Model Training, Evaluation, and Testing Guide

AI Stocks Project - Comprehensive Model Documentation

Table of Contents

Introduction

This document provides comprehensive details on how each model in the AI Stocks project is trained, evaluated, tested, and fine-tuned. The project includes five models:

1. **Baseline Model**: Simple moving average-based technical indicator
2. **MLP Model**: Multi-layer perceptron for stock price direction prediction
3. **Transformer Model**: Transformer encoder for sequence-based stock prediction
4. **Sentiment LSTM Model**: LSTM/GRU-based sentiment classification
5. **Sentiment BERT Model**: Fine-tuned BERT for financial sentiment analysis

Each model serves a specific purpose in the stock prediction pipeline, from technical analysis to sentiment understanding.

1. Baseline Model

1.1 Model Architecture

File: `ml/baseline_model.py`

The baseline model is a simple rule-based system that uses technical indicators to predict stock direction. It does not require training as it uses deterministic calculations.

Key Components:

- **Moving Average Calculation**: Computes short-term (10-day) and long-term (30-day) moving averages

- **Trend Signal Generation**: Compares moving averages to determine trend direction

- **Price Recommendation**: Suggests buy/sell prices based on current price levels

Architecture Overview:

```
Input: StockPriceSeries (historical prices) ↓ Calculate Moving Averages (MA_10, MA_30) ↓ Compare MA_10 vs MA_30 ↓ Generate Direction Signal (up/down/flat) + Confidence ↓ Recommend Buy/Sell Prices ↓ Output: PredictionOutput
```

No trainable parameters - This is a rule-based model.

1.2 Data Preparation

Input Data Format:

- **StockPriceSeries**: Contains a list of `PricePoint` objects with:

- `date`: datetime.date

- `open`, `high`, `low`, `close`: float (prices)

- `volume`: int (trading volume)

Data Requirements:

- Minimum 30 days of historical data

- Data must be sorted chronologically

- No missing values in price data

Preprocessing Steps:

1. Convert `StockPriceSeries` to pandas DataFrame

2. Set date as index

3. Sort by date (ascending)

4. Filter data up to the prediction date (`as_of_date`)

Code Example:

```
def _to_dataframe(series: StockPriceSeries) -> pd.DataFrame: records = [ { "date": p.date, "open": p.open, "high": p.high, "low": p.low, "close": p.close, "volume": p.volume, } for p in series.prices ] df = pd.DataFrame.from_records(records) df["date"] = pd.to_datetime(df["date"]) df.sort_values("date", inplace=True) df.set_index("date", inplace=True)
```

```
return df
```

1.3 Training Process

No training required - This is a deterministic rule-based model.

The model uses fixed rules:

- If short MA > long MA by >2%: Signal = "up"

- If short MA < long MA by >2%: Signal = "down"

- Otherwise: Signal = "flat"

Confidence Calculation:

- Based on relative difference between moving averages

- Normalized by current price

- Range: 0.3 to 0.9

1.4 Evaluation & Testing

Evaluation Method:

- No train/val/test split needed

- Can be evaluated on any historical period

- Performance measured by:

- Direction accuracy (up/down/flat prediction correctness)

- Confidence calibration

Testing Procedure:

1. Load historical stock data

2. For each date in history:

- Use data up to that date

- Generate prediction

- Compare with actual future movement

3. Calculate accuracy metrics

No formal evaluation metrics - Used as a simple baseline for comparison.

1.5 Input/Output Specifications

Input:

- `series`: `StockPriceSeries` object

- Contains historical price data (minimum 30 days)

- `prediction_input`: `PredictionInput` object

- `stock`: Stock object

- `as_of_date`: date (prediction date)

- `horizon_days`: int (not used by baseline)

Output:

- `PredictionOutput` object:

- `should_buy`: bool

- `should_sell`: bool

- `expected_direction`: str ("up", "down", "flat")

- `suggested_buy_price`: float

- `suggested_sell_price`: float

- `confidence`: float (0.0-1.0)

Prediction Parameters:

- `window_short`: 10 days (moving average)

- `window_long`: 30 days (moving average)

- `threshold`: 0.02 (2% relative difference for signal)

- `buy_discount`: 0.98 (2% below current price)

- `sell_premium`: 1.05 (5% above current price)

Example Usage:

```
from ml.baseline_model import run_baseline_model from domain.stocks import
Stock, StockPriceSeries from domain.predictions import PredictionInput
from datetime import date stock = Stock(name="Apple", ticker="AAPL")
series = fetch_price_history(stock, lookback_days=365) pred_input =
PredictionInput(stock=stock, as_of_date=date.today(), horizon_days=30)
prediction = run_baseline_model(series, pred_input) print(f"Direction:
{prediction.expected_direction}") print(f"Confidence:
{prediction.confidence:.2f}")
```

1.5.1 Detailed Example

Let's walk through a concrete example with actual numbers to illustrate how the baseline model works:

Scenario: Predicting Apple (AAPL) stock direction on November 19, 2024.

Step 1: Input Data

Assume we have 40 days of historical price data ending on November 19, 2024. Here's a sample of the last 30 closing prices (in USD):

```
Date Close Price 2024-10-11 175.50 2024-10-14 176.20 2024-10-15 177.80 ... ... 2024-11-15 182.30 2024-11-18 183.50 2024-11-19 185.20 ← Current price (as_of_date)
```

Step 2: Calculate Moving Averages

The model calculates two moving averages:

- **10-day Moving Average (MA_short)**: Average of last 10 closing prices

```

MA_short = (176.20 + 177.80 + ... + 183.50 + 185.20) / 10

MA_short = 181.85

```

- **30-day Moving Average (MA_long)**: Average of last 30 closing prices

```

MA_long = (175.50 + 176.20 + ... + 183.50 + 185.20) / 30

MA_long = 179.20

```

Step 3: Determine Trend Signal

Compare the moving averages:

```
Difference = MA_short - MA_long Difference = 181.85 - 179.20 = 2.65 Relative difference = Difference / Current Price Relative difference = 2.65 / 185.20 = 0.0143 (1.43%)
```

Since the relative difference (1.43%) is less than the 2% threshold, but MA_short > MA_long, the model checks:

- `rel = 0.0143` (positive but < 0.02 threshold)

- Result: **Signal = "flat"** (trend is slightly up but not strong enough)

Alternative Scenario - Strong Uptrend:

If instead:

- MA_short = 186.50

- MA_long = 179.20

- Current price = 185.20

Then:

```
Difference = 186.50 - 179.20 = 7.30 Relative difference = 7.30 / 185.20 = 0.0394 (3.94%)
```

Since `rel = 0.0394 > 0.02`, the signal is **"up"** with confidence calculated as:

```
confidence = min(0.9, rel * 10) = min(0.9, 0.0394 * 10) = min(0.9, 0.394) = 0.394
```

Confidence Calculation Examples:

- For `rel = 0.0394` (3.94%): `confidence = min(0.9, 0.394) = 0.394` (39.4%)

- For `rel = 0.05` (5%): `confidence = min(0.9, 0.5) = 0.5` (50%)

- For `rel = 0.10` (10%): `confidence = min(0.9, 1.0) = 0.9` (90% - capped at maximum)

Step 4: Generate Price Recommendations

Based on the current closing price of $185.20:

- **Suggested Buy Price**: `185.20 × 0.98 = $181.50` (2% below current)

- **Suggested Sell Price**: `185.20 × 1.05 = $194.46` (5% above current)

Step 5: Final Output

For the "flat" scenario:

```
PredictionOutput( should_buy=False, # Not buying in flat market
should_sell=False, # Not selling in flat market expected_direction="flat",
# Trend is neutral suggested_buy_price=181.50, # Entry point if buying
suggested_sell_price=194.46,# Target if selling confidence=0.4 # 40%
confidence (default for flat) )
```

For the "up" scenario (strong uptrend):

```
PredictionOutput( should_buy=True, # Buy signal active should_sell=False,
# Don't sell in uptrend expected_direction="up", # Upward trend detected
suggested_buy_price=181.50, # Entry point suggested_sell_price=194.46,#
Profit target confidence=0.394 # 39.4% confidence )
```

Complete Example with Downward Trend:

If instead:

- MA_short = 172.00

- MA_long = 179.20

- Current price = 185.20

Then:

```
Difference = 172.00 - 179.20 = -7.20 Relative difference = -7.20 / 185.20
= -0.0389 (-3.89%)
```

Since `rel = -0.0389 < -0.02`, the signal is **"down"**:

```
PredictionOutput( should_buy=False, # Don't buy in downtrend
should_sell=True, # Sell signal active expected_direction="down", #
Downward trend detected suggested_buy_price=175.94, # 185.20 × 0.95
(defensive entry) suggested_sell_price=181.50,# 185.20 × 0.98 (take
profit) confidence=0.389 # 38.9% confidence )
```

Key Insights from the Example:

1. **Moving Average Crossover**: The model uses the classic technical analysis pattern of comparing short-term vs long-term moving averages.

2. **Threshold-Based Signals**: The 2% threshold prevents false signals from minor fluctuations while capturing significant trends.

3. **Confidence Scaling**: Confidence increases with the strength of the trend (relative difference), capped at 90% to avoid overconfidence.

4. **Price Recommendations**: Buy/sell prices are simple multipliers of current price, providing actionable entry and exit points.

5. **No Training Required**: All calculations are deterministic based on historical prices, making it fast and interpretable.

1.6 Fine-tuning Procedures

Not applicable - Rule-based model with no parameters to tune.

Adjustable Parameters (manual):

- Moving average windows (10, 30)

- Signal threshold (0.02)

- Buy/sell price multipliers (0.98, 1.05)

1.7 Model Saving & Loading

Not applicable - No model weights to save.

The model logic is embedded in the code and executed directly.

2. MLP Model (Lab 2 Style)

2.1 Model Architecture

File: `ml/lab2_mlp_model.py`

The MLP model is a feedforward neural network that takes tabular features and predicts stock price direction.

