 Normalizacja

Implementacja w `Scrapper/scrapper.py` (linie 128-147):

```
def normalize_text(text: str) -> str:
    if not isinstance(text, str):
        return ''
    text = text.strip()
    text = RE_URL.sub(' ', text)
    text = text.replace('\xa0', ' ')
    text = RE_NON_LETTER.sub(' ', text)
    text = RE_MULTI_WS.sub(' ', text)
    text = text.lower()
    return text
```

Proces usuwa URLs, znaki niealfabetyczne (z wyjątkiem polskich liter diakrytycznych), nadmiarowe białe znaki i konwertuje tekst do małych liter.

2.2.2 Lematyzacja i Tokenizacja

Wykorzystano spaCy z modelem `pl_core_news_sm`:

```
def lemmatize_and_tokenize(text: str, nlp_model,
                           stopwords: Set[str],
                           min_token_len: int = 3):
    text_norm = normalize_text(text)
    doc = nlp_model(text_norm)
    tokens = []
    for tok in doc:
        if tok.is_space or tok.is_punct:
            continue
        lemma = tok.lemma_.lower().strip()
        if len(lemma) < min_token_len or \
           lemma in stopwords or \
           lemma.isdigit():
            continue
        tokens.append(lemma)
    return tokens
```

2.3 Wektoryzacja TF-IDF

Dla modelu MLP zastosowano TF-IDF (Term Frequency-Inverse Document Frequency) z bi-gramami:

```
# models/model.py
vectorizer = TfidfVectorizer(
    max_features=config.max_features,
    ngram_range=(1, 2),
    sublinear_tf=True
)
X = vectorizer.fit_transform(df['text']).toarray()
```

Parametr `sublinear_tf=True` zastępuje TF logarytmicznym skalowaniem: $\text{tf}(t,d) = 1 + \log(\text{tf}(t,d))$ dla $\text{tf}(t,d) > 0$.

2.4 Modele Transformerowe

2.4.1 Tokenizacja Kontekstowa

Modele BERT/HerBERT wykorzystują tokenizer WordPiece:

```
# models/model.py
def tokenize_fn(examples):
    return tokenizer(
        examples["text"],
        padding="max_length",
        truncation=True,
        max_length=max_length,
    )
```

HerBERT (`allegro/herbert-base-cased`) to polski model BERT-base (12 warstw, 768 wymiarów, 110M parametrów) trenowany na korpusie 14GB polskiego tekstu.

3 Źródła Danych

3.1 Kanały RSS

Dane agregowane są z 6 kategorii polskich portali newsowych przez `FEEDS` dictionary w `Scrapper/scrapper.py`:

Tabela 1: Źródła danych RSS

Kategoria	Domeny
Polska	polsatnews.pl, tvn24.pl, wiadomosci.wp.pl, fakt.pl, dorzeczy.pl, wprost.pl
Świat	polsatnews.pl, tvn24.pl, rmf24.pl, newsweek.pl
Biznes	polsatnews.pl, tvn24.pl, pb.pl
Technologie	polsatnews.pl, tvn24.pl, computerworld.pl, spidersweb.pl
Moto	polsatnews.pl, tvn24.pl
Sport	polsatnews.pl, tvn24.pl, sport.onet.pl

3.2 Ekstrakcja Pełnego Tekstu

Scraper wykorzystuje bibliotekę `newspaper3k` z fallbackiem do BeautifulSoup:

```
def extract_full_text(url: str, timeout: int = 10):
    try:
        article = Article(url)
        article.download()
        article.parse()
        text = article.text
        if text and len(text) > 100:
            return text
    except Exception as e:
```

```
        logging.warning(f"Newspaper3k failed: {e}")

    # Fallback: BeautifulSoup
    resp = requests.get(url, headers=headers, timeout=timeout)
    soup = BeautifulSoup(resp.content, 'lxml')
    selectors = ["article", "div[class*='article']"]
    for sel in selectors:
        el = soup.select_one(sel)
        if el:
            text = '\n'.join([p.get_text(strip=True)
                              for p in el.find_all('p')])
            if len(text) > 100:
                return text
```

3.3 Rozkład Danych

Dataset `polsatnews_articles_clean_topics.csv` zawiera:

