Notes on ML for Algorithmic Trading

2021 – 04 – 22

Junfan Zhu

(junfanz@gatech.edu; junfanzhu@uchicago.edu)

https://www.linkedin.com/in/junfan-zhu/

This is a notes composed by Junfan Zhu on the book.

Notes published on https://github.com/junfanz1/Quant-Books-Notes.

Book Link

Machine Learning for Algorithmic Trading: Predictive models to extract signals from market and alternative data for systematic trading strategies with Python, 2nd Edition Paperback – July 31, 2020 by Stefan Jansen

https://www.amazon.com/Machine-Learning-Algorithmic-Trading-alternative/dp/1839217715

Table of Contents

- 10.2. Probabilistic Programming with PyMC3 (2017)
  - 10.2.1. Data and indicators
  - 10.2.2. Generalized Linear Models (GLM)
  - 10.2.3. Exact MAP inference
  - 10.2.4. Approximate Inferece - MCMC
  - 10.2.5. Approximate inference - variational Bayes
- 10.3. Bayesian Model Diagnostics
- 11. Random Forests - Long-Short trading strategy
- 11.1. Decision Trees
- 11.2. Random Forests - making trees more reliable
  - 11.2.1. Bootstrap aggregation = Bagging
  - 11.2.2. Random Forest Pros and Cons
- 12. Boosting Trading Strategy
- 12.1. Adaptive Boosting (AdaBoost)
- 12.2. Gradient Booosting - Ensembles for most tasks
- 12.3. Train and Tune GBM models
  - 12.3.1. Subsampling and Stochastic gradient boostinng
- 12.4. XGBoost, LightGBM and CatBoost
  - 12.4.1. Simpliefied split-finding algorithms
  - 12.4.2. Depth-wise vs. Leaf-wise growth
  - 12.4.3. DART (dropout for additive regression trees)
  - 12.4.4. Categorical Features
  - 12.4.5. Additional features and optimizations
- 12.5. Long Short Trading Strat with boosting
- 12.6. Evaluation
  - 12.6.1. Feature Importance
  - 12.6.2. SHapley Additive exPlanatins (SHAP)
- 12.7. Boosting for Intraday Strat
- 12.8. Summary
- 13. Data-Driven Risk Factors & Asset Allocation w/ Unsupervised Learning
- 13.1. PCA vs. ICA
- 13.2. Manifold learning - nonlinear dim reduction
  - 13.2.1. t-distributed Stochastic Neighbor Embedding
  - 13.2.2. Uniform Manifold Approximation and Projection (UMAP)
- 13.3. Clustering
  - 13.3.1. Density-based spatial clustering of applications with noise (DBSCAN, 1996)
  - 13.3.2. Gaussian Mixture Models (GMM)

1. ML4T

1997 order-handling rules by SEC introduced competition to exchanges through **electronic communication networks (ECNs)**, automated *alternatve trading systems (ATS)* that match buy-and-sell orders at specified prices, registered as broker-dealers, allowing significant brokerages and individual traders in different geographic locations to trade without intermediaries, both on exchanges and after hours.

Dark Pool: private ATS that allow institutional investors to trade large orders without publicly revealing info, contrary to how exchanges managed order books prior to competition from ECNs. Dark pools don't publish pretrade bids and offers, trade prices only become public some time after execution.

Direct Market Access (DMA) gives trader control over execution by allowing them to send orders directly to exchange, using infrastructure and market participant identification of a broker who is a member of an exchange. Sponsored access removes pre-trade risk controls by brokers and is the basis of HFT (automated trades in financial instruments executed with very low latency in microsecond range, goal is to detect inefficiencies in market microstructure).

HFT earns small profits per trade using passive or aggressive strategies.

- Passive strategies: arbitrage trading, small price differentials
- Aggressive strategies include order anticipation (liquidity detection: algo submits small exploratory orders to detect hidden liquidity from large institutional investors and trade ahead of large order to benefit from subsequent price movements), momentum ignition (algorithm executing and canceling a series of orders to spoof other HFT algorithms into buying/selling more aggresively and benefit from price changes).