Architecture:

```
Input: (batch_size, 8) - Tabular feature vector ↓ Linear(input_dim=8 → hidden_dim) + ReLU ↓ [Repeat for num_layers-1 times] ↓ Linear(hidden_dim → hidden_dim) + ReLU ↓ Linear(hidden_dim → 3) - Output logits ↓ Output: (batch_size, 3) - Logits for 3 classes
```

Key Components:

- **Input Layer**: 8-dimensional feature vector

- **Hidden Layers**: Configurable number of fully connected layers

- **Activation**: ReLU between hidden layers

- **Output Layer**: 3 classes (down=0, flat=1, up=2)

Model Configuration (`domain/configs.py`):

```
@dataclass class MLPConfig: input_dim: int = 8 # Feature vector dimension
hidden_dim: int = 64 # Hidden layer dimension num_layers: int = 2 # Number
of hidden layers
```

Parameter Count Example (hidden_dim=64, num_layers=2):

- Layer 1: (8 × 64) + 64 = 576 parameters

- Layer 2: (64 × 64) + 64 = 4,160 parameters

- Output: (64 × 3) + 3 = 195 parameters

- **Total**: ~4,931 trainable parameters

Code Structure:

```
class StockMLP(nn.Module): def __init__(self, config: MLPConfig,
num_classes: int = 3): super().__init__() layers = [] dim_in =
config.input_dim for _ in range(config.num_layers):
layers.append(nn.Linear(dim_in, config.hidden_dim))
layers.append(nn.ReLU()) dim_in = config.hidden_dim
layers.append(nn.Linear(dim_in, num_classes)) self.net =
nn.Sequential(*layers)
```

2.2 Data Preparation

Input Data Format:

- **StockPriceSeries**: Historical price data for multiple stocks

- **Sentiment Score**: Optional float (-1 to 1)

- **Fundamentals**: Dictionary with `pe_ratio` and `ps_ratio`

Feature Engineering (`_build_tabular_features`):

The model extracts 8 features from a 30-day rolling window:

1. `last_close`: Most recent closing price
2. `ma_10`: 10-day moving average of closes
3. `ma_30`: 30-day moving average of closes
4. `std_10`: 10-day standard deviation of closes
5. `std_30`: 30-day standard deviation of closes
6. `sentiment`: Sentiment score (or 0.0 if unavailable)
7. `pe_ratio`: Price-to-earnings ratio (or 0.0)
8. `ps_ratio`: Price-to-sales ratio (or 0.0)

Dataset Structure (`ReturnDataset`):

- **Input (x)**: `(batch_size, 8)` - Feature vector

- **Target (y)**: `(batch_size,)` - Class label (0, 1, or 2)

Label Encoding:

- Class 0 (■■/down): Future return ≤ -1%

- Class 1 (■■/flat): -1% < Future return < +1%

- Class 2 (■■/up): Future return ≥ +1%

Data Splitting:

- **Training Set**: 80% of samples

- **Validation Set**: 20% of samples

- Random split with shuffling

Code Example:

```
class ReturnDataset(Dataset): def __init__(self, series_list:
List[StockPriceSeries], horizon_days: int = 5): self.features:
List[np.ndarray] = [] self.targets: List[int] = [] threshold = 0.01 # 1%
move threshold for series in series_list: prices = series.prices if
len(prices) < 40: continue sentiment = fetch_sentiment_score(series.stock)
fundamentals = fetch_fundamental_snapshot(series.stock) closes =
np.array([p.close for p in prices], dtype=np.float32) for t in range(30,
len(prices) - horizon_days): window_series = StockPriceSeries(
stock=series.stock, prices=prices[t - 30 : t] ) feats, _ =
_build_tabular_features( window_series, sentiment=sentiment,
fundamentals=fundamentals ) future_return = (closes[t + horizon_days] /
closes[t]) - 1.0 if future_return <= -threshold: label = 0 # down elif
future_return >= threshold: label = 2 # up else: label = 1 # flat
self.features.append(feats.astype(np.float32))
self.targets.append(int(label))
```

2.3 Training Process

Training Script: `ml/train_mlp_model.py`

Hyperparameters:

- **Epochs**: 50 (maximum)

- **Batch Size**: 64

- **Learning Rate**: 1e-3 (0.001)

- **Optimizer**: Adam

- **Loss Function**: CrossEntropyLoss

- **Early Stopping Patience**: 5 epochs

- **Horizon Days**: 5 (future return prediction window)

Hyperparameter Search Space:

The training script performs a grid search over:

- `hidden_dim`: [32, 64, 128]

- `num_layers`: [2, 3]

Training Loop:

```
for epoch in range(epochs): model.train() running_loss = 0.0 for x, y in
train_loader: optimizer.zero_grad() logits = model(x) # Forward pass loss
= criterion(logits, y) # Compute loss loss.backward() # Backward pass
optimizer.step() # Update weights running_loss += loss.item() * x.size(0)
train_loss = running_loss / train_size # Validation model.eval() val_loss
= 0.0 with torch.no_grad(): for x, y in val_loader: logits = model(x) loss
= criterion(logits, y) val_loss += loss.item() * x.size(0) val_loss /=
val_size # Early stopping check if val_loss < best_val_loss - 1e-5:
best_val_loss = val_loss best_state = model.state_dict() no_improve = 0
else: no_improve += 1 if no_improve >= patience: break # Early stopping
```

Training Procedure:

1. Load historical data for watchlist stocks (365 days)

2. Create `ReturnDataset` with 5-day horizon

3. Split into train/val (80/20)

4. For each hyperparameter configuration:

- Initialize model with config

- Train for up to 50 epochs

- Monitor validation loss

- Save best model state

- Apply early stopping if no improvement

5. Select best configuration based on validation loss

6. Save best model checkpoint

Best Model Selection:

- Tracks `overall_best_loss` across all configurations

- Saves model state with lowest validation loss

- Includes configuration in checkpoint for loading

2.4 Evaluation & Testing

Validation Strategy:

- **Train/Val Split**: 80/20 random split

- **Validation Frequency**: Every epoch

- **Metric**: Cross-entropy loss on validation set

Evaluation Metrics:

- **Loss**: Cross-entropy loss (lower is better)

- **Accuracy**: Can be computed from predictions

- **Class Distribution**: Check balance across 3 classes

Testing Procedure:

1. Load trained model from checkpoint

2. Prepare test data (separate from train/val)

3. Run inference on test set

4. Compute accuracy and per-class metrics

5. Compare predictions with actual labels

Performance Benchmarks:

- Baseline comparison: Should outperform baseline model

- Expected accuracy: >50% (better than random 33.3%)

- Class balance: Should handle imbalanced classes

Code Example for Evaluation:

```
model.eval() correct = 0 total = 0 with torch.no_grad(): for x, y in
test_loader: logits = model(x) predictions = logits.argmax(dim=1) correct
+= (predictions == y).sum().item() total += y.size(0) accuracy = correct /
total print(f"Test Accuracy: {accuracy:.4f}")
```

2.5 Input/Output Specifications

Input Format:

- **Shape**: `(batch_size, 8)` or `(1, 8)` for single prediction

- **Type**: `torch.Tensor` (float32)

- **Features**:

1. Last close price

2. 10-day MA

3. 30-day MA

4. 10-day std

5. 30-day std

6. Sentiment score

7. PE ratio

8. PS ratio

Output Format:

- **Shape**: `(batch_size, 3)` - Logits for 3 classes

- **Type**: `torch.Tensor` (float32)

- **Classes**:

- Index 0: Down (■■)

- Index 1: Flat (■■)

- Index 2: Up (■■)

Prediction Process:

```
# Build features feats, last_close = _build_tabular_features(series,
sentiment, fundamentals) x = torch.from_numpy(feats).unsqueeze(0) # (1, 8)
# Forward pass with torch.no_grad(): logits = model(x) # (1, 3) probs =
torch.softmax(logits, dim=-1).cpu().numpy()[0] # Get prediction class_idx
= int(np.argmax(probs)) confidence = float(probs[class_idx])
```

Prediction Parameters:

- `horizon_days`: 5 (default)

- `window_size`: 30 days (for feature extraction)

- `threshold`: 0.01 (1% for class boundaries)

Example Usage:

```
from ml.lab2_mlp_model import predict_with_mlp prediction =
predict_with_mlp( series=series, prediction_input=pred_input,
sentiment=sentiment_score, fundamentals=fundamentals_dict,
weights_path="models/stock_mlp.pth" ) if prediction: print(f"Direction:
{prediction.expected_direction}") print(f"Confidence:
{prediction.confidence:.2f}") print(f"Buy: {prediction.should_buy}, Sell:
{prediction.should_sell}")
```

2.5.1 Detailed Example

Let's walk through a concrete example showing how the MLP model processes data and makes predictions:

Scenario: Predicting Apple (AAPL) stock direction on November 19, 2024, using a trained MLP model.

Step 1: Input Data Collection

We need 30 days of historical price data plus sentiment and fundamentals:

Price Data (last 30 days, closing prices):

```
Date Close Price 2024-10-11 175.50 2024-10-14 176.20 ... ... 2024-11-18
183.50 2024-11-19 185.20 ← Current price
```

Additional Data:

- Sentiment Score: `0.65` (positive sentiment from news)

- PE Ratio: `28.5`

- PS Ratio: `7.2`

Step 2: Feature Extraction

The `_build_tabular_features` function extracts 8 features from the 30-day window:

1. **last_close**: `185.20` (most recent closing price)

2. **ma_10**: 10-day moving average

```

ma_10 = (176.20 + 177.80 + ... + 183.50 + 185.20) / 10

ma_10 = 181.85

```

3. **ma_30**: 30-day moving average

```

ma_30 = (175.50 + 176.20 + ... + 183.50 + 185.20) / 30

ma_30 = 179.20

```

4. **std_10**: 10-day standard deviation

```

std_10 = std([176.20, 177.80, ..., 183.50, 185.20])

std_10 = 3.45

```

5. **std_30**: 30-day standard deviation

```

std_30 = std([175.50, 176.20, ..., 183.50, 185.20])

std_30 = 4.12

```

6. **sentiment**: `0.65` (from news sentiment analysis)

7. **pe_ratio**: `28.5` (from fundamentals)

8. **ps_ratio**: `7.2` (from fundamentals)

Feature Vector:

```
features = np.array([ 185.20, # last_close 181.85, # ma_10 179.20, # ma_30
3.45, # std_10 4.12, # std_30 0.65, # sentiment 28.5, # pe_ratio 7.2 #
ps_ratio ], dtype=np.float32)
```

Step 3: Model Forward Pass

The feature vector is passed through the MLP network:

Architecture (example: hidden_dim=64, num_layers=2):

```
Input: (1, 8) feature vector ↓ Linear(8 → 64) + ReLU ↓ Linear(64 → 64) + ReLU ↓ Linear(64 → 3) ↓ Output: (1, 3) logits
```