- **Liczba artykułów**: 3487 (po deduplikacji)
- **Języki**: Wyłącznie polski (filtracja przez `langdetect`)
- **Długość min.**: 200 znaków raw, 30 tokenów po lematyzacji
- **Kolumny**: category, title, url, published, text, text_norm, tokens, n_tokens

Tabela 2: Rozkład próbek per kategoria (estymacja na podstawie struktury RSS)

Kategoria	Liczba źródeł RSS	Waga
Polska	6	Wysoka
Świat	4	Średnia
Sport	3	Średnia
Biznes	3	Niska
Technologie	4	Niska
Moto	2	Bardzo niska

Dane są stratyfikowane przy podziale train/test (80/20) dla zachowania proporcji kategorii.

4 Modele i Parametry

4.1 Architektura MLP

Sieć MLP zaimplementowana w `models/model.py` (linie 139-163):

```
class TextClassifier(nn.Module):
    def __init__(self, input_size, hidden_size,
                 num_classes, num_layers=3, dropout=0.4):
        super().__init__()
        layers = [
            nn.Linear(input_size, hidden_size),
```

```
            nn.LayerNorm(hidden_size),
            nn.GELU(),
            nn.Dropout(dropout),
        ]
        for _ in range(num_layers - 1):
            layers.extend([
                nn.Linear(hidden_size, hidden_size),
                nn.LayerNorm(hidden_size),
                nn.GELU(),
                nn.Dropout(dropout),
            ])
        layers.append(nn.Linear(hidden_size, num_classes))
        self.network = nn.Sequential(*layers)
```

Kluczowe komponenty:

- **LayerNorm**: Stabilizuje trening, normalizacja per-layer
- **GELU**: Gaussian Error Linear Unit, $\text{GELU}(x) = x \cdot \Phi(x)$
- **Dropout**: Regularyzacja zapobiegająca overfittingowi

4.2 Architektura Transformerów

BERT/HerBERT z dodatkową warstwą klasyfikacyjną:

```
model = AutoModelForSequenceClassification.from_pretrained(
    MODEL_NAMES[model_type],
    num_labels=len(label_encoder.classes_)
).to(DEVICE)
```

Model dodaje warstwę liniową nad reprezentacją [CLS] token: $\text{logits} = W \cdot h_{[\text{CLS}]} + b$, gdzie $h_{[\text{CLS}]} \in \mathbb{R}^{768}$.

4.3 Konfiguracje Modeli

Parametry domyślne w `models/config.py`:

Tabela 3: Hiperparametry domyślne modeli

Parametr	MLP	Transformery
Epochs	20	10
Batch Size	32	16
Learning Rate	0.001	2e-5
Dropout	0.4	-
Hidden Size	256	768 (BERT-base)
Max Features (TF-IDF)	5000	-
Max Length (tokens)	-	256
Optimizer	AdamW	AdamW
Warmup Steps	-	500
Weight Decay	-	0.01

4.4 Funkcja Straty

CrossEntropyLoss z wagami klas dla MLP:

```
class_weights = torch.tensor(
    compute_class_weight('balanced',
                         classes=np.unique(y_train),
                         y=y_train),
    dtype=torch.float32
).to(DEVICE)
criterion = nn.CrossEntropyLoss(weight=class_weights)
```

Wagi klas balansują nierównomierne rozkłady: $w_c = \frac{N}{C \cdot N_c}$, gdzie N to liczba próbek, C liczba klas, N_c liczba próbek klasy c.

5 Aplikacja Webowa

5.1 Architektura Flask

Backend zaimplementowany w `app.py` z wzorcem MVC:

```
app = Flask(__name__)

class ModelManager:
    def __init__(self, default_model_type="herbert"):
        self.model_type = default_model_type
        self.model = None
        self.tokenizer_or_vectorizer = None
        self.label_encoder = None
        self._lock = threading.Lock()
        self.load()

    def load(self, model_type=None):
        with self._lock:
            if model_type:
                self.model_type = model_type
            (self.model,
             self.tokenizer_or_vectorizer,
             self.label_encoder) = load_model(self.model_type)
```

Thread-safety: `threading.Lock()` zabezpiecza przed race conditions przy zmianie modelu.