1.1. Factor Investing and Smart Beta

Fixed income value strategy: **riding the yield curve** is a form of duration premium. In commodities, it's called **roll return**, with a return > 0 for upward-sloping futures curve. In FX, value strategy is **carry**.

Conventional data includes economic stats, trading data and corporate reports. **Alternative data** is broader and includes sources like satellite images, credit card sales, sentiment analysis, mobile geolocation ddata, and website scrapping, which includes any data source containing potential trading signals.

https://alternativedata.org/

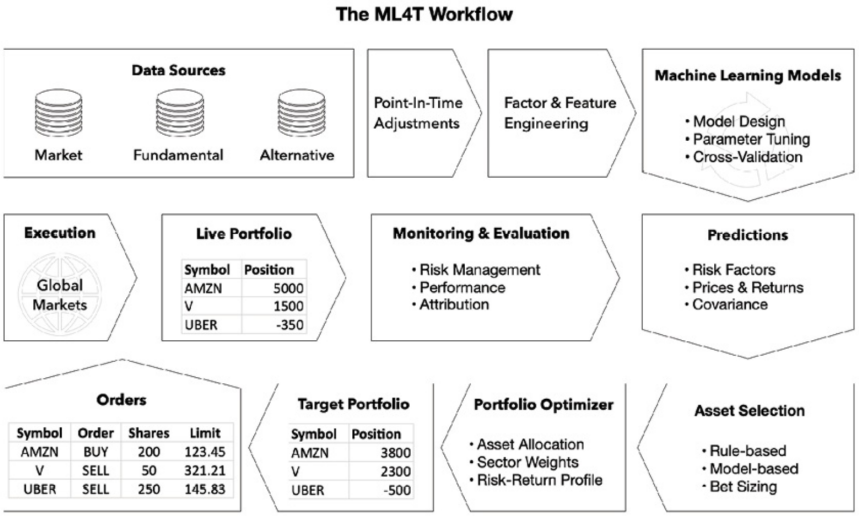

Figure 1: ML4T workflow

1.2. Designing and Executing ML-driven strategy

To obtain unbiased performance estimates for strategy, we need **backtest engine** that simulates execution in a realistic manner.

2. Market and Fundamental Data - Sources and Techniques

Market microstructure studies how institutional environment affect trading process and shapes outcomes (price discovery, bid-ask spreads, intraday trading, transaction costs).

Alternative Trading Systems (ATS) such as electronic communication networks (ECNs) include dark pools. Regulation National Market System (Reg NMS) established National Best Bid and Offer (NBBO) mandate for brokers to route orders to venues that offer the best price.

US stock markets provide quotes in 3 tiers:

- Level 1 (L1): rel-time bid/ask price info
- L2: Adds info about bid and ask by specific market makers and size and time of recent transactions.
- L3: Adds ability to enter or change quotes, execute orders, and confirm traddes and is available only to market makers and exchange member firms.

Access to L3 quotes permirs registered brokers to meet best execution requirements.

2.1. Nasdaq TotalView-ITCH data feed

Nasdaq offers TotalView-ITCH **direct data-feed protocal**, allowing subscribers to track individual orders for equity instruments from placement to execution or cancellation.

In addition to matching market and limit orders, Nasdaq operates **auctions or crosses** that execute a large number of trades at market opening and closing. Crosses are important since passive investing is growing. It also provide Net Order Imbalance Indicator (NOII).

