시계열 1조

비타민 12 & 13기 학기 프로젝트

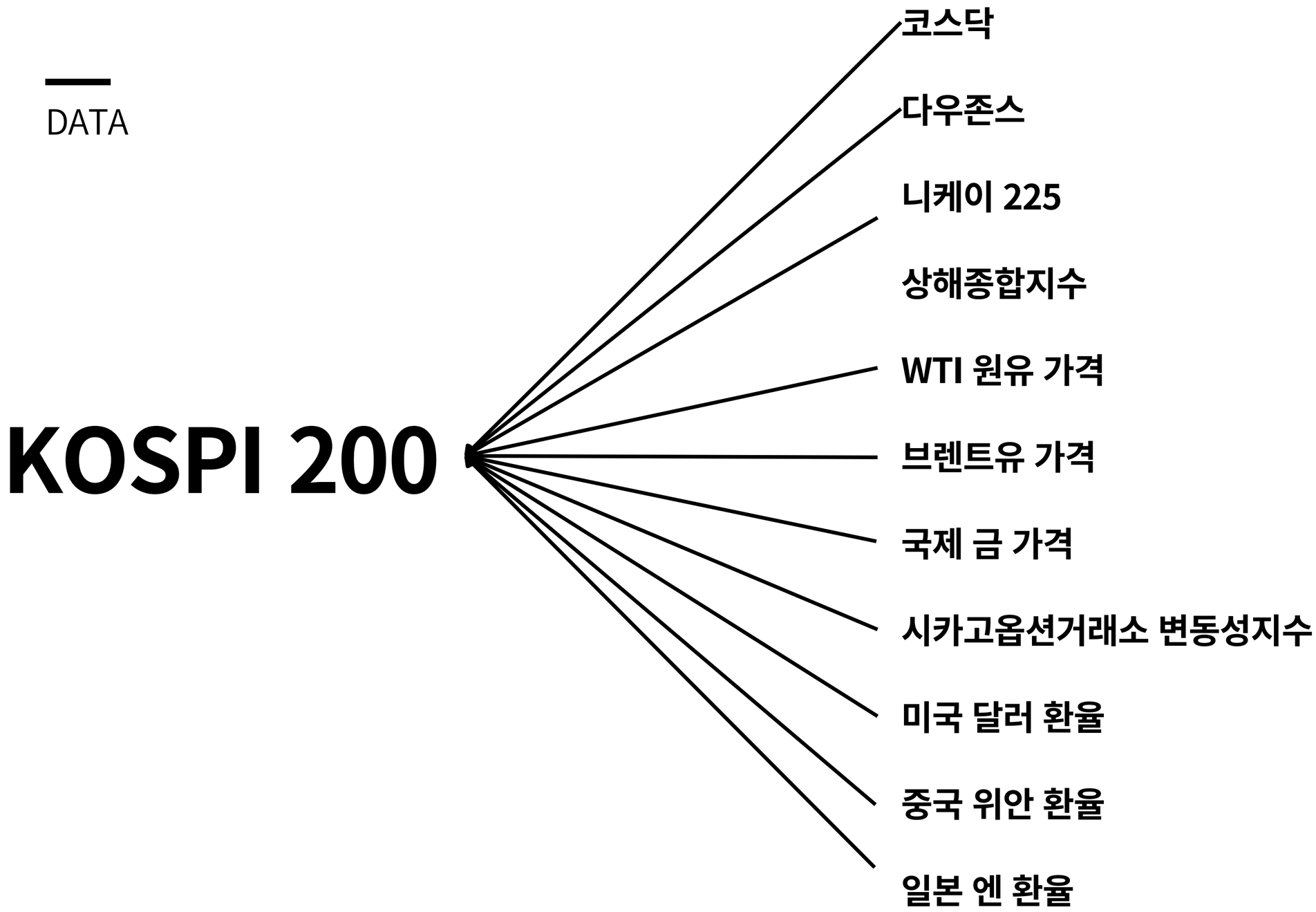
피쳐 중요도 분석을 통한 KOSPI200 지수 예측

비타민 학기 프로젝트 최종 발표 목차

피쳐 중요도 분석을 통한 KOSPI 지수 예측

데이터 소개

DATA


KOSPI 200
코스닥
다우존스
니케이 225
상해종합지수
WTI 원유 가격
브렌트유 가격
국제 금 가격
시카고옵션거래소 변동성지수
미국 달러 환율
중국 위안 환율
일본 엔 환율

데이터 전처리

ex) 다우존스의 5월 15일 종가는 우리나라 시간으로 5월 16일

-> 날짜를 하루 앞당겨서 사용

```
# 날짜 하루씩 조정
dow['날짜']= pd.to_datetime(dow['날짜'])
dow['edit_날짜']= dow['날짜'] + pd.Timedelta(days=1)
dow.head()
```