Forward Pass Calculation (simplified, actual weights are learned):

1. **First Layer**: `Linear(8 → 64)`

```

h1 = ReLU(W1 @ features + b1)

h1 shape: (64,)

```

2. **Second Layer**: `Linear(64 → 64)`

```

h2 = ReLU(W2 @ h1 + b2)

h2 shape: (64,)

```

3. **Output Layer**: `Linear(64 → 3)`

```

logits = W3 @ h2 + b3

logits shape: (3,)

```

Example Output Logits:

```
logits = np.array([-0.5, 0.2, 1.8]) # [down, flat, up]
```

Step 4: Probability Calculation

Apply softmax to convert logits to probabilities:

```
probs = softmax(logits) probs = np.array([0.10, 0.25, 0.65]) # [down, flat, up]
```

Step 5: Prediction

Select the class with highest probability:

```
class_idx = np.argmax(probs) # = 2 (up) confidence = probs[class_idx] # = 0.65 (65%)
```

Step 6: Generate Output

Since `class_idx = 2` (up):

```
PredictionOutput( should_buy=True, # Buy signal (upward trend) should_sell=False, # Don't sell expected_direction="up", # Upward prediction suggested_buy_price=181.50, # last_close * 0.98 = 185.20 * 0.98 suggested_sell_price=194.46, # last_close * 1.05 = 185.20 * 1.05 confidence=0.65 # 65% confidence )
```

Alternative Scenario - Down Prediction:

If logits were `[-1.2, 0.3, -0.5]`:

```
probs = softmax([-1.2, 0.3, -0.5]) = [0.15, 0.60, 0.25] class_idx = 1 # flat confidence = 0.60
```

Output:

```
PredictionOutput( should_buy=False, should_sell=False, expected_direction="flat", suggested_buy_price=183.35, # 185.20 * 0.99 suggested_sell_price=187.05, # 185.20 * 1.01 confidence=0.60 )
```

Key Insights from the Example:

1. **Feature Engineering**: The model combines technical indicators (MAs, std) with sentiment and fundamentals for richer input.

2. **Non-linear Processing**: Multiple layers with ReLU activation allow the model to learn complex patterns.

3. **Probabilistic Output**: Softmax provides probability distribution, not just a single class, giving confidence scores.

4. **Multi-class Classification**: Three classes allow nuanced predictions (up/flat/down) compared to binary classification.

5. **End-to-end Pipeline**: From raw prices to actionable buy/sell recommendations with confidence scores.

2.6 Fine-tuning Procedures

Fine-tuning Approach:

The model uses hyperparameter search during initial training. For fine-tuning:

1. **Load Pre-trained Model**:

```python

checkpoint = torch.load("models/stock_mlp.pth")

config = MLPConfig(**checkpoint["config"])

model = StockMLP(config, num_classes=3)

model.load_state_dict(checkpoint["state_dict"])

```

2. **Adjust Learning Rate**:

- Use lower learning rate (e.g., 1e-4) for fine-tuning

- Freeze early layers if needed

3. **Continue Training**:

- Train on new data with reduced learning rate

- Monitor validation loss

- Apply early stopping

Transfer Learning:

- Can initialize from pre-trained weights

- Fine-tune on domain-specific data

- Adjust final layer if class distribution changes

Hyperparameter Adjustments:

- **Learning Rate**: Reduce by 10x for fine-tuning

- **Batch Size**: Keep same or reduce slightly

- **Epochs**: Fewer epochs needed (10-20)

Best Practices:

- Always validate on held-out set

- Monitor for overfitting

- Save checkpoints regularly

- Compare with baseline before/after fine-tuning

2.7 Model Saving & Loading

Checkpoint Format:

```
checkpoint = { "config": { "input_dim": 8, "hidden_dim": 64, "num_layers":
2, }, "state_dict": model.state_dict(), } torch.save(checkpoint,
"models/stock_mlp.pth")
```

Model File Location:

- **Path**: `models/stock_mlp.pth`

- **Format**: PyTorch checkpoint (.pth)

Loading Procedure:

```
def load_mlp_model(weights_path: str) -> Optional[StockMLP]: if not
os.path.exists(weights_path): return None state = torch.load(weights_path,
map_location="cpu") if isinstance(state, dict) and "state_dict" in state:
cfg_dict = state["config"] config = MLPConfig(
input_dim=cfg_dict.get("input_dim", 8),
hidden_dim=cfg_dict.get("hidden_dim", 64),
num_layers=cfg_dict.get("num_layers", 2), ) model = StockMLP(config)
model.load_state_dict(state["state_dict"]) else: # Backward compatibility
config = MLPConfig() model = StockMLP(config) model.load_state_dict(state)
model.eval() return model
```

Usage in Frontend:

The frontend (`frontend/app.py`) loads the model automatically when making predictions:

```
mlp_pred = predict_with_mlp( series, pred_input, sentiment=sentiment,
fundamentals=fundamentals )
```

3. Transformer Model (Lab 5 Style)

3.1 Model Architecture

File: `ml/lab5_transformer_model.py`

The Transformer model uses a Transformer encoder to process sequences of daily stock features and predict price direction.

Architecture:

```
Input: (batch_size, seq_len=30, feature_dim=8) ↓ Input Projection:
Linear(8 → d_model) ↓ Positional Encoding: Sinusoidal encoding added ↓
Transformer Encoder: num_layers × TransformerEncoderLayer ■■ Multi-Head
Attention (nhead heads) ■■ Feed-Forward Network (dim_feedforward) ■■
Layer Normalization + Residual connections ↓ Sequence Aggregation: Use
last time step ↓ Classification Head: LayerNorm + Linear(d_model → 3) ↓
Output: (batch_size, 3) - Logits for 3 classes
```

Key Components:

1. **Input Projection Layer**:

- Projects 8-dimensional features to `d_model` dimensions

- `nn.Linear(feature_dim, d_model)`

2. **Positional Encoding**:

- Sinusoidal positional encoding

- Added to input embeddings

- Max length: `max_len` (default 128)

3. **Transformer Encoder**:

- Stack of `num_layers` encoder layers

- Each layer contains:

- Multi-head self-attention (`nhead` heads)

- Feed-forward network (`dim_feedforward` hidden units)

- Layer normalization

- Residual connections

- Dropout for regularization

4. **Classification Head**:

- Takes last time step from encoder output

- Layer normalization

- Linear projection to 3 classes

Model Configuration (`domain/configs.py`):

```
@dataclass class TransformerConfig: d_model: int = 32 # Model dimension
nhead: int = 4 # Number of attention heads num_layers: int = 2 # Number of
encoder layers dim_feedforward: int = 64 # FFN hidden dimension dropout:
float = 0.1 # Dropout rate max_len: int = 128 # Maximum sequence length
```

Parameter Count Example (d_model=64, nhead=8, num_layers=3, dim_feedforward=128):

- Input projection: (8 × 64) + 64 = 576

- Positional encoding: 0 (not trainable)

- Encoder layers (×3):

- Attention: (64 × 64 × 4) × 3 + biases ≈ 49,152

- FFN: (64 × 128) × 2 × 3 + biases ≈ 49,152

- Classification head: (64 × 3) + 3 = 195

- **Total**: ~99,075 trainable parameters

Code Structure:

```
class StockTransformer(nn.Module): def __init__(self, feature_dim: int,
config: TransformerConfig, num_classes: int = 3): super().__init__()
self.input_proj = nn.Linear(feature_dim, config.d_model) self.pos_encoder
= PositionalEncoding(config.d_model, max_len=config.max_len) encoder_layer
= nn.TransformerEncoderLayer( d_model=config.d_model, nhead=config.nhead,
dim_feedforward=config.dim_feedforward, dropout=config.dropout,
batch_first=True, ) self.encoder = nn.TransformerEncoder(encoder_layer,
num_layers=config.num_layers) self.head = nn.Sequential(
nn.LayerNorm(config.d_model), nn.Linear(config.d_model, num_classes), )
```

3.2 Data Preparation

Input Data Format:

- **StockPriceSeries**: Historical price sequences

- **Sentiment Score**: Optional float (broadcast across sequence)

- **Fundamentals**: Dictionary with `pe_ratio` and `ps_ratio`

Feature Engineering (`_build_sequence_features`):

For each day in the sequence, extract 8 features:

1. `open`: Opening price
2. `high`: High price
3. `low`: Low price
4. `close`: Closing price
5. `volume`: Trading volume
6. `sentiment`: Sentiment score (same for all days)

7. `pe_ratio`: PE ratio (same for all days)

8. `ps_ratio`: PS ratio (same for all days)

Sequence Structure:

- **Window Size**: 30 days (default)

- **Sequence Length**: Variable (up to `max_len`)

- **Feature Dimension**: 8 per time step

- **Output Shape**: `(seq_len, 8)`

Dataset Structure (`SequenceReturnDataset`):

- **Input (x)**: `(batch_size, seq_len=30, feature_dim=8)`

- **Target (y)**: `(batch_size,)` - Class label (0, 1, or 2)

Label Encoding (same as MLP):

- Class 0: Future return ≤ -1%

- Class 1: -1% < Future return < +1%

- Class 2: Future return ≥ +1%

Data Splitting:

- **Training Set**: 80% of samples

- **Validation Set**: 20% of samples

- Random split with shuffling

Code Example:

```
class SequenceReturnDataset(Dataset): def __init__(self, series_list:
List[StockPriceSeries], window: int = 30, horizon_days: int = 5):
self.sequences: List[np.ndarray] = [] self.targets: List[int] = []
threshold = 0.01 for series in series_list: prices = series.prices if
len(prices) < window + horizon_days + 1: continue sentiment =
fetch_sentiment_score(series.stock) fundamentals =
fetch_fundamental_snapshot(series.stock) closes = np.array([p.close for p
in prices], dtype=np.float32) for t in range(window, len(prices) -
horizon_days): window_series = StockPriceSeries( stock=series.stock,
prices=prices[t - window : t] ) feats = _build_sequence_features(
window_series, sentiment=sentiment, fundamentals=fundamentals,
max_len=window ) future_return = (closes[t + horizon_days] / closes[t]) -
1.0 # Label encoding same as MLP
self.sequences.append(feats.astype(np.float32))
self.targets.append(int(label))
```

3.3 Training Process

Training Script: `ml/train_transformer_model.py`

Hyperparameters:

- **Epochs**: 50 (maximum)