2.1.1. Parse Binary order messages

Parser translates message specs into format strings and named tuples

```
# Get ITCH specs and create formatting (type, length) tuples
specs = pd.read_csv('...')
specs['formats'] = specs[['value','length']].apply(typle, axis=1).map(formats)
# Formatting for alpha fields
alpha_fieldsd = specs[specs.value == 'alpha'].set_index('name')
alpha_msgs = alpha_fields.groupby('message_type')
alpha_formats = {k: v.to_dict() for k,v in alpha_msgs.formats}
alpha_length = {k: v.add(5).to_dict() for k,v in alpha_msgs.length}
# generate message classes as named tuples and format strings
message_fields, fstring = {}, {}
for t, message in specs.groupby('message_type'):
    message_fields[t] = namedtuple(typename = t, field_names = message.name.tolist())
    fstring[t] = '>' + ''.join(message.formats.tolist())

def format_alpha(mtype, data):
    for col in alpha_formats.get(mtype).keys():
        if mtype != 'R' and col == 'stock':
            data = data.drop(col, axis=1)
            continue
        data.loc[:,col] = (data.loc[:,col]
                           .str.decode('utf-8')
                           .str.strip())
        if encoding.get(col):
            data.loc[:,col] = data.loc[:,col].map(encoding.get(col))
    return data
```

Binary file for a single day has 300,000,000 messages with over 9GB. Scripts appends the parsed result iteratively to HDF5 format file to avoid memory contraints.

```
with (data_path / file_name).open('rb') as data:
    while True:
        message_size = int.from_bytes(data.read(2), byteorder = 'big', signed=False)
        message_type = data.read(1).decode('ascii')
        message_type_counter.update([message_type])
        record = data.read(message_size - 1)
        message = message_fields[message_type]._make(unpack(fstring[message_type], record))
        message[message_type].append(message)

        # deal with system events like market open/close
        if message_type == 'S':
            timestamp = int.from_bytes(message.timestamp, byteorder='big')
            if message.event_code.decode('ascii') == 'C': # close
                store_messages(messages)
                break
```

2.1.2. Summarizing trading activity for r8500 stocks

```
with pd.HDFStore(itch_store) as store:
    stocks = store['R'].loc[:, ['stock_locate','stock']]
    trades = (store['P'].append(
        store['Q'].rename(columns={'cross_price':'price'}), sort=False
    ).merge(stocks))
trades['value'] = trades.shares.mul(trades.price)
trades['value_share'] = trades.value.div(trades.value.sum())
trade_summary = (trades.groupby('stock').value_share.sum().sort_values(ascending=False))
trade_summary.iloc[:50].plot.bar(figsize=(14,6),
                    color = 'darkblue',
                    title ='Share of Traded Value')
f = lambda y, _ : '{:.0%}'.format(y)
plt.gca().yaxis.set_major_formatter(FuncFormatter(f))

# collect orders for single stock that affects trading
def get_messages(date, stock=stock):
    # collect trading messages for given stock
    with pd.HFStore(itch_store) as store:
        stock_locate = store.select('R', where = 'stock = stock').stock_locate.iloc[0]
        target = 'stock_locate = stock_locate'
        data= {}
        messages = ['A','F','E','C','X','D','U', 'P','Q']
        for m in messages:
```

```
            data[m] = store.select(m,
              where=target).drop('stock_locate', axis=1).assign(type=m)
    order_cols = ['order_reference_number','buy_sell_indicator','shares','price']
    orders = pd.concat([data['A'], data['F']], sort = False, ignore_index = True).loc[:,orde
    for m in messages[2:-3]:
        data[m] = data[m].mergee(orders, how='left')
    data['U'] = data['U'].merge(orders, hoow='left', right_on = 'order_reference_number', le
        suffixes=['','_replaced'])
    data['Q'].rename(columns={'cross_price':'price'}, inplace=True)
    data['X']['shares'] = data['X']['cancelled_shares']
    data['X'] = data['X'].dropna(subset=['price'])
    data = pd.concat([data[m] for m in messages], ignore_index=True, sort=True)

def get_trades(m):
    trade_dict = {'executed_shares':'shares','execution_price':'price'}
    cols = ['timestamp','executed_shares']
    trades = pd.concat([m.loc[m.type == 'E',
                cols+['price']].rename(columns=trade_dict),
                m.loc[m.type == 'C',
                cols+['execution_price']]
                .rename(columns=trade_dict),
                m.loc[m.type == 'P', ['timestamp','price','shares']],
                m.loc[m.type == 'Q', ['timestamp', 'price', 'shares']]
                .assign(cross=1),], sort = False).dropna(subset=['price']).fillna(0)
    return trades.set_index('timestamp').sort_index().astype(int)

def add_orders(orders, buysell, nlevels):
    new_order = []
    items = sorted(orders.copy().items())
    if buysell == 1:
        items = reversed(items)
    for i, (p,s) in enumerate(items,1):
        new_order.append((p,s))
        if i == nlevels:
            break
    return orders, new_order

# iterate over ITCH messages
for message in messages.itertuples():
    i = message[0]
    if np.isnan(message.buy_sell_indicator):
        continue
    message_counter.update(message.type)
    buysell = message.buy_sell_indicator
    price, shares = None, None
    if message.type in ['A','F','U']:
```

```
        price, shares = int(message.price), int(message.shares)
        current_orders[buysell].update({price: shares})
        current_orders[buysell], new_order = add_orders(current_orders[buysell], buysell, nl
        order_book[buysell][message.timestamp] = new_order
    if message.type in ['E','C','X','D','U']:
        if message.type == 'U':
            if not np.isnan(message.shares_replaced):
                price = int(message.price_replaced)
                shares = -int(message.shares_replace)
        else:
            if not np.isnan(message.price):
                price = int(message.price)
                shares = -int(message.shares)
        if price is not None:
            current_orders[buysell].update({price: shares})
            if current_orders[buysell][price] <= 0:
                current_orders[buysell].pop(price)
            current_orders[buysell], new_order = add_orders(current_orders[buysell], buysell
            order_book[buysell][message.timestamp] = new_order
```