날짜를 하루 앞당긴 변수 : 다우존스, 원/달러 환율, WTI 원유 가격, 브렌트유 가격, 국제 금 가격, VIX 지수

KOSPI 200을 기준으로 데이터프레임 병합

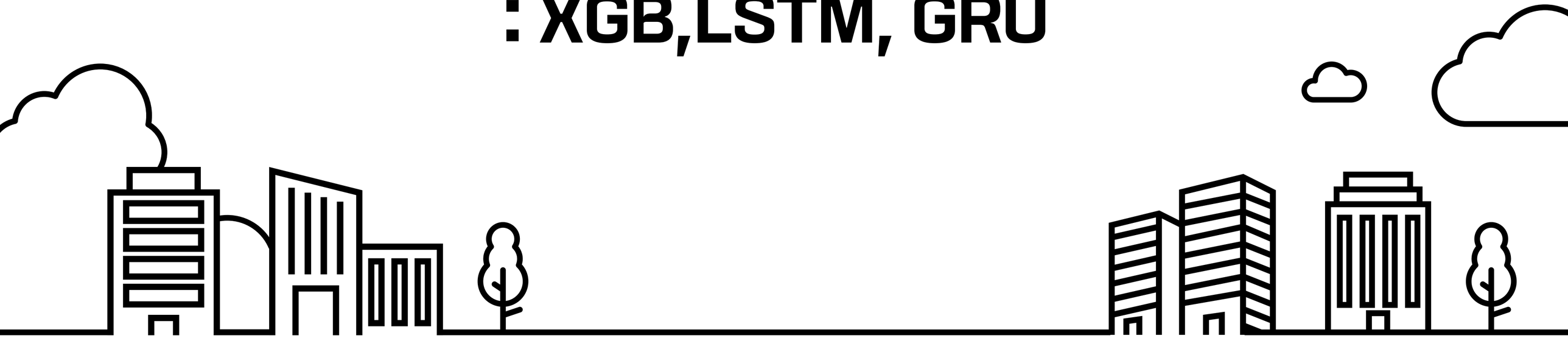
```
# kospi200을 기준으로 나머지 데이터프레임을 병합
from functools import reduce
lst=[kosdaq, dow, nikkei, ssec,
usd_krw, cny_krw, jpy_krw,
wti, brent, gold, vix]
df = reduce(lambda left, right: pd.merge(left, right, on='날

# 결과 출력
df.head()
```

```
1   kospi200   1739 non-null   float64
2   kosdaq     1738 non-null   object
3   dow        1347 non-null   object
4   nikkei     1646 non-null   object
5   ssec       1659 non-null   object
6   usd_krw    1401 non-null   object
7   cny_krw    1739 non-null   float64
8   jpy_krw    1739 non-null   float64
9   wti        1706 non-null   float64
10  brent      1391 non-null   float64
11  gold       1400 non-null   object
12  vix        1365 non-null   float64
```

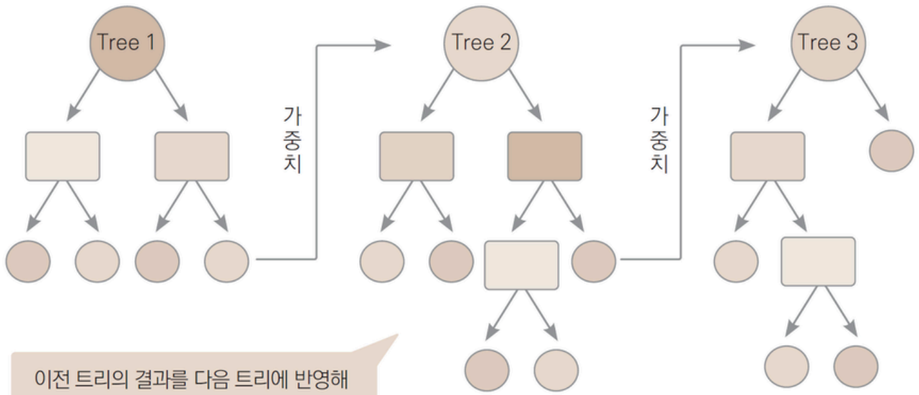
-> df.fillna(method='ffill', inplace=True)로 결측치 처리

피쳐 중요도 분석을 통한 KOSPI 지수 예측

모델 소개
: XGB,LSTM, GRU



XGBoost



트리 기반의 앙상블 학습 알고리즘

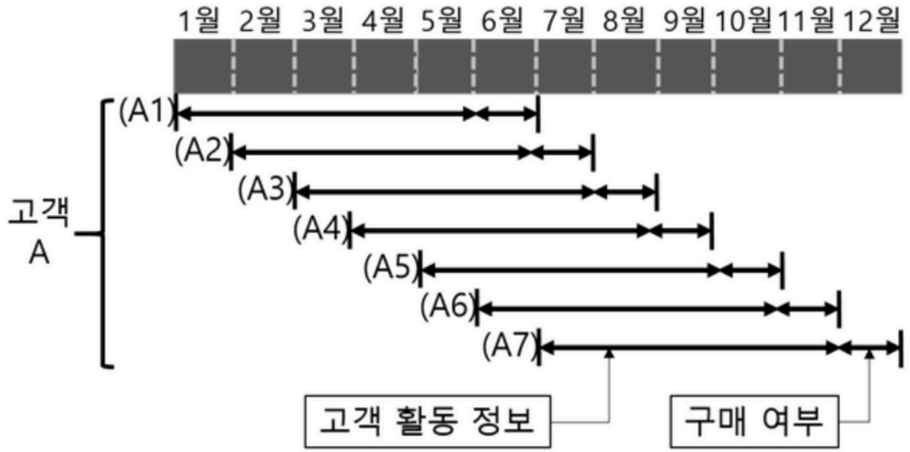
기본 학습기를 의사결정나무 (Decision Tree)로 하여
그래디언트 부스팅과 같이 그래디언트를 이용해 이전 모형의
약점을 보완하는 방식으로 학습하는 모델.

본 프로젝트에서는 이진분류 모델을 구현했으므로
손실함수로 ‘binary: kogistic’ 사용함

»

병렬처리가 가능해
빠른 속도로 뛰어난 예측 성능 보임.

슬라이딩 윈도우 방법론



»

그림처럼, 1명의 고객으로 7개의 분석용 데이터를 얻을 수 있음.
A라는 동일한 사람이 A1, A2, A3..으로 시간 차를 두고
복제됨으로써 기존에 1,000개의 정보가 있었다면
슬라이딩 윈도우 기법을 통해 7,000건의 고객 정보를 확보!

데이터의 시간적 순서를 유지하며 동시에
과거 데이터를 기반으로 미래를 예측하는 효과적 방법!

```python
# 슬라이딩 윈도우 데이터 생성 함수
def create_window_data(data, window_size=5):
    X = []
    y = []
    for i in range(len(data) - window_size):
        X.append(data.iloc[i:(i + window_size), :-1].values)
        y.append(data.iloc[i + window_size, -1])
    return np.array(X), np.array(y)
```

누적된 과거 데이터를 윈도우로 정의
-> 윈도우를 일정 간격으로 이동시키면서 다음 시점의 값을 예측.