- **Batch Size**: 64
- **Learning Rate**: 1e-3 (0.001)
- **Optimizer**: Adam
- **Loss Function**: CrossEntropyLoss
- **Early Stopping Patience**: 5 epochs
- **Window Size**: 30 days
- **Horizon Days**: 5

Hyperparameter Search Space:

The training script searches over:

- `d_model`: [32, 64]
- `nhead`: [4, 8]
- `num_layers`: [2, 3]
- `dim_feedforward`: [64, 128]

Training Loop:

```
for epoch in range(epochs): model.train() running_loss = 0.0 for x, y in
train_loader: optimizer.zero_grad() # x: (batch, seq_len, feature_dim)
logits = model(x) # Forward pass loss = criterion(logits, y)
loss.backward() optimizer.step() running_loss += loss.item() * x.size(0)
train_loss = running_loss / train_size # Validation model.eval() val_loss
= 0.0 with torch.no_grad(): for x, y in val_loader: logits = model(x) loss
= criterion(logits, y) val_loss += loss.item() * x.size(0) val_loss /=
val_size # Early stopping (same as MLP)
```

Training Procedure:

1. Load historical data for watchlist stocks (365 days)
2. Create `SequenceReturnDataset` with 30-day window
3. Split into train/val (80/20)
4. For each hyperparameter configuration:

- Initialize Transformer with config
- Train for up to 50 epochs
- Monitor validation loss
- Save best model state
- Apply early stopping

5. Select best configuration
6. Save best model checkpoint

Key Differences from MLP:

- Processes sequences instead of single feature vectors
- Uses attention mechanism to capture temporal dependencies

- More parameters, potentially better for complex patterns
- Requires more memory due to sequence processing

3.4 Evaluation & Testing

Validation Strategy:
- **Train/Val Split**: 80/20 random split
- **Validation Frequency**: Every epoch
- **Metric**: Cross-entropy loss

Evaluation Metrics:
- **Loss**: Cross-entropy loss
- **Accuracy**: Classification accuracy
- **Per-class Metrics**: Precision, recall, F1-score

Testing Procedure:

Same as MLP model - load model and evaluate on test set.

Performance Benchmarks:
- Should outperform MLP on complex temporal patterns
- Expected accuracy: >55% (better than MLP)
- Better at capturing long-term dependencies

3.5 Input/Output Specifications

Input Format:
- **Shape**: `(batch_size, seq_len=30, feature_dim=8)`
- **Type**: `torch.Tensor` (float32)
- **Sequence Features** (per time step):

1. Open price
2. High price
3. Low price
4. Close price
5. Volume
6. Sentiment score
7. PE ratio
8. PS ratio

Output Format:

- **Shape**: `(batch_size, 3)` - Logits for 3 classes

- **Type**: `torch.Tensor` (float32)

- **Classes**: Same as MLP (0=down, 1=flat, 2=up)

Prediction Process:

```
# Build sequence features feats = _build_sequence_features( series,
sentiment=sentiment, fundamentals=fundamentals, max_len=30 ) x =
torch.from_numpy(feats).unsqueeze(0) # (1, 30, 8) # Forward pass with
torch.no_grad(): logits = model(x) # (1, 3) probs = torch.softmax(logits,
dim=-1).cpu().numpy()[0] # Get prediction (same as MLP)
```

Prediction Parameters:

- `window`: 30 days (sequence length)

- `horizon_days`: 5 (future prediction window)

- `threshold`: 0.01 (1% for class boundaries)

Example Usage:

```
from ml.lab5_transformer_model import predict_with_transformer prediction
= predict_with_transformer( series=series, prediction_input=pred_input,
sentiment=sentiment_score, fundamentals=fundamentals_dict,
weights_path="models/stock_transformer.pth" )
```

3.5.1 Detailed Example

Let's walk through a concrete example showing how the Transformer model processes sequence data:

Scenario: Predicting Apple (AAPL) stock direction on November 19, 2024, using a trained Transformer model.

Step 1: Input Data Collection

We need 30 days of sequential price data (OHLCV) plus sentiment and fundamentals:

Price Data (last 30 days):

```
Date Open High Low Close Volume 2024-10-11 175.20 176.10 174.80 175.50
45,200,000 2024-10-14 175.60 177.50 175.40 176.20 48,500,000 2024-10-15
176.50 178.20 176.10 177.80 52,100,000 ... ... ... ... ... ... 2024-11-18
183.20 184.50 182.80 183.50 55,300,000 2024-11-19 183.80 186.00 183.50
185.20 58,700,000 ← Current day
```

Additional Data (broadcast across all days):

- Sentiment Score: `0.65`

- PE Ratio: `28.5`

- PS Ratio: `7.2`

Step 2: Sequence Feature Extraction

The `_build_sequence_features` function creates a 30×8 matrix (30 days × 8 features per day):

Feature Matrix (shape: 30×8):

```
sequence_features = np.array([ # Day 1 (2024-10-11) [175.20, 176.10,
174.80, 175.50, 45200000, 0.65, 28.5, 7.2], # Day 2 (2024-10-14) [175.60,
177.50, 175.40, 176.20, 48500000, 0.65, 28.5, 7.2], # Day 3 (2024-10-15)
[176.50, 178.20, 176.10, 177.80, 52100000, 0.65, 28.5, 7.2], # ... (days
4-29) # Day 30 (2024-11-19) [183.80, 186.00, 183.50, 185.20, 58700000,
0.65, 28.5, 7.2], ], dtype=np.float32)
```

Key Points:

- Each row represents one day

- Features 0-4: Price data (open, high, low, close, volume)

- Features 5-7: Sentiment and fundamentals (same for all days)

Step 3: Model Forward Pass

The sequence is processed through the Transformer encoder:

Architecture (example: d_model=64, nhead=8, num_layers=3):

1. **Input Projection**: `(30, 8) → (30, 64)`

```

projected = Linear(8 → 64)(sequence_features)

# Shape: (30, 64)

```

2. **Positional Encoding**: Add sinusoidal positional encoding

```

pos_encoded = projected + positional_encoding(positions=0..29)

# Shape: (30, 64)

```

3. **Transformer Encoder** (3 layers):

Layer 1:

- Multi-head self-attention (8 heads):

- Each head: `(30, 64) → (30, 64)`

- Concatenate 8 heads: `(30, 64)`

- Feed-forward: `(30, 64) → (30, 128) → (30, 64)`

- Residual connection + LayerNorm

Layer 2: Same as Layer 1

Layer 3: Same as Layer 1

Output: `(30, 64)` - Encoded sequence representation

4. **Sequence Aggregation**: Take last time step

```
last_hidden = encoder_output[-1, :] # Shape: (64,)
```

5. **Classification Head**:

```
normalized = LayerNorm(last_hidden) # Shape: (64,)
logits = Linear(64 → 3)(normalized) # Shape: (3,)
```

Attention Mechanism Example:

The self-attention allows the model to focus on important days. For example:

- Day 30 (current) might attend strongly to:
- Day 29 (recent trend)
- Day 25 (support level)
- Day 20 (previous peak)
- Attention weights might look like:

```
Day 30 attends to:
Day 29: 0.25 (recent price action)
Day 25: 0.15 (support level)
Day 20: 0.10 (previous high)
Day 15: 0.08 (trend continuation)
... (other days with lower attention)
```

Step 4: Output Generation

Example Output Logits:

```
logits = np.array([-0.8, 0.1, 1.5]) # [down, flat, up]
```

Probability Calculation:

```
probs = softmax(logits) probs = np.array([0.12, 0.28, 0.60]) # [down, flat, up]
```

Prediction:

```
class_idx = np.argmax(probs) # = 2 (up) confidence = probs[class_idx] # =
0.60 (60%)
```

Step 5: Final Output

```
PredictionOutput( should_buy=True, should_sell=False,
expected_direction="up", suggested_buy_price=181.50, # 185.20 * 0.98
suggested_sell_price=194.46, # 185.20 * 1.05 confidence=0.60 )
```

Key Differences from MLP Example:

1. **Sequence Processing**: Transformer processes entire 30-day sequence simultaneously, not just aggregated features.

2. **Temporal Dependencies**: Self-attention mechanism captures relationships between different days in the sequence.

3. **Context Awareness**: The model can identify patterns like:

- Support/resistance levels

- Trend reversals

- Volume spikes

- Price momentum

4. **Positional Information**: Positional encoding helps the model understand temporal order.

5. **Rich Representations**: Each day's representation is influenced by all other days through attention, creating richer feature representations.

Visualization of Attention (conceptual):

```
Day 30 (current) ■■attention■■> Day 29 (recent) ■■■attention■■> Day 25
(support) ■■■attention■■> Day 20 (peak) ■■■attention■■> Day 15 (trend)
■■■attention■■> ... (other days)
```

This allows the model to make predictions based on complex temporal patterns rather than simple aggregations.

3.6 Fine-tuning Procedures

Fine-tuning Approach:

Similar to MLP, but with additional considerations:

1. **Load Pre-trained Model**:

```python

checkpoint = torch.load("models/stock_transformer.pth")

config = TransformerConfig(**checkpoint["config"])
```

model = StockTransformer(feature_dim=8, config=config, num_classes=3)

model.load_state_dict(checkpoint["state_dict"])

```

2. **Learning Rate Scheduling**:

- Use lower learning rate for fine-tuning (1e-4)

- Can use learning rate scheduler (ReduceLROnPlateau)

3. **Layer-wise Fine-tuning**:

- Option to freeze encoder layers

- Fine-tune only classification head

- Or fine-tune last N layers

**Transfer Learning**:

- Can initialize from pre-trained Transformer

- Fine-tune on new stock data

- Adjust sequence length if needed

**Hyperparameter Adjustments**:

- **Learning Rate**: 1e-4 for fine-tuning

- **Dropout**: May increase to 0.2 for regularization

- **Batch Size**: Can reduce if memory constrained

**Best Practices**:

- Monitor attention patterns

- Check for overfitting on sequences

- Validate on different time periods

- Compare with MLP baseline

## 3.7 Model Saving & Loading

**Checkpoint Format**:

```
checkpoint = { "config": { "d_model": 64, "nhead": 8, "num_layers": 3,
"dim_feedforward": 128, "dropout": 0.1, "max_len": 128, }, "state_dict":
model.state_dict(), } torch.save(checkpoint,
"models/stock_transformer.pth")
```

**Model File Location**:

- **Path**: `models/stock_transformer.pth`

- **Format**: PyTorch checkpoint (.pth)
```

Loading Procedure:

```
def load_transformer_model(feature_dim: int, weights_path: str) ->
Optional[StockTransformer]: if not os.path.exists(weights_path): return
None state = torch.load(weights_path, map_location="cpu") if
isinstance(state, dict) and "state_dict" in state: cfg_dict =
state["config"] config = TransformerConfig(
d_model=cfg_dict.get("d_model", 32), nhead=cfg_dict.get("nhead", 4),
num_layers=cfg_dict.get("num_layers", 2),
dim_feedforward=cfg_dict.get("dim_feedforward", 64),
dropout=cfg_dict.get("dropout", 0.1), max_len=cfg_dict.get("max_len",
128), ) model = StockTransformer(feature_dim=feature_dim, config=config,
num_classes=3) model.load_state_dict(state["state_dict"]) else: # Backward
compatibility config = TransformerConfig() model =
StockTransformer(feature_dim=feature_dim, config=config, num_classes=3)
model.load_state_dict(state) model.eval() return model
```

4. Sentiment LSTM Model

4.1 Model Architecture

File: `ml/sentiment_lstm_model.py`

The Sentiment LSTM model uses LSTM or GRU layers to classify financial text sentiment into 5 classes.