2.2. From ticks to bars: regularize market data

Bid-ask bounce causes price to oscillate between bid ask prices when trade initiation alternates betwween buy/sell orders. To improve noise-signal ratio, we resample and regularize tick data by aggregating the trading.

2.2.1. Raw material: tick bars

```
stock, date = 'AAPL', '20210103'
title = '{} | {}'.format(stock, pd.to_datetime(date).date())
with pd.HDFStore(itch_store) as store:
    sys_events = store['S'].set_index('event_code') # system events
    sys_events.timestamp = sys_events.timestamp.add(pd.to_datetime(date)).dt.time
    market_open = sys_events.loc['Q', 'timestamp']
    market_close = sys_events.loc['M','timestamp']
with pd.HDFStore(stock_store) as store:
    trades = store['{}/trades'.format(stock)].reset_index()
trades = trades[trades.cross == 0] # excluding data from open/close crossings
trades.price = trades.price.mul(1e-4) # format price
trades = trades[trades.cross == 0] # exclude crossing trades
trades = trades.between_time(market_open,market_close) # market hours only
tick_bars = trades.set_index('timestamp')
tick_bars.index = tick_bars.index.time
tick_bars.price.plot(figsize=(10,5), title=title), lw=1)
```

2.2.2. Plain-vanilla denoising - time bars

```
def get_bar_stats(agg_trades):
    vwap = agg_trades.apply(lambda x:np.average(x.price, weights = x.shares)).to_frame('vwap
    ohlc = agg_trades.price.ohlc()
    vol = agg_trades.shares.sum().to_frame('vol')
    txn = agg_trades.shares.size().to_frame('txn')
    return pd.concat([ohlc, vwap, vol, txn], axis=1)
resampled = trades.groupby(pd.Grouper(freq='1Min'))
time_bars = get_bar_stats(resampled)

def price_volume(df, price='vwap', vol='vol', suptitle=title, fname=None):
    fig, axes = plt.subplots(nrows=2, sharex=True, figsize=(15,8))
    axes[0].plot(df.index, df[price])
    axes[1].bar(df.indedx, df[vol], width=1/(len(df.index)), color='r')
    xfmt = mpl.dates.DateFormatter('%H:%M')
    axes[1].xaxis.set_major_locator(mpl.dates.HourLocator(interval=3))
    axes[1].xaxis.set_major_formatter(xfmt)
    axes[1].get_xaxis().set_tick_params(which='major', pad=25)
    axes[0].set_title('Price', fontsize=14)
    axes[1].set_title('Volume', fontsize=14)
    fig.autofmt_xdate()
    fig.suptitle(suptitle)
    fig.tight_layout()
    plt.subplots_adjusts(top=0.9)
price_volume(time_bars)
```