(✔) **시간 종속성을 유지하며 데이터의 패턴 반영해**
모델이 더 많은 정보를 기반으로 학습, 예측 가능

(✔) **윈도우 크기만큼 데이터의 피쳐가 증가함**

XGBoost 구축_ 종가 데이터

```
# X,y 지정
X,y= create_window_data(df, window_size=100)

# 데이터 분할
X_train, X_test, y_train, y_test= train_test_split(X,y, test_size=0.2, random_state=156)
```

```
# 모델 생성 및 훈련
model= xgb.XGBClassifier(learning_rate=0.05, max_depth=5, n_estimators=200,objective='binary:logistic', eval_metric='logloss')
model.fit(X_train.reshape(X_train.shape[0],-1), y_train)

# 예측
y_pred = model.predict(X_test.reshape(X_test.shape[0],-1))
y_proba= model.predict_proba(X_test.reshape(X_test.shape[0],-1))[:,1]

# 평가
# 정확도 계산
accuracy= accuracy_score(y_test, y_pred)
print(f'Accuracy : {accuracy:.4f}')
```

```
Accuracy : 0.5518
```

(✔) 하이퍼 파라미터 튜닝 결과)
learning_rate=0.05, max_depth=5, n_estimators=200, window_size=100

(✔) 0.5518의 accuracy를 보임

XGBoost 구축_ 변화율 데이터

```
# X,y 지정
X,y= create_window_data(df_scaled, window_size=10)

# 데이터 분할
X_train, X_test, y_train, y_test= train_test_split(X,y, test_size=0.2, random_state=156)
```

```
# 모델 생성 및 훈련
model= xgb.XGBClassifier(learning_rate=0.05, max_depth=5, n_estimators=100,objective='binary:logistic', eval_metric='logloss')
model.fit(X_train.reshape(X_train.shape[0],-1), y_train)

# 예측
y_pred = model.predict(X_test.reshape(X_test.shape[0],-1))
y_proba= model.predict_proba(X_test.reshape(X_test.shape[0],-1))[:,1]

# 평가
# 정확도 계산
accuracy= accuracy_score(y_test, y_pred)
print(f'Accuracy : {accuracy:.4f}')
```

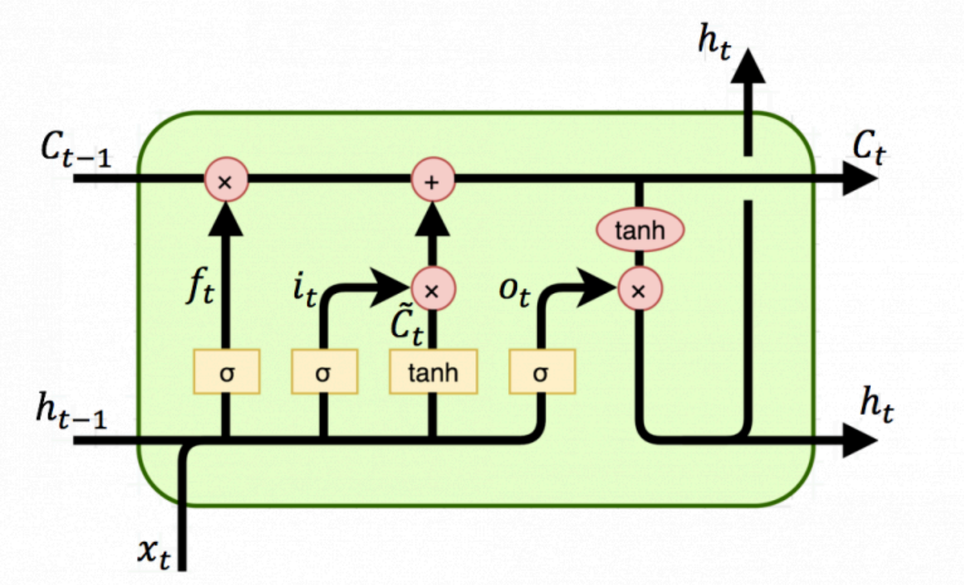
```
Accuracy : 0.6098
```

(✔) 하이퍼 파라미터 튜닝 결과)
learning_rate=0.05, max_depth=5, n_estimators=100

(✔) 0.6098의 accuracy를 보임

(✔) 변화율 데이터를 이용한 모델이 0.05 정도의 성능 개선을 보임

LSTM



긴 시간동안 RNN의 메모리를 유지

LSTM은 Cell state와 Gate 메커니즘을 통해 긴 시퀀스 내의 중요한 정보를 오래 유지하고, 불필요한 정보는 잊어버려 장기 의존성 문제를 효과적으로 해결함

Input, Output, Forget Gate

Input, Output, Forget Gate를 사용하여 시계열 데이터의 중요한 패턴과 추세를 학습하고 보존함

»