Architecture:

```
Input: (batch_size, seq_len) - Token indices ↓ Embedding Layer:
Embedding(vocab_size, embedding_dim) ■■ Random initialization OR ■■
Pre-trained GloVe embeddings ↓ RNN Layer: LSTM or GRU ■■ Input:
(batch_size, seq_len, embedding_dim) ■■ Hidden: (num_layers, batch_size,
hidden_dim) ■■ Output: (batch_size, seq_len, hidden_dim) ↓ Sequence
Aggregation: Use last time step ↓ Dropout: Dropout(dropout_rate) ↓
Classification Head: Linear(hidden_dim → num_classes) ↓ Output:
(batch_size, 5) - Logits for 5 classes
```

Key Components:

1. **Embedding Layer**:

- Maps token indices to dense vectors

- `embedding_dim`: 100 (default)

- Supports random or pre-trained (GloVe) embeddings

- Can freeze embeddings during training

2. **RNN Layer**:

- **LSTM** (default) or **GRU** (optional)

- `hidden_dim`: 128 (default)

- `num_layers`: 2 (default)

- Bidirectional: False (unidirectional)

- Dropout between layers (if num_layers > 1)

3. **Classification Head**:

- Dropout layer

- Linear projection to `num_classes` (5)

Model Configuration:

```
class SentimentLSTM(nn.Module): def __init__( self, vocab_size: int,
embedding_dim: int = 100, hidden_dim: int = 128, num_layers: int = 2,
num_classes: int = 5, dropout: float = 0.5, use_gru: bool = False,
embedding_matrix: Optional[np.ndarray] = None, freeze_embeddings: bool =
False, ):
```

Parameter Count Example (vocab_size=5000, embedding_dim=100, hidden_dim=128, num_layers=2):

- Embedding: vocab_size × embedding_dim = 500,000 (if random)

- LSTM layer 1: $4 \times (100 \times 128 + 128^2 + 128) \approx 118{,}784$

- LSTM layer 2: $4 \times (128 \times 128 + 128^2 + 128) \approx 131{,}584$

- Classification: $(128 \times 5) + 5 = 645$

- **Total**: ~750,000+ parameters (depends on vocab size)

Code Structure:

```
class SentimentLSTM(nn.Module): def __init__(self, ...): # Embedding layer
if embedding_matrix is not None: self.embedding =
nn.Embedding.from_pretrained( torch.tensor(embedding_matrix,
dtype=torch.float32), freeze=freeze_embeddings, padding_idx=0, ) else:
self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=0) #
RNN layer rnn_class = nn.GRU if use_gru else nn.LSTM self.rnn = rnn_class(
embedding_dim, hidden_dim, num_layers, batch_first=True, dropout=dropout
if num_layers > 1 else 0, bidirectional=False, ) # Classification head
self.dropout = nn.Dropout(dropout) self.fc = nn.Linear(hidden_dim,
num_classes)
```

4.2 Data Preparation

Input Data Format:

- **Text Data**: Financial news headlines, sentences from Financial PhraseBank

- **Labels**: Integer labels (0-4) for 5-class sentiment

Label Encoding:

- **Class 0**: Very Negative

- **Class 1**: Negative

- **Class 2**: Neutral

- **Class 3**: Positive

- **Class 4**: Very Positive

Data Sources (`ml/sentiment_data.py`):

1. **Financial PhraseBank**: Pre-labeled financial sentences
2. **News Headlines**: Collected from cache, labeled using VADER sentiment analyzer

Preprocessing Pipeline (`ml/sentiment_preprocessing.py`):

1. **Text Cleaning**:

- Remove HTML tags
- Remove URLs
- Remove special characters (keep punctuation)
- Normalize whitespace
- Optional: Remove stopwords

2. **Tokenization**:

- Uses NLTK `word_tokenize` (or simple regex fallback)
- Converts to lowercase
- Splits into word tokens

3. **Vocabulary Building**:

- Builds vocabulary from training texts
- Filters by minimum frequency (`min_freq`)
- Special tokens: (index 0), (index 1)

4. **Sequence Encoding**:

- Converts tokens to indices
- Handles unknown words with
- Pads/truncates to `max_len`

Dataset Structure (`ml/sentiment_dataset.py`):

- **Input (x)**: `(batch_size, seq_len)` - Token indices
- **Target (y)**: `(batch_size,)` - Class label (0-4)

Data Splitting:

- **Training Set**: 70% of samples
- **Validation Set**: 15% of samples
- **Test Set**: 15% of samples
- Stratified split to maintain class distribution

Code Example:

```
from ml.sentiment_data import prepare_sentiment_datasets from
ml.sentiment_dataset import create_datasets # Prepare datasets train_df,
val_df, test_df = prepare_sentiment_datasets() # Create PyTorch datasets
```

```
train_dataset, val_dataset, test_dataset = create_datasets( train_df,
val_df, test_df, max_len=128, min_freq=1, ) vocab =
train_dataset.preprocessor.vocab print(f"Vocabulary size: {len(vocab)}")
```

4.3 Training Process

Training Script: `ml/train_sentiment_model.py`

Hyperparameters:

- **Epochs**: 50 (maximum)

- **Batch Size**: 64

- **Learning Rate**: 1e-3 (0.001)

- **Optimizer**: Adam

- **Loss Function**: CrossEntropyLoss

- **Early Stopping Patience**: 5 epochs

- **Learning Rate Scheduler**: ReduceLROnPlateau (factor=0.5, patience=2)

Model Hyperparameters:

- **Embedding Dimension**: 100

- **Hidden Dimension**: 128

- **Number of Layers**: 2

- **Dropout**: 0.5

- **Max Sequence Length**: 128

- **Min Word Frequency**: 1

- **Use GRU**: False (uses LSTM by default)

- **Embedding Type**: "random" or "glove"

- **Freeze Embeddings**: False (can be True for pre-trained)

Training Loop:

```
for epoch in range(epochs): # Train model.train() train_loss, train_acc =
train_epoch(model, train_loader, criterion, optimizer, device) # Validate
model.eval() val_loss, val_acc = validate(model, val_loader, criterion,
device) # Learning rate scheduling scheduler.step(val_loss) # Check for
improvement if val_loss < best_val_loss: best_val_loss = val_loss
best_val_acc = val_acc patience_counter = 0 # Save best model torch.save({
"model_state_dict": model.state_dict(), "vocab": vocab, "config": {...},
}, model_save_path) else: patience_counter += 1 if patience_counter >=
patience: break # Early stopping
```

Training Procedure:

1. Prepare sentiment datasets (Financial PhraseBank + news headlines)

2. Create train/val/test splits (70/15/15)

3. Build vocabulary from training texts

4. Create PyTorch datasets with preprocessing
5. Optionally load GloVe embeddings
6. Initialize model (LSTM or GRU)
7. Train for up to 50 epochs:
- Forward pass through embedding → RNN → classifier
- Compute loss and backpropagate
- Update weights with Adam optimizer
- Validate on validation set
- Adjust learning rate if needed
- Save best model based on validation loss
8. Evaluate on test set
9. Save final model checkpoint

Key Features:
- Supports both random and pre-trained embeddings
- Learning rate scheduling for better convergence
- Early stopping to prevent overfitting
- Comprehensive evaluation metrics

4.4 Evaluation & Testing

Validation Strategy:
- **Train/Val/Test Split**: 70/15/15 stratified split
- **Validation Frequency**: Every epoch
- **Metrics**: Loss and accuracy

Evaluation Metrics (`ml/sentiment_evaluation.py`):
- **Accuracy**: Overall classification accuracy
- **Precision**: Per-class and macro-averaged
- **Recall**: Per-class and macro-averaged
- **F1-Score**: Per-class, macro-averaged, and weighted
- **Confusion Matrix**: Class-wise prediction distribution
- **Per-class Accuracy**: Accuracy for each sentiment class

Testing Procedure:
1. Load best model from checkpoint
2. Run inference on test set

3. Compute all evaluation metrics

4. Print detailed evaluation report

5. Compare with baseline or other models

Evaluation Code:

```
from ml.sentiment_evaluation import evaluate_model,
print_evaluation_report # Evaluate on test set test_results =
evaluate_model(model, test_loader, device, num_classes=5)
print_evaluation_report(test_results, "Test")
```

Performance Benchmarks:

- Expected accuracy: >60% (better than random 20%)

- Macro F1-score: >0.55

- Should handle class imbalance (neutral class often dominant)

4.5 Input/Output Specifications

Input Format:

- **Shape**: `(batch_size, seq_len)` - Token indices

- **Type**: `torch.Tensor` (int64/long)

- **Sequence Length**: Up to `max_len` (default 128)

- **Padding**: Sequences shorter than `max_len` are padded with 0 (PAD token)

- **Truncation**: Sequences longer than `max_len` are truncated

Output Format:

- **Shape**: `(batch_size, 5)` - Logits for 5 classes

- **Type**: `torch.Tensor` (float32)

- **Classes**:

- Index 0: Very Negative

- Index 1: Negative

- Index 2: Neutral

- Index 3: Positive

- Index 4: Very Positive

Prediction Process:

```
# Preprocess text preprocessor = TextPreprocessor(vocab=vocab,
max_len=128) sequence = preprocessor.transform([text])[0] # List of
indices x = torch.tensor(sequence).unsqueeze(0) # (1, seq_len) # Forward
pass model.eval() with torch.no_grad(): logits = model(x) # (1, 5) probs =
torch.softmax(logits, dim=-1).cpu().numpy()[0] predicted_class =
int(np.argmax(probs)) confidence = float(probs[predicted_class])
```