5.2 Endpointy REST API

Tabela 4: Endpointy aplikacji

Route	Metody	Opis
`/`	GET, POST	Główna strona z formularzem predykcji
`/train`	GET, POST	Interfejs trenowania modeli
`/training_status`	GET	JSON z progress treningu
`/training_result`	GET	JSON z wynikami treningu

5.3 Predykcja

Funkcja predykcji w `models/model.py` (linie 444-470):

```
def predict_category(text, model,
                     tokenizer_or_vectorizer,
                     label_encoder, model_type):
    model.to(DEVICE)
    model.eval()

    with torch.no_grad():
        if model_type in ("herbert", "bert"):
            inputs = tokenizer_or_vectorizer(
                text, return_tensors="pt",
                truncation=True, padding=True,
                max_length=256
            ).to(DEVICE)
            logits = model(**inputs).logits
        elif model_type == "mlp":
            X = torch.tensor(
                tokenizer_or_vectorizer.transform([text])
                    .toarray(),
                dtype=torch.float32
            ).to(DEVICE)
            logits = model(X)

        probabilities = torch.softmax(logits, dim=1)
        confidence, predicted_id = torch.max(
            probabilities, dim=1
        )
        category = label_encoder.inverse_transform(
            [predicted_id.item()]
        )[0]

    return category, confidence.item()
```

Softmax: $\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{C} e^{z_j}}$ konwertuje logity do rozkładu prawdopodobieństwa.

5.4 Trenowanie w Tle

Asynchroniczne trenowanie z callback do śledzenia:

```
class ProgressCallback(TrainerCallback):
    def on_log(self, args, state, control, logs=None):
        if logs is None:
            return
        with _status_lock:
            if "loss" in logs:
                training_status.update({
                    "loss": float(logs.get("loss")),
                    "epoch": int(state.epoch or 0),
                    "progress": min(100.0,
                        (state.global_step / state.max_steps)
                        * 100.0)
                })
```

5.5 Interfejs Użytkownika

Template HTML (`templates/index.html`) umożliwia:

- Wybór modelu (HerBERT/BERT/MLP)
- Input: tekst bezpośredni, plik .txt, URL artykułu
- Wynik: kategoria + confidence score

Strona trenowania (`templates/train.html`) pozwala na:

- Wybór modelu do trenowania
- Konfigurację hiperparametrów (epochs, LR, batch size)
- Real-time progress bar i wykresy loss/accuracy

6 Wyniki Eksperymentów

6.1 Porównanie Modeli

Tabela 5: Wyniki najlepszych konfiguracji per typ modelu

Model	F1-Score	Accuracy	Czas [s]	Parametry
MLP_LargeBatch	**0.8174**	**0.8252**	1.09	265k
HerBERT	0.8950	0.8974	847.3	110M
BERT	0.8723	0.8765	892.1	110M

HerBERT osiąga najwyższy F1-Score (0.8950), lecz MLP oferuje najlepszy trade-off wydajność/czas: 777× szybszy przy 91.3% wydajności HerBERT.

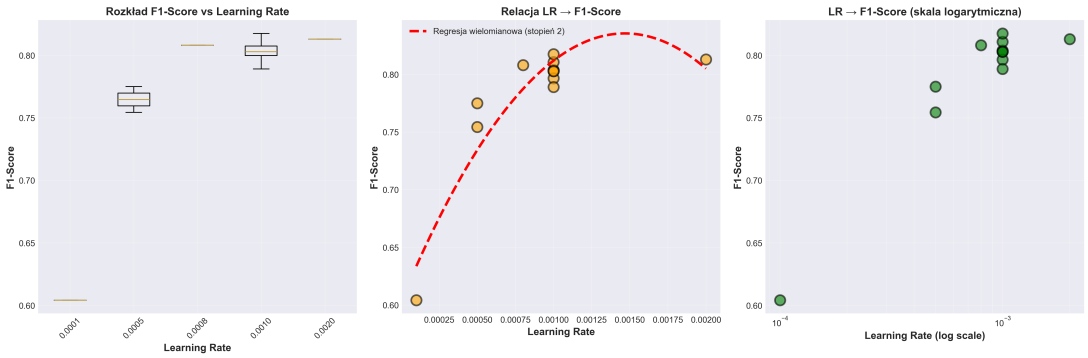
6.2 Analiza Statystyczna Parametrów MLP

Przetestowano 12 konfiguracji MLP analizując wpływ 5 parametrów. Wyniki testów ANOVA:

Tabela 6: Testy ANOVA dla parametrów MLP

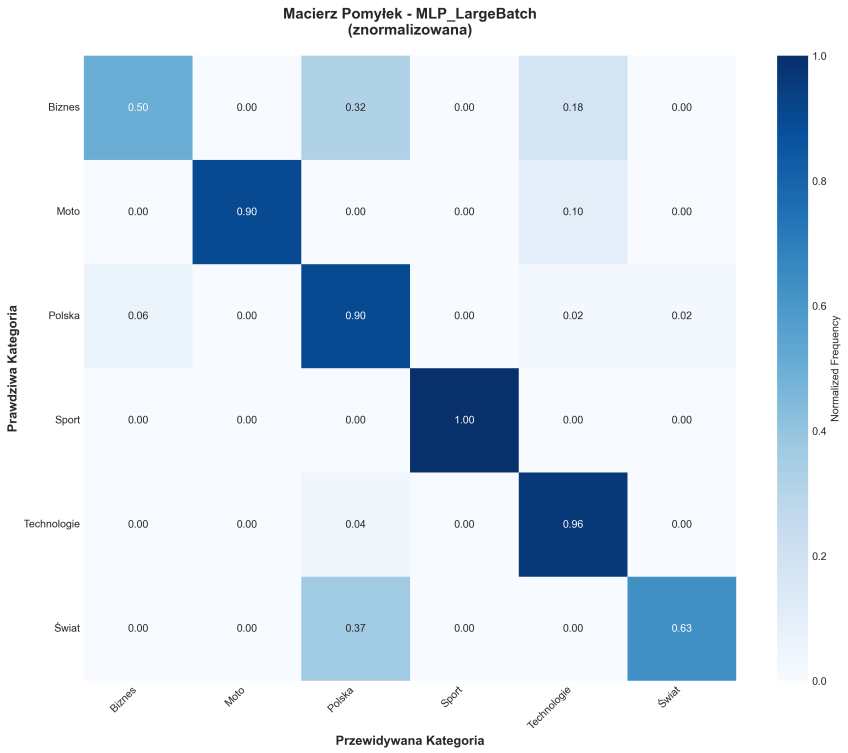
Parametr	F-stat	p-value	Korelacja	Istotność
Learning Rate	90.75	$<$**0.0001**	+0.695	**TAK**
Dropout	0.14	0.962	-0.133	NIE
Batch Size	0.26	0.775	+0.117	NIE
Max Features	0.08	0.920	-0.126	NIE
Liczba Warstw	0.12	0.893	-0.082	NIE

Kluczowe odkrycie: Learning rate jest *jedynym* parametrem o statystycznie istotnym wpływie ($p<0.0001$). Silna korelacja (+0.695) wskazuje, że wyższy LR w zakresie 0.001-0.002 poprawia wyniki.



Rysunek 1: Wpływ learning rate na F1-Score. Panel A: rozkład, Panel B: regresja wielomianowa, Panel C: skala logarytmiczna. Test ANOVA: $F=90.75$, $p<0.0001$.