2.2.3. Volume bars: Aggregate trade data by volume

Figure 2.6

```
min_per_trading_day = 60*7.5
trades_per_min = trades.shares.sum()/min_per_trading_day
trades['cumul_vol'] = trades.shares.cumsum()
df = trades.reset_index()
by_vol = (df.groupby(df.cumul_vol.
                     div(trades_per_min)
                     .round().astype(int)))
vol_bars = pd.concat([by_vol.timestamp.last().to_frame('timestamp'),
                 get_bar_stats(by_vol)], axis=1)
price_volume(vol_bars.set_index('timestamp'))
```

2.2.4. Dollar bars: account for price changes

```
value_per_min = trades.shares.mul(trades.price).sum()/(60*7.5) # min per trading day
trades['cumul_val'] = trades.shares.mul(trades.price).cumsum()
df = trades.reset_index()
by_value = df.groupby(df.cumul_val.div(value_per_min).round().astype(int))
dollar_bars = pd.concat([by_value.timestamp.last().to_frame('tiimestamp'), get_bar_stats(by_
price_volume(dollar_bars.set_index('timestamp'), suptitile=f'Dollar Bars | {stock} | {pd.to_
```

2.2.5. Process AlgoSeek intraday data

Minute bar data comes in 4 versions: with/without quote info, and with/without FINRA's reported volume. There's one zipped folder per day containing one csv file per ticker.

```
directories = [Path(d) for d in ['1min_trades']]
target = directory / 'parquet'
for zipped_file in directory.glob('*/**/*.zip'):
    fname = zipped_file.stem
    print('\t', fname)
    zf = ZipFile(zipped_file)
    files = zf.namelist()
    data = (pd.concat([pd.read_csv(zf.open(f),
    parse_dates = [['Date'],['TimeBarStart']]
      for f in file], ginore_index = True)
      .rename(columns = lambda x:x.lower())
      .rename(columns = {'date_timebarstart':'date_time'})
      .set_index(['ticker', 'date_time']))
    data.to_parquet(target/(fname+'.parquet'))
```

Combine `parquet` files into a single piece of HDF5 storage, yielding 53.8 million records that consume 3.2 GB memory covering 5 years of 100 stocks. Use `plotly` to quickly create interactive candlestick plot for 1 day of AAPL data.

```
path = Path('1min_trades/parquest')
df = pd.concat([pd.read_parquet(f) for f in path.glob('*.parquet')]).dropna(how='all', axis=
df.columns = ['open','high','low','close','trades','volume','vwap']
df.to_hdf('data.h5','1min_trades')
print(df.info(null_counts = True))

idx = pd.IndexSlice
with pd.HFStore('data.h5') as store:
    print(store.info())
    df = (store['1min_tradeds']
```

```
            .loc[idx['AAPL','2021-01-02'], :]
            .reset_indedx())
fig = go.Figure(data = go.Ohlc(x = df.date_time,
                               open = df.open,
                               high = df.high,
                               low = df.low,
                               close = df.close))
```

2.2.6. API access to market data: Zipline

```
%load_ext zipline
%%zipline -- start 2010-1-1 -- end 2021-1-1 --data-frequency daily
from zipline.api import order_target, record, symbol
def initialize(context):
    context.i = 0
    context.assets = [symbol('FB'), symbol('GOOG'), symbol('AMZN')]
def handle_data(context, data):
    df = data.history(context.assets, fields=['price','volume'], bar_count = 1, frequency='1
    df = df.to_frame().reset_index()
    if context.i == 0:
        df.columns = ['date','asset','price','volume']
        df.to_csv('stock_data.csv', index = False)
    else:
        df.to_csv('stock_data.csv', index=False, mode = 'a', header = None)
    context.i += 1
df = pd.read_csv('stock_data.csv')
df.date = pd.to_datetime(df.date)
df.set_index('date').groupby('asset').price.plot(lw=2, legend=True, figsize=(14,6))
```

Automated processing: XBRL by SEC, standard for electronic representation and exchange of business reports.