Prediction Parameters:

- `max_len`: 128 (maximum sequence length)

- `vocab`: Vocabulary object (must match training)

- `num_classes`: 5

Example Usage:

```
from ml.sentiment_lstm_model import create_model from
ml.sentiment_preprocessing import TextPreprocessor # Load model checkpoint
= torch.load("models/sentiment_lstm.pth") vocab = checkpoint["vocab"]
config = checkpoint["config"] model = create_model( vocab=vocab,
embedding_dim=config["embedding_dim"], hidden_dim=config["hidden_dim"],
num_layers=config["num_layers"], num_classes=5, dropout=config["dropout"],
) model.load_state_dict(checkpoint["model_state_dict"]) model.eval() #
Predict sentiment text = "Apple stock surges on strong earnings report!"
preprocessor = TextPreprocessor(vocab=vocab, max_len=128) sequence =
preprocessor.transform([text])[0] x = torch.tensor(sequence).unsqueeze(0)
with torch.no_grad(): logits = model(x) predicted_class =
logits.argmax(dim=1).item() sentiment_labels = ["Very Negative",
"Negative", "Neutral", "Positive", "Very Positive"] print(f"Sentiment:
{sentiment_labels[predicted_class]}")
```

4.5.1 Detailed Example

Let's walk through a concrete example showing how the Sentiment LSTM model processes text and predicts sentiment:

Scenario: Classifying sentiment of the financial news headline: "Apple stock surges on strong earnings report!"

Step 1: Input Text

```
Text: "Apple stock surges on strong earnings report!"
```

Step 2: Text Preprocessing

The text goes through preprocessing pipeline:

1. **Text Cleaning**:

- Remove HTML tags (none in this case)

- Remove URLs (none)

- Keep punctuation (important for sentiment)

- Normalize whitespace

- Result: `"Apple stock surges on strong earnings report!"`

2. **Tokenization**:

```

Tokens: ["apple", "stock", "surges", "on", "strong", "earnings", "report", "!"]

```

3. **Vocabulary Lookup**:

Assume vocabulary mappings (example):

```
Vocabulary:
<PAD>: 0
<UNK>: 1
"apple": 42
"stock": 156
"surges": 892
"on": 15
"strong": 234
"earnings": 567
"report": 189
"!": 23
```

4. **Sequence Encoding**:

```python
sequence = [42, 156, 892, 15, 234, 567, 189, 23]
# Length: 8 tokens
```

5. **Padding/Truncation** (max_len=128):

```python
# Pad to max_len=128
padded_sequence = [42, 156, 892, 15, 234, 567, 189, 23] + [0] * 120
# Shape: (128,)
```

Step 3: Model Forward Pass

The sequence is processed through the LSTM network:

Architecture (embedding_dim=100, hidden_dim=128, num_layers=2):

1. **Embedding Layer**: `(128,) → (128, 100)`

```
embedded = Embedding(vocab_size, 100)(padded_sequence)
# Shape: (128, 100)
```

Each token index → 100-dimensional embedding vector
```

**Example Embeddings** (simplified):
```
Token "apple" (idx=42) → [0.12, -0.34, 0.56, ..., 0.23] (100 dims)
Token "surges" (idx=892) → [0.45, 0.12, -0.67, ..., 0.89] (100 dims)
Token "strong" (idx=234) → [0.23, 0.45, 0.12, ..., -0.34] (100 dims)
...
```

2. **LSTM Layers** (2 layers):

**Layer 1**:
```
lstm_out_1, (h1, c1) = LSTM(embedded)
Input: (128, 100)
Output: (128, 128) - hidden states for each time step
Hidden: (1, 128) - final hidden state
Cell: (1, 128) - final cell state
```

**Layer 2**:
```
lstm_out_2, (h2, c2) = LSTM(lstm_out_1)
Input: (128, 128)
Output: (128, 128)
Hidden: (1, 128) - final hidden state
```

3. **Sequence Aggregation**: Take last time step
```
last_hidden = lstm_out_2[-1, :] # Shape: (128,)
# This represents the entire sequence's meaning
```

4. **Classification Head**:
```

```
dropped = Dropout(0.5)(last_hidden) # Shape: (128,)
logits = Linear(128 → 5)(dropped) # Shape: (5,)
```

LSTM Processing Visualization:

```
Time Step 0: "apple" → h0, c0 Time Step 1: "stock" → h1, c1 (depends on
h0, c0) Time Step 2: "surges" → h2, c2 (depends on h1, c1) Time Step 3:
"on" → h3, c3 (depends on h2, c2) Time Step 4: "strong" → h4, c4 (depends
on h3, c3) Time Step 5: "earnings" → h5, c5 (depends on h4, c4) Time Step
6: "report" → h6, c6 (depends on h5, c5) Time Step 7: "!" → h7, c7
(depends on h6, c6) Time Steps 8-127: <PAD> → h8...h127 (mostly
unchanged) Final hidden state h7 captures the sentiment of the entire
sequence.
```

Step 4: Output Generation

Example Output Logits:

```
logits = np.array([-2.1, -0.8, 0.3, 1.5, 2.8]) # [Very Negative, Negative,
Neutral, Positive, Very Positive]
```

Probability Calculation:

```
probs = softmax(logits) probs = np.array([0.02, 0.08, 0.15, 0.25, 0.50]) #
[Very Negative: 2%, Negative: 8%, Neutral: 15%, Positive: 25%, Very
Positive: 50%]
```

Prediction:

```
predicted_class = np.argmax(probs) # = 4 (Very Positive) confidence =
probs[predicted_class] # = 0.50 (50%)
```

Step 5: Interpretation

The model predicts **"Very Positive"** sentiment with 50% confidence. This makes sense because:

- "surges" indicates strong upward movement
- "strong earnings" is positive financial news
- "!" adds emphasis
- Overall tone is very bullish

Alternative Scenario - Negative Sentiment:

If the text was: "Apple stock crashes after disappointing earnings report"

Preprocessing:

```
Tokens: ["apple", "stock", "crashes", "after", "disappointing",
"earnings", "report"] Sequence: [42, 156, 1203, 89, 456, 567, 189] +
[0]*121
```

Example Output:

```
logits = np.array([2.5, 1.2, 0.1, -0.5, -1.8]) probs = softmax(logits) =
[0.45, 0.25, 0.15, 0.10, 0.05] predicted_class = 0 # Very Negative
confidence = 0.45 # 45%
```

Key Insights from the Example:

1. **Sequential Processing**: LSTM processes tokens sequentially, building up understanding as it reads.

2. **Context Preservation**: Hidden states carry information from earlier tokens, allowing the model to understand context (e.g., "surges" is positive in financial context).

3. **Word Order Matters**: The sequence order affects the final representation - "stock surges" vs "surges stock" would have different meanings.

4. **Embedding Quality**: Pre-trained embeddings (like GloVe) can improve performance by providing semantic relationships between words.

5. **Padding Handling**: Padding tokens don't contribute to final prediction but allow batch processing.

6. **Multi-class Output**: Five classes provide fine-grained sentiment analysis beyond simple positive/negative.

4.6 Fine-tuning Procedures

Fine-tuning Approach:

1. **Load Pre-trained Model**:

```python
checkpoint = torch.load("models/sentiment_lstm.pth")
vocab = checkpoint["vocab"]
config = checkpoint["config"]
model = create_model(vocab=vocab, **config)
model.load_state_dict(checkpoint["model_state_dict"])
```

2. **Continue Training**:

- Use lower learning rate (1e-4)
- Train on new domain-specific data
- Monitor validation metrics
- Apply early stopping

3. **Transfer Learning with Pre-trained Embeddings**:

- Load GloVe embeddings
- Initialize embedding layer with GloVe
- Optionally freeze embeddings

- Fine-tune RNN and classifier layers

Hyperparameter Adjustments:

- **Learning Rate**: Reduce to 1e-4 for fine-tuning

- **Batch Size**: Keep same or reduce

- **Epochs**: Fewer epochs needed (10-20)

- **Dropout**: May increase to 0.6 for regularization

Best Practices:

- Always validate on held-out set

- Monitor per-class metrics (handle imbalance)

- Use learning rate scheduling

- Compare with baseline before/after fine-tuning

- Consider class weights if severe imbalance

Using Pre-trained Embeddings:

```
from ml.sentiment_preprocessing import load_glove_embeddings,
create_embedding_matrix # Load GloVe embeddings glove_embeddings =
load_glove_embeddings("glove.6B.100d.txt", vocab, embedding_dim=100)
embedding_matrix = create_embedding_matrix(vocab, glove_embeddings,
embedding_dim=100) # Create model with pre-trained embeddings model =
create_model( vocab=vocab, embedding_dim=100,
embedding_matrix=embedding_matrix, freeze_embeddings=False, # Set True to
freeze )
```

4.7 Model Saving & Loading

Checkpoint Format:

```
checkpoint = { "model_state_dict": model.state_dict(), "vocab": vocab, #
Vocabulary object "embedding_type": "random" or "glove", "embedding_dim":
100, "hidden_dim": 128, "num_layers": 2, "num_classes": 5, "dropout": 0.5,
"use_gru": False, "config": { "max_len": 128, "embedding_dim": 100,
"hidden_dim": 128, "num_layers": 2, "num_classes": 5, "dropout": 0.5,
"use_gru": False, }, } torch.save(checkpoint, "models/sentiment_lstm.pth")
```

Model File Location:

- **Path**: `models/sentiment_lstm.pth`

- **Format**: PyTorch checkpoint (.pth)

Loading Procedure:

```
checkpoint = torch.load("models/sentiment_lstm.pth", map_location=device,
weights_only=False) vocab = checkpoint["vocab"] config =
checkpoint["config"] model = create_model( vocab=vocab,
embedding_dim=config["embedding_dim"], hidden_dim=config["hidden_dim"],
num_layers=config["num_layers"], num_classes=config["num_classes"],
dropout=config["dropout"], use_gru=config.get("use_gru", False), )
model.load_state_dict(checkpoint["model_state_dict"]) model.eval()
```

Important Notes:

- Vocabulary must be saved and loaded with model

- Preprocessor must use same vocabulary

- Max sequence length should match training

5. Sentiment BERT Model

5.1 Model Architecture

File: `ml/sentiment_bert_model.py`

The Sentiment BERT model fine-tunes a pre-trained BERT model for financial sentiment classification.