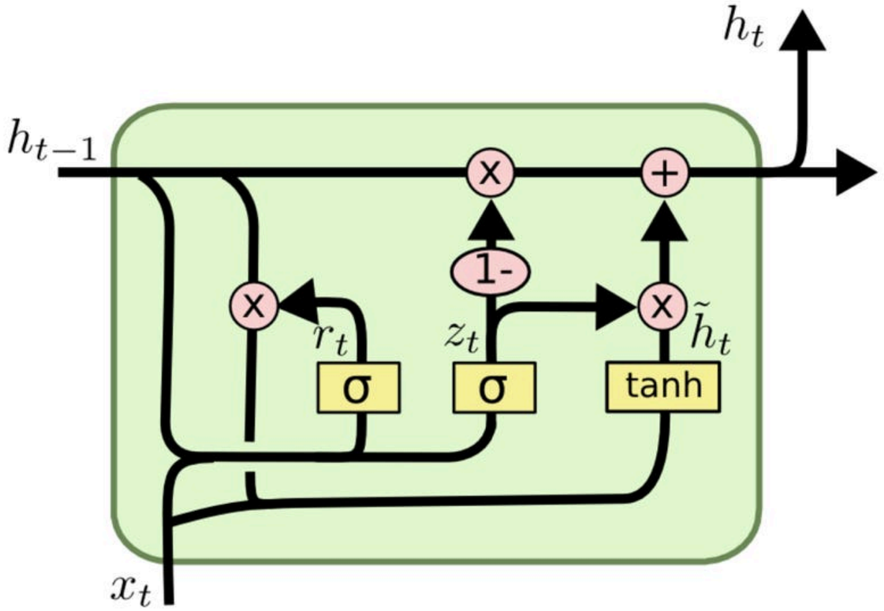
시계열 데이터에 효과적인
딥러닝 프레임워크

LSTM 구축

```
model = Sequential()
model.add(LSTM(units=128, return_sequences=True, input_shape=(X.shape[1], X.shape[2])))
model.add(Dropout(0.1))
model.add(LSTM(units=64))
model.add(Dropout(0.1))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
earlystop = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
model.fit(X_train, y_train, epochs=100, batch_size=32, validation_split=0.2, callbacks=[earlystop])
```

- 두 개의 LSTM 레이어를 사용
- 각 레이어 후에 드롭아웃을 추가하여 과적합을 방지
- EarlyStopping 콜백을 사용하여, 가장 좋은 가중치를 복원

GRU



긴 시간동안 RNN의 메모리를 유지

LSTM과 유사하게 게이트 메커니즘을 통해 중요한 정보를 유지하고 불필요한 정보를 제거함. GRU는 셀 상태 없이 두 개의 Gate만 사용하여 시퀀스 데이터의 장기 의존성 문제를 효과적으로 해결

Reset Gate와 Update Gate

Reset Gate와 Update Gate를 사용하여 시계열 데이터의 중요한 패턴과 추세를 학습하고 보존

시계열 데이터에 효과적인
딥러닝 프레임워크

GRU 구축

```
# 모델 구성
model = Sequential([
    GRU(128, return_sequences=True, input_shape=(X_train.shape[1], X_train.shape[2])),
    Dropout(0.2),
    GRU(64, return_sequences=False),
    Dropout(0.2),
    Dense(1, activation='sigmoid')
])
```

```
# 모델 컴파일
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# 조기 종료 설정
early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)

# 모델 학습
model.fit(X_train, y_train, epochs=100, batch_size=32, validation_split=0.2, callbacks=[early_stopping])
```

- 두 개의 GRU 레이어를 사용
- 각 레이어 후에 드롭아웃을 추가하여 과적합을 방지
- EarlyStopping 콜백을 사용하여, 가장 좋은 가중치를 복원

BITAmin 시계열 1조 최종 발표

피쳐 중요도 분석을 통한 KOSPI 지수 예측

피쳐중요도분석
: Attention, SHAP

Attention

중요한 부분에 더 ‘집중(Attention)’ 하자!

보통 자연어 처리와 컴퓨터 비전 분야에서 사용되는 기술

시계열 데이터에 적용하면 특정 시간 단계에서 중요한 정보에 집중

여러 개의 입력 벡터를 받고, 그 벡터를 세 종류 벡터로 변환

- 쿼리(Query): 주어진 문장에서 **현재 주목하는 단어**를 나타내는 벡터
- 키(Key): 각 **단어의 특성**을 나타내는 벡터
- 밸류(Value): 각 단어의 **실제 정보**를 나타내는 벡터

각 단어의 가중치를 계산하고 반영해 어떤 것에 집중하자

Attention 변수 중요도 가중치

가중치는 **각 단어의 중요도(기여도)**를 의미한다.

각 쿼리 벡터와 모든 키 벡터 사이의 유사도를 계산 (내적 등)

Attention 가중치 - 계산된 유사도 점수를 Softmax 함수에 통과시켜 가중치 생성

가중합(Weighted Sum) - 각 밸류 벡터에 해당 가중치를 곱한 후,
이를 모두 더해 최종 출력 벡터를 만듦

더 높은 가중치를 부여하는 것은 그만큼 더 주의(Attention)를 기울인다는 것

Attention

```
class TransformerBlock(tf.keras.layers.Layer):
    def __init__(self, embed_dim, num_heads, ff_dim, rate=0.1):
        super(TransformerBlock, self).__init__()
        self.att = MultiHeadAttention(num_heads=num_heads, key_dim=embed_dim)
        self.ffn = tf.keras.Sequential([
            Dense(ff_dim, activation='relu'),  # First dense layer
            Dense(embed_dim)  # Output layer
        ])
        self.layernorm1 = LayerNormalization(epsilon=1e-6)
        self.layernorm2 = LayerNormalization(epsilon=1e-6)
        self.dropout1 = Dropout(rate)
        self.dropout2 = Dropout(rate)

    def call(self, inputs, training=False):
        attn_output = self.att(inputs, inputs)
        attn_output = self.dropout1(attn_output, training=training)
        out1 = self.layernorm1(inputs + attn_output)
        ffn_output = self.ffn(out1)
        ffn_output = self.dropout2(ffn_output, training=training)
        return self.layernorm2(out1 + ffn_output)
```