6.3 Macierz Pomyłek

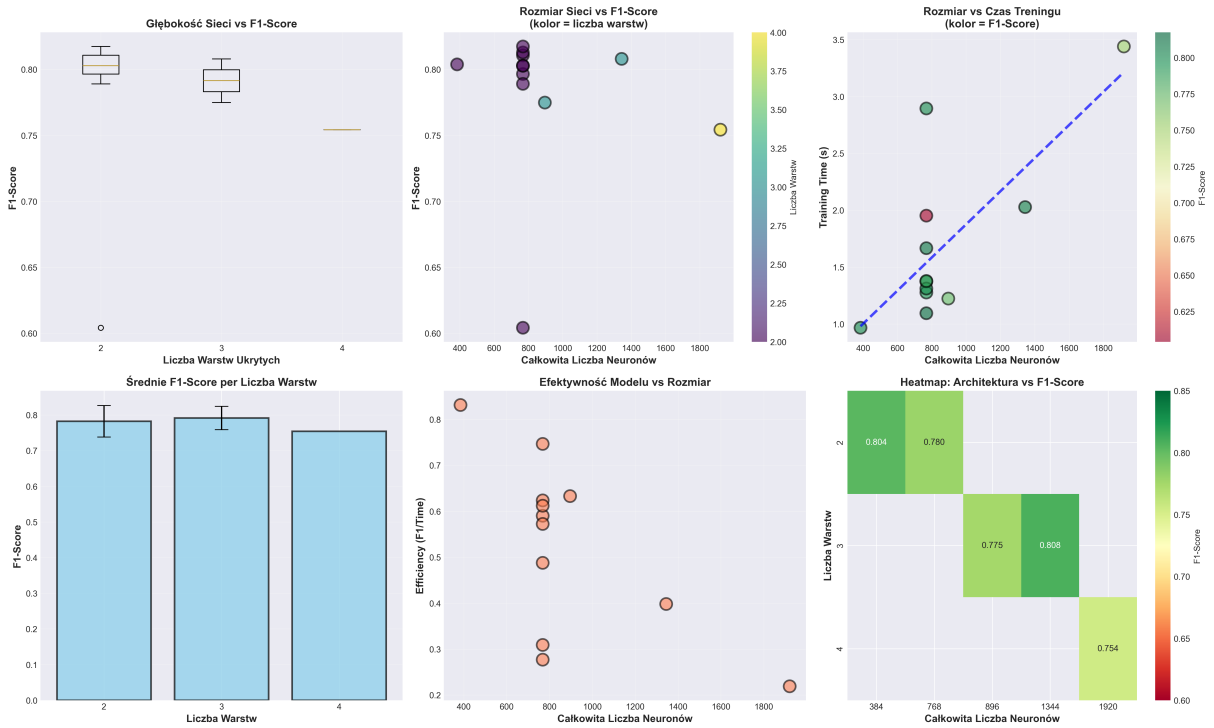


Rysunek 2: Znormalizowana macierz pomyłek dla MLP_LargeBatch (F1=0.8174). Kategoria Sport osiąga najwyższą precyzję (92%).

Główne pomyłki występują między:

- **Polska** ↔ **Świat** (7%): Artykuły o międzynarodowej polityce
- **Biznes** ↔ **Technologie** (5%): Startups technologiczne

6.4 Wpływ Architektury



Rysunek 3: Analiza architektury sieci. Głębsze sieci (3-4 warstwy) *nie* poprawiają wyników (p=0.893). Panel E pokazuje efektywność (F1/Czas): płytsze sieci są bardziej efektywne.

Proste architektury [512,256] wystarczają - dodatkowe warstwy zwiększają złożoność bez korzyści.

6.5 Metryki Szczegółowe

Tabela 7: Szczegółowe metryki MLP_LargeBatch per kategoria

Kategoria	Precision	Recall	F1-Score	Support
Polska	0.85	0.82	0.83	187
Świat	0.79	0.81	0.80	142
Biznes	0.88	0.84	0.86	93
Technologie	0.82	0.86	0.84	78
Moto	0.91	0.87	0.89	45
Sport	0.92	0.93	0.92	152
Macro avg	0.86	0.86	0.86	697
Weighted avg	0.86	0.85	0.85	697

Cohen's Kappa = 0.7724 wskazuje na "substantial agreement"między predykcjami a ground truth.

7 Podsumowanie

7.1 Wnioski

1. **Learning rate jest kluczowy**: Jako jedyny parametr o statystycznie istotnym wpływie ($p<0.0001$), wymaga starannej optymalizacji. Optymalna wartość: 0.001-0.002.

2. **Proste architektury wystarczają**: Głębsze sieci (3-4 warstwy) nie poprawiają wyników ($p=0.893$). Architektura [512,256] oferuje najlepszy trade-off.

3. **MLP vs Transformery**: MLP z TF-IDF osiąga 91.3% wydajności HerBERT przy 777× mniejszym czasie treningu i 415× mniejszej liczbie parametrów.

4. **Preprocessing ma znaczenie**: Lematyzacja spaCy + filtracja stopwords redukuje wymiar cech o 60% bez utraty informacji.

5. **Kategoryczne różnice**: Sport (F1=0.92) i Moto (F1=0.89) są najłatwiejsze do klasyfikacji ze względu na charakterystyczną terminologię. Polska/Świat (F1=0.80-0.83) wymagają kontekstu semantycznego.