Architecture:

```
Input: Text strings ↓ BERT Tokenizer: Tokenize and encode text ■■ Input
IDs: (batch_size, max_length) ■■ Attention Mask: (batch_size, max_length)
↓ BERT Encoder: Pre-trained BERT model ■■ Embedding Layer ■■ 12
Transformer Encoder Layers (bert-base-uncased) ■ ■■ Multi-Head
Self-Attention ■ ■■ Feed-Forward Network ■ ■■ Layer Normalization ■■
Pooler Layer ↓ [CLS] Token Representation: (batch_size, hidden_size=768) ↓
Classification Head: Linear(hidden_size → num_classes) ↓ Output:
(batch_size, 5) - Logits for 5 classes
```

Key Components:

1. **BERT Tokenizer**:

- WordPiece tokenization

- Adds special tokens: `[CLS]`, `[SEP]`, `[PAD]`

- Max length: 128 (default)

- Returns `input_ids` and `attention_mask`

2. **BERT Encoder**:

- Pre-trained BERT model (default: `bert-base-uncased`)

- 12 transformer encoder layers

- Hidden size: 768

- Attention heads: 12

- Total parameters: ~110M (pre-trained)

3. **Classification Head**:

- Takes `[CLS]` token representation

- Linear projection to 5 classes

- Dropout for regularization

Model Variants:

- **bert-base-uncased**: Default (110M parameters)

- **yiyanghkust/finbert-pretrain**: Financial domain BERT (recommended for financial text)

Code Structure:

```
class SentimentBERT(nn.Module): def __init__( self, model_name: str =
"bert-base-uncased", num_classes: int = 5, dropout: float = 0.1, ):
super().__init__() self.model =
AutoModelForSequenceClassification.from_pretrained( model_name,
num_labels=num_classes, hidden_dropout_prob=dropout,
attention_probs_dropout_prob=dropout, )
```

5.2 Data Preparation

Input Data Format:

- **Text Data**: Raw text strings (financial news, sentences)

- **Labels**: Integer labels (0-4) for 5-class sentiment

Label Encoding (same as LSTM):

- Class 0: Very Negative

- Class 1: Negative

- Class 2: Neutral

- Class 3: Positive

- Class 4: Very Positive

Preprocessing:

- **Minimal preprocessing** - BERT tokenizer handles most preprocessing

- Text is passed directly to tokenizer

- Tokenizer handles:

- Lowercasing (for uncased models)

- WordPiece tokenization

- Special token addition

- Padding and truncation

Dataset Structure (`SentimentBERTDataset`):

- **Input**: Dictionary with:

- `input_ids`: `(batch_size, max_length)` - Token IDs

- `attention_mask`: `(batch_size, max_length)` - Attention mask

- `labels`: `(batch_size,)` - Class labels

Data Splitting:

- Same as LSTM: 70/15/15 train/val/test split

Code Example:

```
from ml.sentiment_bert_model import SentimentBERTDataset from transformers
import AutoTokenizer tokenizer =
AutoTokenizer.from_pretrained("bert-base-uncased") train_dataset =
SentimentBERTDataset( texts=train_df["text"].tolist(),
labels=train_df["label"].tolist(), tokenizer=tokenizer, max_length=128, )
```

5.3 Training Process

Training Script: `ml/sentiment_bert_model.py` (function: `train_bert_model`)

Hyperparameters:

- **Epochs**: 10 (typically sufficient for fine-tuning)

- **Batch Size**: 16 (smaller due to BERT's memory requirements)

- **Learning Rate**: 2e-5 (lower than LSTM - standard for BERT fine-tuning)

- **Optimizer**: AdamW (with weight decay)

- **Weight Decay**: 0.01

- **Loss Function**: CrossEntropyLoss (built into model)

- **Early Stopping Patience**: 3 epochs

- **Max Length**: 128 tokens

- **Mixed Precision**: FP16 (if CUDA available)

Training Arguments (Hugging Face Trainer):

```
training_args = TrainingArguments( output_dir=output_dir,
num_train_epochs=epochs, per_device_train_batch_size=batch_size,
per_device_eval_batch_size=batch_size, learning_rate=learning_rate,
weight_decay=0.01, logging_dir=f"{output_dir}/logs", logging_steps=50,
eval_strategy="epoch", save_strategy="epoch", load_best_model_at_end=True,
metric_for_best_model="eval_loss", greater_is_better=False,
save_total_limit=2, fp16=torch.cuda.is_available(), )
```

Training Loop (Hugging Face Trainer):

```
from transformers import Trainer, EarlyStoppingCallback trainer = Trainer(
model=model.model, # Underlying BERT model args=training_args,
train_dataset=train_dataset, eval_dataset=val_dataset,
compute_metrics=compute_metrics,
callbacks=[EarlyStoppingCallback(early_stopping_patience=patience)], ) #
Train train_result = trainer.train() # Evaluate val_results =
trainer.evaluate() test_results = trainer.evaluate(test_dataset)
```

Training Procedure:

1. Load pre-trained BERT model and tokenizer

2. Prepare datasets (same as LSTM)

3. Create `SentimentBERTDataset` with tokenization

4. Initialize `AutoModelForSequenceClassification`

5. Set up Hugging Face `Trainer` with:

- Training arguments

- Compute metrics function

- Early stopping callback

6. Train model (fine-tune BERT)

7. Evaluate on validation and test sets

8. Save final model and tokenizer

Key Features:

- Uses Hugging Face Transformers library

- Leverages pre-trained BERT weights

- Fine-tuning approach (not training from scratch)

- Automatic mixed precision training

- Built-in evaluation and checkpointing

Compute Metrics Function:

```
def compute_metrics(eval_pred): predictions, labels = eval_pred
predictions = np.argmax(predictions, axis=1) from sklearn.metrics import
accuracy_score, f1_score accuracy = accuracy_score(labels, predictions) f1
= f1_score(labels, predictions, average="macro") return { "accuracy":
accuracy, "f1": f1, }
```

5.4 Evaluation & Testing

Validation Strategy:

- **Train/Val/Test Split**: 70/15/15

- **Validation Frequency**: Every epoch

- **Metrics**: Loss, accuracy, F1-score

Evaluation Metrics:

- **Loss**: Cross-entropy loss

- **Accuracy**: Overall classification accuracy

- **F1-Score**: Macro-averaged F1-score

- Can compute additional metrics (precision, recall, confusion matrix)

Testing Procedure:

1. Load fine-tuned model

2. Run `trainer.evaluate()` on test dataset

3. Compute detailed metrics

4. Compare with LSTM baseline

Performance Benchmarks:

- Expected accuracy: >70% (better than LSTM)

- F1-score: >0.65

- Should outperform LSTM due to pre-trained knowledge

- Financial BERT (`finbert-pretrain`) may perform even better

5.5 Input/Output Specifications

Input Format:

- **Raw Text**: String (e.g., "Apple stock surges on strong earnings report!")

- **Tokenization**: Handled by BERT tokenizer

- **After Tokenization**:

- `input_ids`: `(batch_size, max_length)` - Token indices

- `attention_mask`: `(batch_size, max_length)` - Mask for padding

Output Format:

- **Shape**: `(batch_size, 5)` - Logits for 5 classes

- **Type**: `torch.Tensor` (float32)

- **Classes**: Same as LSTM (0-4: Very Negative to Very Positive)

Prediction Process:

```
from transformers import AutoTokenizer, AutoModelForSequenceClassification
# Load model and tokenizer model_path =
"models/bert_sentiment/final_model" tokenizer =
AutoTokenizer.from_pretrained(model_path) model =
AutoModelForSequenceClassification.from_pretrained(model_path)
model.eval() # Predict text = "Apple stock surges on strong earnings
report!" encoding = tokenizer( text, truncation=True,
padding="max_length", max_length=128, return_tensors="pt", ) with
torch.no_grad(): outputs = model( input_ids=encoding["input_ids"],
attention_mask=encoding["attention_mask"], ) logits = outputs.logits
predicted_class = logits.argmax(dim=1).item() probs =
torch.softmax(logits, dim=-1)[0]
```

Prediction Parameters:

- `max_length`: 128 (maximum sequence length)

- `model_name`: "bert-base-uncased" or "yiyanghkust/finbert-pretrain"

- `num_classes`: 5

Example Usage:

```
from ml.sentiment_bert_model import load_bert_model, predict_with_bert #
Load model model, tokenizer =
load_bert_model("models/bert_sentiment/final_model") # Predict texts = [
"Apple stock surges on strong earnings report!", "Company faces challenges
in competitive market.", ] predictions = predict_with_bert(model,
```

```
tokenizer, texts) sentiment_labels = ["Very Negative", "Negative",
"Neutral", "Positive", "Very Positive"] for text, pred in zip(texts,
predictions): print(f"{text} -> {sentiment_labels[pred]}")
```

5.5.1 Detailed Example

Let's walk through a concrete example showing how the Sentiment BERT model processes text using its pre-trained knowledge:

Scenario: Classifying sentiment of the financial news headline: "Apple stock surges on strong earnings report!"