TransfomerBlock 생성

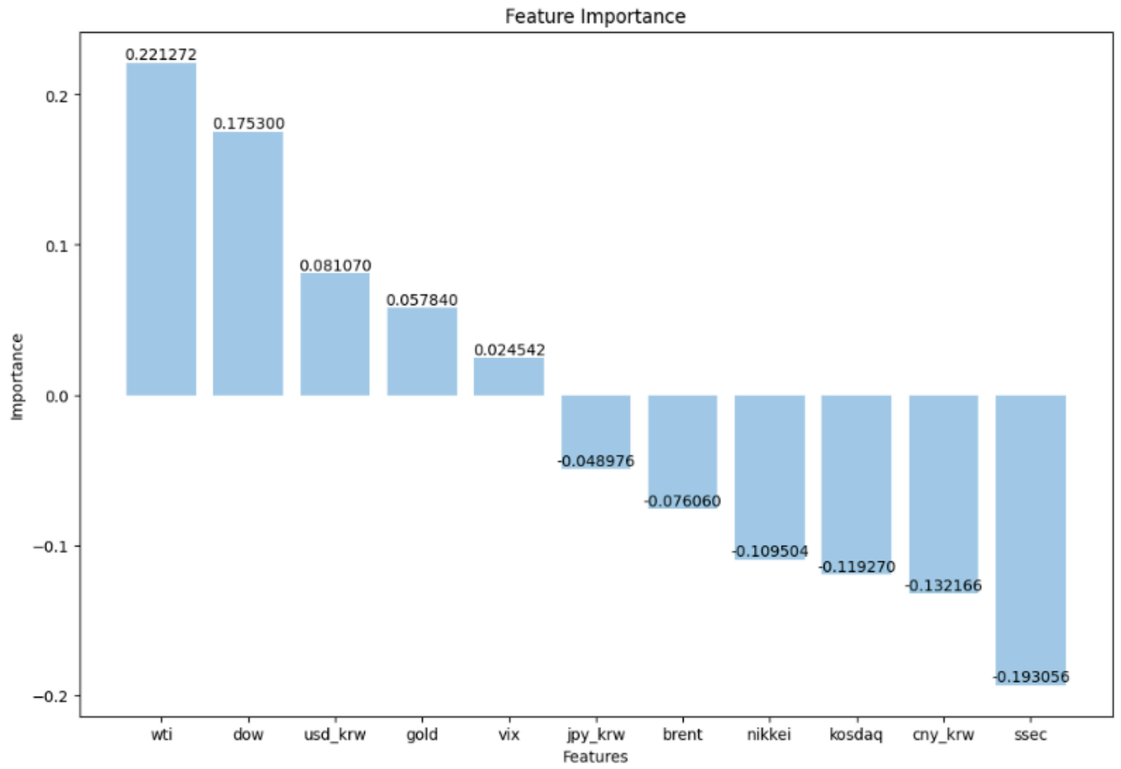
```
def build_model(input_shape, embed_dim, num_heads, ff_dim, rate=
    inputs = Input(shape=(input_shape,))
    x = Dense(input_shape, activation='relu')(inputs)  # 입력 차
    x = tf.expand_dims(x, axis=1)  # Add sequence dimension
    transformer_block = TransformerBlock(embed_dim, num_heads, f
    x = transformer_block(x)
    x = tf.squeeze(x, axis=1)  # Remove sequence dimension
    outputs = Dense(1)(x)  # Regression output
    model = Model(inputs=inputs, outputs=outputs)
    return model
```

Transformer 블록을 포함한 모델을 정의

```
def get_attention_weights(model):
    for layer in model.layers:
        if isinstance(layer, TransformerBlock):
            return layer.att.get_weights()[0]

attention_weights = get_attention_weights(model)
```