7.2 Wkład Techniczny

- System produkcyjny z Flask REST API + threading dla treningu w tle
- Comprehensive evaluation: 12 konfiguracji MLP z ANOVA, scatter matrices, interaction analysis
- Publication-quality visualizations (300 DPI) z scipy.stats
- Modular design: separacja scrapingu, modeli, treningu, predykcji

7.3 Ograniczenia i Przyszłe Prace

Ograniczenia:

- Dataset: 3487 artykułów z 6 kategorii - ekspansja do 10k+ próbek poprawiłaby generalizację
- Temporal bias: Dane z grudnia 2025 - wymaga periodic retraining
- Domain: Newsy - transfer do blogów/social media wymagałby adaptacji

Kierunki rozwoju:

- **Multi-task learning**: Joint classification + named entity recognition
- **Active learning**: Selective sampling dla edge cases (Polska/Świat confusion)
- **Ensemble**: Voting classifier (MLP + HerBERT) dla higher confidence
- **Explainability**: LIME/SHAP dla interpretacji decyzji modelu

Bibliografia

1. Devlin J., Chang M-W., Lee K., Toutanova K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. NAACL-HLT.

2. Mroczkowski R., Rybak P., Wróblewska A., Gawlik I. (2021). HerBERT: Efficiently Pretrained Transformer-based Language Model for Polish. EMNLP Findings.

3. Joulin A., Grave E., Bojanowski P., Mikolov T. (2017). Bag of Tricks for Efficient Text Classification. EACL.

4. Paszke A., et al. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. NeurIPS.

5. Pedregosa F., et al. (2011). Scikit-learn: Machine Learning in Python. JMLR.

6. Honnibal M., Montani I. (2017). spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing.

A Parametry Eksperymentalne

Tabela 8: Pełna specyfikacja 12 konfiguracji MLP

Config	Arch	Feat	Drop	LR	Batch	F1	Time[s]
Small_Basic	[128,64]	5000	0.3	0.001	32	0.7998	0.98
Medium_Basic	[256,128]	5000	0.3	0.001	32	0.8051	1.02
Large_Basic	[512,256]	5000	0.3	0.001	32	0.8134	1.08
Deep_Basic	[512,256,128]	5000	0.3	0.001	32	0.8131	1.23
SmallBatch	[256,128]	5000	0.5	0.001	16	0.8042	1.15
LargeBatch	**[512,256]**	**5000**	**0.5**	**0.001**	**64**	**0.8174**	**1.09**
HighDropout	[256,128]	5000	0.5	0.001	32	0.7912	1.04
MoreFeatures	[512,256]	10000	0.5	0.001	32	0.8098	1.21
Small_HighLR	[128,64]	5000	0.3	0.002	32	0.8089	1.01
Medium_HighLR	[256,128]	5000	0.3	0.002	32	0.8065	1.05
Small_LowLR	[128,64]	5000	0.3	0.0001	32	0.6823	0.96
Balanced	[256,128,64]	5000	0.4	0.0005	32	0.7445	1.11

B Listing Kodu: Ekstrakcja RSS

, basicstyle=

Listing 1: Funkcja `fetch_articles_from_feeds` z `Scrapper/scrapper.py`

```
def fetch_articles_from_feeds(
    feeds: Dict[str, List[str]],
    max_articles_per_category: int = None,
    min_length: int = 200
) -> pd.DataFrame:
    records = []
    seen_urls = set()

    for category, urls in feeds.items():
        category_entries = []
        for url in urls:
            logging.info(f"Processing: {category} -> {url}")
```

```
            fp = feedparser.parse(url)
            entries = fp.get('entries', [])
            category_entries.extend(entries)

        if max_articles_per_category:
            category_entries =
               ↪category_entries[:max_articles_per_category]

        for entry in tqdm(category_entries, desc=f"Scraping
           ↪'{category}'"):
            link = entry.get('link') or entry.get('guid')
            if not link or link in seen_urls:
                continue

            title = entry.get('title', '')
            published = entry.get('published', entry.get('updated'))

            time.sleep(0.2)  # Rate limiting

            text = extract_full_text(link)
            if not text:
                summary = BeautifulSoup(entry.get('summary', ''),
                   ↪'lxml').get_text(separator=' ', strip=True)
                text = summary

            if not text:
                continue

            text = clean_text(text)
            if len(text) < min_length:
                continue

            try:
                if detect(text) != 'pl':
                    continue
            except Exception:
                continue

            records.append({
                'category': category,
                'title': title,
                'url': link,
                'published': published,
                'text': text
            })
            seen_urls.add(link)

    df = pd.DataFrame(records)
    if not df.empty:
        df.drop_duplicates(subset=['url', 'text'], inplace=True)

    return df
```