2.3. Efficient Data Storage with pandas

- csv
- HDF5: Hierarchical data format, fast and scalable storage format for numerical data.
- Parquet: Apache Hadoop ecosystem, binary columnar storage format providing efficient data compression. It's the best choice for read/write operations for a mix of numerical and text data.

3. Alternative Data for Finance - Categories and Use Cases

How to evaluate trading strategies driven by alternative data using historical data, backtests, to estimate amount of alpha in dataset.

3.1. Parsing Data from HTML with Requests and BeautifulSoup

```
from bs4 import BeautifulSoup
import requests
# parse new html -> soup object
soup = BeautifulSoup(html.text, 'html.parser')
# for each span tag, print out text
for entry in soup.find_all(name = 'span', attrs={'class': 'rest-row-name-text'}):
    print(entry.text)

def parse_html(html):
    data, item  = pd.DataFrame(), {}
    soup = BeautifulSoup(html, 'lxml')
    for i, resto in enumerate(soup.find_all('div', class_ = 'rest-row-info')):
        item['name'] = resto.find('span', class_ = 'rest-row-name-text').text
        booking = resto.find('div', class_ = 'booking')
        item['bookings'] = re.search('\d+', boooking.text).group()\
            if booking else 'NA'
        rating  = resto.fin('div', class_ = 'star-rating-score')
        item['rating'] = float(rating['aria-label'].split()[0])\
            if rating else 'NA'
        reviews = resto.fin('span',class_ = 'unerline-hover')
        item['reviews'] = int(re.search('\d+', reviews.text).group())\
            if reviews else 'NA'
        item['price'] = int(resto.find('div', class_ = 'rest-row-pricing')
            .find('i').text.count('$'))
        cuisine_class = 'rest-row-meta--cuisine rest-row-meta-text...'
        item['cuisine'] = resto.find('span', class_cuisine_class).text
        location_class = '...'
        item['location'] = resto.find('span', class_= location_class).text
        data[i] = pd.Series(item)
    return data.T
```

4. Financial Feature Engineering: Research Alpha Factors

The portion of an asset's return not explained by exposure to this benchmark is alpha, signals that aim to produce uncorrelated returns are alpha factors.

Cross-asset relative value strategies focus on mispricing across asset classes, like convertible bond arbitrage involves trades on relative value between bond that can be turned into equity and underlying stock of a single company. Likewise, trades between credit andequity vol, using credit signals to trade equities or trades between commodities and related equities.

Load Quandl Wiki stock price data on US equities, select time slice by `pd.IndexSlice` to `pd.MultiIndex`, and unpivot adjusted close price column with `.stack()` method to convert DataFrame into wide format, with tickers in columns and timestamps in rows.