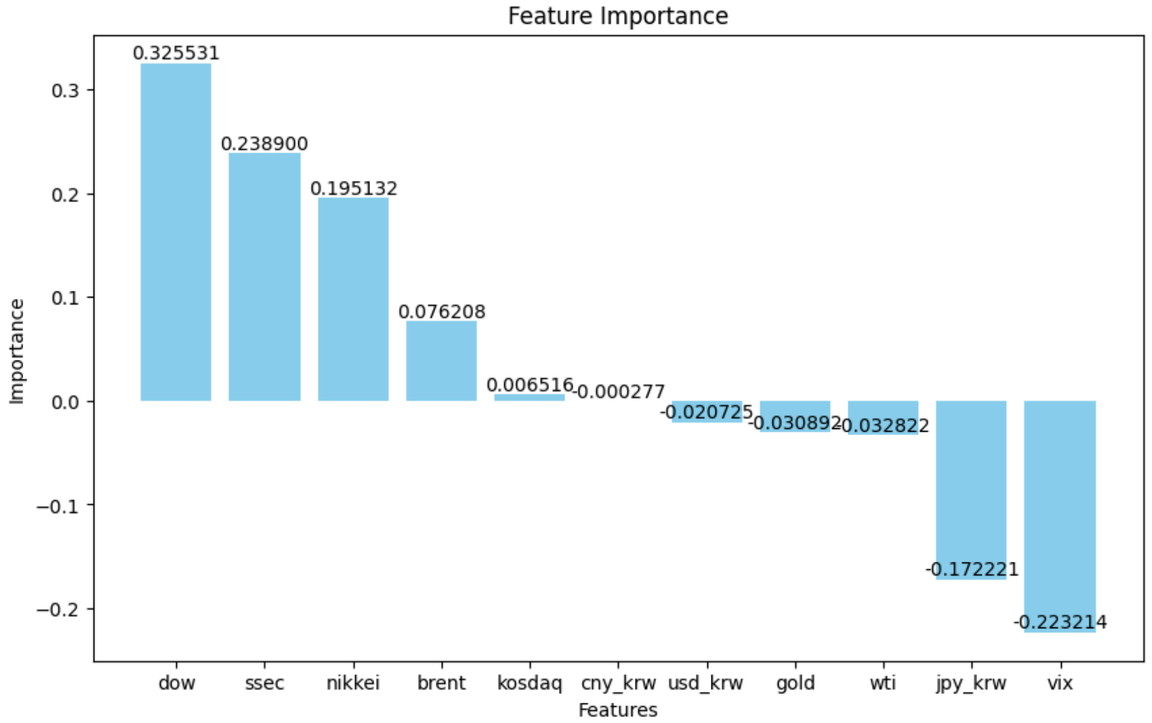
Attention 가중치 추출

종가



1. WTI - WTI 원유 가격
2. DOW - 다우존스 산업평균지수
3. USD_KRW - 달러 환율

변화율



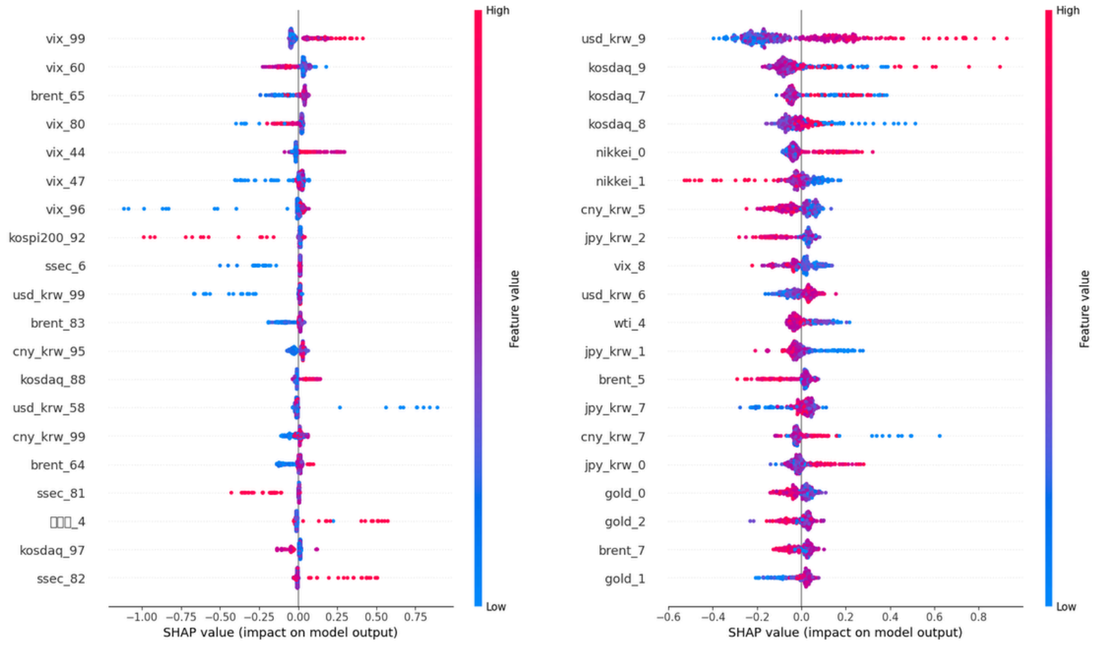
1. DOW - 다우존스 산업평균지수
2. SSEC - 상해종합지수
3. NIKKEI - 니케이 225 평균주가

SHAP

게임이론에 기반한, 머신러닝 모델의 출력을 설명하기 위한 접근법

```
import shap
# SHAP 값 계산
explainer = shap.TreeExplainer(model)
shap_values = explainer.shap_values(X_test.reshape(X_test.shape[0], -1))
```

```
# 시각화
# SHAP 요약 플롯 생성
shap.summary_plot(shap_values, X_test.reshape(X_test.shape[0], -1), feature_names=feature_names)
```



피쳐 중요도 시각화 해석

상위 20개의 피쳐가 표시되었다. (왼 : 종가, 오 : 변화율)

피처의 중요도: Y축의 순서대로 피처가 모델 예측에 미치는 중요도가 높다.
SHAP 값의 크기와 방향: X축에서 SHAP 값의 절대값이 클수록 해당 피처가 모델 예측에 더 큰 영향을 미친다. SHAP 값이 양수이면 해당 피처 값이 클수록 예측 확률이 증가함을 의미하고, 음수이면 예측 확률이 감소함을 의미.
피쳐 값의 효과: 빨간 색은 피쳐값이 높음, 파란색은 피쳐값이 낮음을 의미. 빨간 점이 오른쪽에 많이 분포한다면 해당 피쳐 값이 클수록 예측 확률이 증가함을 의미한다.

-> SHAP를 통해서 중요도 상위 피쳐를 찾아낼 수 있었다!