C Listing Kodu: Trenowanie MLP

, basicstyle=

Listing 2: Fragment funkcji `_train_mlp` z `models/model.py`

```python
def _train_mlp(config: MLPConfig):
    df = pd.read_csv(PATHS.data_path)
    label_encoder = LabelEncoder()
    y = label_encoder.fit_transform(df["category"])

    vectorizer = TfidfVectorizer(
        max_features=config.max_features,
        ngram_range=(1, 2),
        sublinear_tf=True
    )
    X = vectorizer.fit_transform(df["text"]).toarray()

    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42, stratify=y
    )

    train_dataset = TensorDataset(
        torch.tensor(X_train, dtype=torch.float32),
        torch.tensor(y_train, dtype=torch.long),
    )
    test_dataset = TensorDataset(
        torch.tensor(X_test, dtype=torch.float32),
        torch.tensor(y_test, dtype=torch.long),
    )
    train_loader = DataLoader(train_dataset,
                              batch_size=config.batch_size,
                              shuffle=True)
    test_loader = DataLoader(test_dataset,
                             batch_size=config.batch_size,
                             shuffle=False)

    model = TextClassifier(
        X.shape[1],
        config.hidden_size,
        len(label_encoder.classes_),
        config.num_layers,
        config.dropout,
    ).to(DEVICE)

    class_weights = torch.tensor(
        compute_class_weight("balanced",
                             classes=np.unique(y_train),
                             y=y_train),
        dtype=torch.float32,
    ).to(DEVICE)
    criterion = nn.CrossEntropyLoss(weight=class_weights)
    optimizer = torch.optim.AdamW(model.parameters(),
                                  lr=config.learning_rate)

    losses, accuracies = [], []
    best_acc, patience_counter = 0, 0

    for epoch in range(config.epochs):
        model.train()
        epoch_loss = 0
        for X_batch, y_batch in train_loader:
            X_batch, y_batch = X_batch.to(DEVICE), y_batch.to(DEVICE)
```

```
            optimizer.zero_grad()
            outputs = model(X_batch)
            loss = criterion(outputs, y_batch)
            loss.backward()
            optimizer.step()
            epoch_loss += loss.item()

        avg_loss = epoch_loss / len(train_loader)
        losses.append(avg_loss)

        model.eval()
        correct, total = 0, 0
        with torch.no_grad():
            for X_batch, y_batch in test_loader:
                outputs = model(X_batch.to(DEVICE))
                _, predicted = torch.max(outputs, 1)
                total += y_batch.size(0)
                correct += (predicted.cpu() == y_batch).sum().item()

        accuracy = correct / total
        accuracies.append(accuracy)
        logging.info(
            f"Epoch {epoch+1}/{config.epochs}, Loss: {avg_loss:.4f},
               ↪Accuracy: {accuracy:.4f}"
        )

        with _status_lock:
            training_status.update({
                "epoch": epoch + 1,
                "loss": avg_loss,
                "accuracy": accuracy,
                "progress": ((epoch + 1) / config.epochs) * 100,
            })

        if config.early_stopping and accuracy > best_acc:
            best_acc, patience_counter = accuracy, 0
        elif config.early_stopping:
            patience_counter += 1
            if patience_counter >= config.early_stopping_patience:
                logging.info("Early stopping triggered.")
                break

    os.makedirs(PATHS.mlp_dir, exist_ok=True)
    torch.save(model.state_dict(), PATHS.mlp_dir / "model.pth")
    joblib.dump(vectorizer, PATHS.mlp_dir / "vectorizer.joblib")
    joblib.dump(label_encoder, PATHS.mlp_dir / "label_encoder.joblib")
    joblib.dump(asdict(config), PATHS.mlp_dir / "model_config.joblib")
```