Step 1: Input Text

```
Text: "Apple stock surges on strong earnings report!"
```

Step 2: BERT Tokenization

BERT uses WordPiece tokenization, which splits words into subwords:

1. **Add Special Tokens**:

```

"[CLS] Apple stock surges on strong earnings report! [SEP]"

```

- `[CLS]`: Classification token (used for final prediction)

- `[SEP]`: Separator token

2. **WordPiece Tokenization**:

```

Tokens: ["[CLS]", "apple", "stock", "sur", "##ges", "on", "strong", "earn", "##ings", "report", "!", "[SEP]"]

```

- "surges" → "sur" + "##ges" (subword tokens)

- "earnings" → "earn" + "##ings" (subword tokens)

- "##" indicates continuation of previous token

3. **Convert to IDs**:

```python

input_ids = [101, 6207, 3466, 2791, 10047, 2006, 2607, 4013, 10047, 3466, 999, 102]

# [CLS]=101, apple=6207, stock=3466, sur=2791, ##ges=10047, ..., [SEP]=102

```

4. **Create Attention Mask**:

```python
attention_mask = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1] + [0] * 116
# 1 = real token, 0 = padding
```

5. **Padding to max_length=128**:

```python
input_ids = input_ids + [0] * 116 # Pad with 0 (PAD token)
# Shape: (128,)
```

Step 3: BERT Encoder Forward Pass

The tokenized input passes through pre-trained BERT:

Architecture (bert-base-uncased: 12 layers, 768 hidden size, 12 attention heads):

1. **Embedding Layer**:

```
Token Embeddings: (128, 768)
Position Embeddings: (128, 768)
Segment Embeddings: (128, 768) # All 0s for single sentence
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
Combined: (128, 768)
```

2. **Transformer Encoder Layers** (12 layers):

Layer 1-12 (each layer):

- **Multi-head Self-Attention** (12 heads):

```
Query, Key, Value: (128, 768) → (128, 768)
Attention scores: (128, 128) - attention weights between all token pairs
Attention output: (128, 768)
```

- **Feed-Forward Network**:

```
FFN: (128, 768) → (128, 3072) → (128, 768)
```

- **Residual Connections + LayerNorm**

3. **Pooler Layer** (takes [CLS] token):

```

cls_representation = pooler(encoder_output[0, :]) # Shape: (768,)

```

Attention Visualization (conceptual):

The [CLS] token attends to important words:

```
[CLS] attends to: "surges": 0.18 (strong positive signal) "strong": 0.15 (positive modifier) "earnings": 0.12 (financial context) "report": 0.10 (news context) "stock": 0.08 (market context) "apple": 0.07 (entity) ... (other tokens with lower attention)
```

Step 4: Classification Head

```
logits = Linear(768 → 5)(cls_representation) # Shape: (5,)
```

Step 5: Output Generation

Example Output Logits:

```
logits = np.array([-3.2, -1.5, 0.2, 2.1, 3.8]) # [Very Negative, Negative, Neutral, Positive, Very Positive]
```

Probability Calculation:

```
probs = softmax(logits) probs = np.array([0.01, 0.05, 0.12, 0.28, 0.54]) # [Very Negative: 1%, Negative: 5%, Neutral: 12%, Positive: 28%, Very Positive: 54%]
```

Prediction:

```
predicted_class = np.argmax(probs) # = 4 (Very Positive) confidence = probs[predicted_class] # = 0.54 (54%)
```

Step 6: Interpretation

The model predicts **"Very Positive"** sentiment with 54% confidence. BERT's advantages:

1. **Pre-trained Knowledge**: BERT understands that "surges" + "strong earnings" = very positive in financial context

2. **Context Understanding**: BERT considers the entire sentence, not just individual words

3. **Subword Handling**: WordPiece tokenization handles out-of-vocabulary words better

4. **Bidirectional**: BERT reads both left-to-right and right-to-left, capturing full context

Comparison with LSTM:

| Aspect | LSTM | BERT |

|--------|------|------|

| **Processing** | Sequential (left-to-right) | Bidirectional (full context) |

| **Embeddings** | Learned from scratch | Pre-trained on massive corpus |

| **Context** | Limited by hidden state | Full attention to all tokens |

| **Performance** | ~60-65% accuracy | ~70-75% accuracy |

Alternative Scenario - Complex Sentiment:

Text: "Apple stock initially surges but then falls on earnings miss"

BERT Tokenization:

```
Tokens: ["[CLS]", "apple", "stock", "initially", "sur", "##ges", "but",
"then", "falls", "on", "earn", "##ings", "miss", "[SEP]"]
```

BERT's Advantage:

- Understands contrast ("surges" vs "falls")

- Recognizes "earnings miss" as negative

- Weighs "falls" and "miss" more heavily than "surges"

- Final prediction: **Negative** or **Very Negative**

Key Insights from the Example:

1. **Pre-trained Knowledge**: BERT brings knowledge from pre-training on billions of words, understanding financial terminology.

2. **Bidirectional Context**: Unlike LSTM, BERT sees the entire sentence simultaneously, understanding relationships between distant words.

3. **Attention Mechanism**: Self-attention allows BERT to focus on important words (e.g., "surges", "strong") while downplaying less important ones.

4. **Subword Tokenization**: WordPiece handles rare words and typos better than word-level tokenization.

5. **Transfer Learning**: Fine-tuning leverages pre-trained weights, requiring less data and training time than training from scratch.

6. **Domain Adaptation**: Financial BERT (`finbert-pretrain`) performs even better on financial text due to domain-specific pre-training.

5.6 Fine-tuning Procedures

Fine-tuning Approach:

BERT fine-tuning is the primary training method (not transfer learning from another task). However, you can fine-tune further:

1. **Domain-Specific Fine-tuning**:

- Start with `yiyanghkust/finbert-pretrain` (financial BERT)

- Fine-tune on your specific financial dataset

- Use lower learning rate (1e-5)

2. **Continual Fine-tuning**:

- Load previously fine-tuned model

- Continue training on new data

- Use very low learning rate (5e-6)

3. **Layer-wise Fine-tuning**:

- Freeze early BERT layers

- Fine-tune only last N layers

- Fine-tune classification head

Hyperparameter Adjustments:

- **Learning Rate**: 1e-5 to 5e-6 for further fine-tuning

- **Epochs**: 3-5 epochs typically sufficient

- **Batch Size**: Keep small (8-16) due to memory

- **Weight Decay**: 0.01 (standard)

Best Practices:

- Use financial BERT (`finbert-pretrain`) for financial text

- Monitor for overfitting (BERT can overfit quickly)

- Use early stopping aggressively

- Validate on held-out set

- Compare with LSTM baseline

Example: Fine-tuning Financial BERT:

```
from ml.sentiment_bert_model import train_bert_model results =
train_bert_model( train_dataset=train_dataset, val_dataset=val_dataset,
test_dataset=test_dataset, model_name="yiyanghkust/finbert-pretrain", #
Financial BERT num_classes=5, batch_size=16, learning_rate=2e-5,
epochs=10, max_length=128, output_dir="models/bert_sentiment_financial",
patience=3, )
```

5.7 Model Saving & Loading

Checkpoint Format (Hugging Face):

- Model and tokenizer saved in separate files

- Standard Hugging Face format:

- `config.json`: Model configuration

- `pytorch_model.bin`: Model weights

- `tokenizer_config.json`: Tokenizer configuration

- `vocab.txt`: Vocabulary file

Model File Location:

- **Path**: `models/bert_sentiment/final_model/`

- **Format**: Hugging Face model directory

Saving Procedure:

```
# Trainer automatically saves during training
trainer.save_model(f"{output_dir}/final_model")
tokenizer.save_pretrained(f"{output_dir}/final_model")
```

Loading Procedure:

```
from transformers import AutoTokenizer, AutoModelForSequenceClassification
model_path = "models/bert_sentiment/final_model" tokenizer =
AutoTokenizer.from_pretrained(model_path) model =
AutoModelForSequenceClassification.from_pretrained(model_path)
model.eval()
```

Usage in Production:

```
from ml.sentiment_bert_model import load_bert_model, predict_with_bert
model, tokenizer = load_bert_model("models/bert_sentiment/final_model") #
Single prediction text = "Strong earnings beat expectations" encoding =
tokenizer(text, return_tensors="pt", truncation=True,
padding="max_length", max_length=128) with torch.no_grad(): outputs =
model(**encoding) predicted_class = outputs.logits.argmax(dim=1).item()
```

Summary and Comparison

Model Comparison Table

Model	Type	Input Format	Output	Parameters	Training Time	Best Use Case
Baseline	Rule-based	Price series	Direction + Prices	0	Instant	Simple baseline
MLP	Feedforward NN	Tabular (8 features)	3 classes	~5K	Fast (< 1 min)	Quick predictions
Transformer	Transformer Encoder	Sequence (30×8)	3 classes	~100K	Medium (5-10 min)	Temporal patterns
Sentiment LSTM	LSTM/GRU	Text sequences	5 classes	~750K	Medium (10-20 min)	Text sentiment
Sentiment BERT	Fine-tuned BERT	Text strings	5 classes	~110M	Slow (30-60 min)	Best text accuracy

Training Summary

Stock Prediction Models (MLP, Transformer):

- **Data**: Historical stock prices + sentiment + fundamentals

- **Task**: Predict future price direction (up/down/flat)

- **Evaluation**: Classification accuracy, loss

- **Best Model**: Transformer (better temporal understanding)

Sentiment Analysis Models (LSTM, BERT):

- **Data**: Financial text (news, phrases)

- **Task**: Classify sentiment (5 classes)

- **Evaluation**: Accuracy, F1-score, per-class metrics

- **Best Model**: BERT (pre-trained knowledge, better accuracy)

Key Takeaways

1. **Baseline Model**: Simple rule-based, no training needed, serves as comparison baseline

2. **MLP Model**: Fast training, good for quick predictions, limited to tabular features

3. **Transformer Model**: Better for temporal patterns, requires more data and computation

4. **Sentiment LSTM**: Good balance of speed and accuracy for text classification

5. **Sentiment BERT**: Best accuracy for text, requires more resources but leverages pre-trained knowledge

Recommendations

- **For Stock Prediction**: Use Transformer model for better temporal understanding

- **For Sentiment Analysis**: Use BERT (especially financial BERT) for best accuracy

- **For Quick Prototyping**: Use MLP or LSTM for faster iteration

- **For Production**: Consider ensemble of multiple models

Appendix: File Locations

Model Files

- `ml/baseline_model.py` - Baseline model implementation
- `ml/lab2_mlp_model.py` - MLP architecture
- `ml/train_mlp_model.py` - MLP training script
- `ml/lab5_transformer_model.py` - Transformer architecture
- `ml/train_transformer_model.py` - Transformer training script
- `ml/sentiment_lstm_model.py` - LSTM architecture
- `ml/train_sentiment_model.py` - LSTM training script
- `ml/sentiment_bert_model.py` - BERT architecture and training
- `ml/sentiment_evaluation.py` - Evaluation metrics
- `ml/sentiment_data.py` - Data preparation
- `ml/sentiment_preprocessing.py` - Text preprocessing
- `ml/sentiment_dataset.py` - Dataset classes

Configuration Files

- `domain/configs.py` - Model configurations (MLPConfig, TransformerConfig)
- `domain/predictions.py` - Input/output data structures

Saved Models

- `models/stock_mlp.pth` - Trained MLP weights
- `models/stock_transformer.pth` - Trained Transformer weights
- `models/sentiment_lstm.pth` - Trained LSTM weights
- `models/bert_sentiment/final_model/` - Fine-tuned BERT model

End of Document