SHAP_ XGB

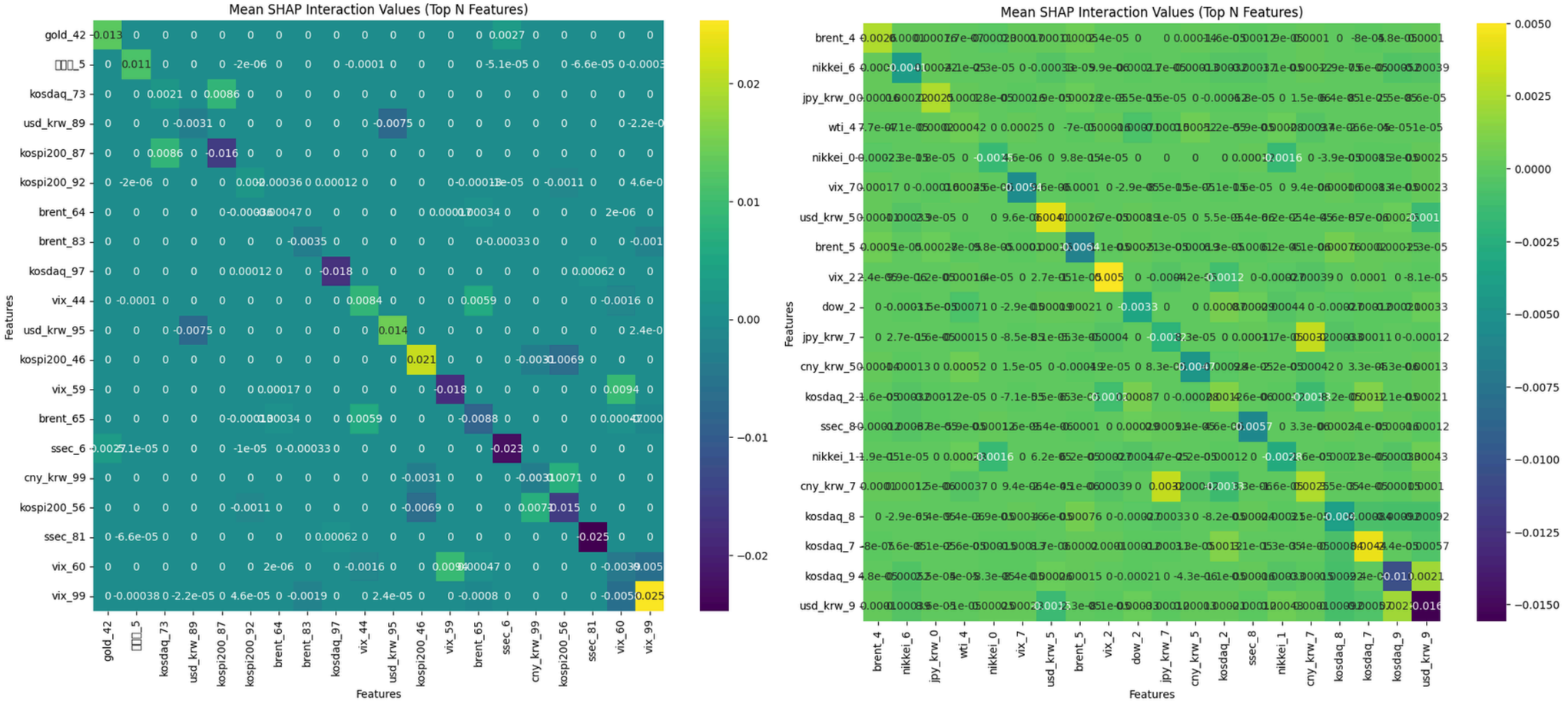
피쳐 간 상호작용 확인

```
# SHAP 상호작용 값 계산
shap_interaction_values = explainer.shap_interaction_values(X_test.reshape(X_test.shape[0], -1))
```

```
# 상위 N개의 피처만 선택 (여기서는 상위 20개로 가정)
N = 20
important_features = np.argsort(np.abs(mean_shap_interaction_values).sum(axis=0))[-N:]
important_features_names = [feature_names[i] for i in important_features]

# 상위 N개의 피처에 대한 상호작용 값 선택
mean_shap_interaction_values_topN = mean_shap_interaction_values[important_features][:, important_features]

# 히트맵을 사용한 시각화
plt.figure(figsize=(12, 10))
sns.heatmap(mean_shap_interaction_values_topN, xticklabels=important_features_names, yticklabels=important_features_names, cmap='viridis', annot=True)
plt.title('Mean SHAP Interaction Values (Top N Features)')
plt.xlabel('Features')
plt.ylabel('Features')
plt.show()
```

SHAP_ XGB

피쳐 간 상호작용 확인



-> 피쳐 간 상호작용의 정도는 미미하였다! (왼: 종가, 오: 변화율)

SHAP_ XGB

SHAP 결과 상위 피쳐만 사용하여 모델링_ 종가

```
# 상위 500개 피처 선택
shap_sum = np.abs(shap_values).mean(axis=0)
top_indices = np.argsort(shap_sum)[-500:]
top_features = [feature_names[i] for i in top_indices]

# 선택된 피처 인덱스
selected_feature_indices = [feature_names.index(feature) for feature in top_features]

def select_top_features(X, selected_indices):
    # 3차원 배열을 2차원 배열로 재구성한 후, 필요한 피처 선택
    X_reshaped = X.reshape(X.shape[0], -1)
    X_selected = X_reshaped[:, selected_indices]
    return X_selected

# 상위 20개 피처 선택 적용
X_train_selected = select_top_features(X_train, selected_feature_indices)
X_test_selected = select_top_features(X_test, selected_feature_indices)

# 모델 생성 및 훈련
model = xgb.XGBClassifier(
    learning_rate=0.05,
    max_depth=5,
    n_estimators=200,
    objective='binary:logistic',
    eval_metric='logloss'
)
model.fit(X_train_selected.reshape(X_train_selected.shape[0], -1), y_train)

# 예측
y_pred = model.predict(X_test_selected.reshape(X_test_selected.shape[0], -1))
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy:.4f}')
```

```
Accuracy: 0.5396
```

SHAP 변수 중요도 분석 결과 상위 피쳐만 사용하여 모델링 결과

1300개의 피쳐 (window_size=100) 중 상위 몇 백개만 사용하여 모델링.

1) 원래 모델 성능 : 0.5518
2) 300개의 피쳐만 사용한 결과 성능 : 0.52
3) 500개의 피쳐만 사용한 결과 성능 : 0.5396

-> SHAP 변수 중요도 분석을 통해 중요 변수를 뽑아낸 결과,
훨씬 적은 피쳐만으로도 성능을 비슷하게 유지할 수 있었다!

*상위 20개의 피쳐에 가중치를 부여하여 모델링도 해보았으나,
모델의 성능에 차이가 없음도 발견할 수 있었다.

SHAP_ XGB

SHAP 결과 상위 피쳐만 사용하여 모델링_ 변화율

```
# 상위 10개 피처 선택
shap_sum = np.abs(shap_values).mean(axis=0)
top_indices = np.argsort(shap_sum)[-10:]
top_features = [feature_names[i] for i in top_indices]

# 선택된 피처 인덱스
selected_feature_indices = [feature_names.index(feature) for feature in top_features]

def select_top_features(X, selected_indices):
    # 3차원 배열을 2차원 배열로 재구성한 후, 필요한 피처 선택
    X_reshaped = X.reshape(X.shape[0], -1)
    X_selected = X_reshaped[:, selected_indices]
    return X_selected

# 상위 20개 피처 선택 적용
X_train_selected = select_top_features(X_train, selected_feature_indices)
X_test_selected = select_top_features(X_test, selected_feature_indices)

# 모델 생성 및 훈련
model = xgb.XGBClassifier(
    learning_rate=0.05, max_depth=5, n_estimators=100,objective='binary:logistic', eval_metric='logloss')
model.fit(X_train_selected.reshape(X_train_selected.shape[0], -1), y_train)

# 예측
y_pred = model.predict(X_test_selected.reshape(X_test_selected.shape[0], -1))
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy:.4f}')
```

```
Accuracy: 0.5954
```

SHAP 변수 중요도 분석 결과 상위 피쳐만 사용하여 모델링 결과

130개의 피쳐 (window_size=10) 중 상위 몇 개만 사용하여 모델링.

1) 원래 모델 성능 : 0.6098
2) 5개의 피쳐만 사용한 결과 성능 : 0.5925
3) 10개의 피쳐만 사용한 결과 성능 : 0.5954

-> SHAP 변수 중요도 분석을 통해 중요 변수를 뽑아낸 결과,
훨씬 적은 피쳐만으로도 성능을 비슷하게 유지할 수 있었다!

피쳐 중요도 분석을 통한 KOSPI 지수 예측

결과 비교 및 결론

Conclusion

피쳐 중요도 분석을 통한
KOSPI200 지수 예측 모델 성능 비교

종가 vs
변화율

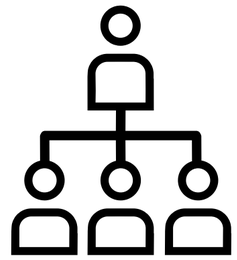
Attention을
이용한
Feature
Importance

XGBoost,
GRU, LSTM 간의
성능 차이

결과 비교

종가와 변화율 사용에 따른 모델 성능 비교

	Xgboost	LSTM	GRU
종가 데이터 사용	0.5518	0.5374	0.49
변화율 데이터 사용	0.6098	0.5432	0.51



XGBoost가 가장 우수한 성능 보여줌

LSTM이나 GRU에 비해 XGBoost가 더 높은 성능을 보였다.

변화율 데이터에서 성능 향상을 보여줌

변화율 데이터를 사용하여 성능이 향상된 것은,
변화율 데이터가 종가 데이터보다 더 유의미한 패턴과 트렌드를
제공함으로써 모델의 예측 정확도를 높이고, 노이즈를 줄이며,
일반화 능력을 향상시키는 데 효과적임을 확인함

결과 비교

변수중요도 적용유무에 따른 모델 성능 비교

	Xgboost	LSTM	GRU
변수 중요도 적용 x	0.5518	0.5067	0.49
변수 중요도 적용	0.5427	0.5374	0.51

XGBoost가 가장 우수한 성능 보여줌

LSTM이나 GRU에 비해 XGBoost가 더 높은 성능을 보였다.

attention을 이용한 Feature Importance 적용한 결과: LSTM, GRU 성능 증가 / XGB 성능 약간 감소

LSTM, GRU 의 경우 변수중요도를 적용하지 않은 데이터세트보다
변수 중요도를 적용한 데이터 세트에서 약간의 성능 향상을 보임.
XGBoost의 경우 어텐션 결과를 적용한 결과 약간의 성능 하강을 보임.

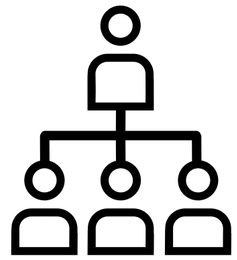
이는 LSTM과 GRU가 RNN계열의 모델로, 어텐션 매커니즘을 적용함으로써
시퀀스 데이터 내의 중요 정보를 잘 파악할 수 있게 하는 반면
XGB는 트리기반의 모델로, 피쳐 간 순차적 관계를 고려하는 어텐션 매커니즘
이 오히려 노이즈를 늘릴 수 있기 때문이라고 추정할 수 있다.

*트리기반 모델은 순차적 피쳐 관계 고려 X

결과 비교

변수 중요도 적용 하는 방법에 따른 모델 성능 비교

	Xgboost	LSTM	GRU
변수 중요도 계수 적용	0.5427	0.5374	0.51
변수 중요도 양수값만 적용	0.5183	0.5297	0.51



Xgboost와 LSTM이 각각 우수한 성능 보여줌

변수 중요도 계수를 적용한 데이터세트에서는 Xgboost가,
변수 중요도가 양수인 계수만 적용한 데이터세트에서는 LSTM이
우수한 성능을 보여줌

변수 중요도 계수 적용한 경우 성능이 가장 우수

각 변수의 중요도를 곱한 데이터에서 모델의 성능이 가장 우수함
변수 중요도 양수값만 적용한 경우 약 0.01의 성능이 감소함
음수가 나온 칼럼들이 원래 데이터의 절반정도임을 고려하였을 때,
의미있는 결과로 볼 수 있음

결론

종가 VS 변화율 데이터

변화율 데이터가 데이터의 잡음을 줄이고
추세성을 반영함으로써 종가 데이터보다
정확한 성능을 보인다.
-> 주가 예측에서는 변화율 데이터를 사용하는 것이 좋다.

Attention 매커니즘 활용 변수 중요도 분석

- 매커니즘을 활용해 변수 중요도 가중치를 얻을 수 있다.
- LSTM, GRU의 예측 성능이 증가했다.

-> Attention 변수 중요도 정보로 시계열 분석에서의
RNN 계열 모델 정확도를 올릴 수 있다.

변수 중요도 분석 의의

- 변수 중요도 분석을 통해 모델의 성능의 개선을 꾀할 수 있다.
- 변수 중요도 분석을 통해 변수 선택 (Feature Selection)을 적용하여,
  적은 피쳐만으로도 모델의 성능을 유지할 수 있다.

➡ 주가 예측에서 데이터의 피쳐가 많을 경우의 처리가 용이하다.

감사합니다

BITAmin 시계열 1조