To capture time-series dynamics like momentum patterns, we compute historical multi-period returns with `pct_change(n_periods)`, where `n_periods` is number of lags, then we convert wide result back into long format with `.stack()`, use `.pipe()` to apply `.clip()` method to resulting DataFrame, and winsorize returns at [1%, 99%] levels (cap outliers at these percentiles). Finally, normalize returns using geometric average, change orders of `MultiIndex` levels with `.swaplevel()`.

```
idx = pd.IndexSlice
with pd.HDFStore('...h5') as store:
    prices = (store['quandl/wiki/prices']
                .loc[idx['2000':'2018',:], 'adj_close']
                .unstack('ticker'))
prices.info()
# Resampling from daily to monthly frequency
monthly_prices = prices.resample('M').last()

outlier_cutoff = 0.01
data = pd.DataFrame()
lags = [1,2,3,6,9,12]
for lag in lags:
    data[f'return_{lag}m'] = (monthly_prices
                          .pct_change(lag)
                          .stack()
                          .pipe(lambda x:
                              x.clip(lower = x.quantile(outlier_cutoff),
                              upper = x.quantile(1-outlier_cutoff)))
                          .add(1)
                          .pow(1/lag)
                          .sub(1))
```

```
data = data.swaplevel().dropna()
data.info()

for lag in [2,3,6,9,12]:
    data[f'momentum_{lag}'] = data[f'return_{lag}m'].sub(data.return_1m)
data[f'momentum_3_12'] = data[f'return_12m'].sub(data.return_3m)

# use .shift() to move historical returns up to current period
for i in range(1,7):
    data[f'return_1m_t-{t}'] = data.groupby(level='ticker').return_1m.shift(t)
factors = ['Mkt-RF','SMB','HML','RMW','CMA']
factor_data = web.DataReader('F-F_Research_Data_5_Factors_2x3','famafrench',start = '2000')
factor_data.index = factor_data.index.to_timestamp()
factor_data = factor_data.resample('M').last().div(100)
factor_data.index.name = 'date'
factor_data = factor_data.join(data['return_1m']).sort_index()
T = 24
betas = (factor_data
        .groupby(level='ticker', group_keys = False)
        .apply(lambda x:PandasRollingOLS(window=min(T, x.shape[0]-1),
        y = x.return_1m, x=x.drop('return_1m',axis=1)).beta))
```

4.1. Denoising Alpha factors with Kalman Filter

Dynamic linear model of sequential data that adapts to new info arriving. Kalman Filter incorporates new data into estimates of current value of time series based on prob model.

- Prediction step: estimate current state of process
- Measurement step: use noisy observations to update its estimate by averaging info from both steps in a way that weighs more certain estimates higher.

It flexibly adapts to non-stationary data with changing distributional characteristics. Kalman filter has been extended to nonlinear dynamics in the form of unscented Kalman filters. Particle filter is an alternative approach that uses sampling-based Monte Carlo approaches to estimate nonnormal distributions.

Apply Kalman filter to smoothen S&P 500 stock price series. Compared to MA, Kalman Filter is more sensitive to changes in time series.

```
with pd.HDFStore(DATA_STORE) as store:
    sp500 = store['sp500/stooq'].loc['2008':'2009','close']
from pykalman import KalmanFilter
kf = KalmanFilter(transition_matrices = [1],
```

```
                  observation_matrices = [1],
                  initial_state_mean = 0,
                  initial_state_covariance = 1,
                  observation_covariance = 1,
                  transition_covariance = .01)
state_means, _ = kf.filter(sp500)
sp500_smoothed = sp500.to_frame('close')
sp500_smoothed['Kalman Filter'] = state_means
for months in [1,2,3]:
    sp500_smoothed[f'MA ({months}m)'] = (sp500.rolling(window=months*21).mean())
    ax = sp500_smoothed.plot(title='Kalman Filter vs Moving Average', figsize=(14,6), lw=1,
```

4.2. Preprocess noisy signals with Wavelets

Wavelets combines sine and cosine waves at different frequencies to approximate noisy signals, it can filter out specific patterns that may occur at different scales, which in turn may correspond to a frequency range.

To denoise a signal, we can use wavelet shrinkage and thresholding methods. We choose a specific wavelet pattern to decompose a dataset, the wavelet transform yields coefficients that correspond to details in dataset.

Thresholding is to omit all coefficients below a cutoff, assuming they represent minor details that aren't necessary to represent true signal. After that, we use inverse transform back.

```
import pywt
pywt.families(short=False)
# See all kinds of wavelet libraries

wavelet = 'db6'
for i, scale in enumerate([.1, .5]):
    coefficients = pywt.wavedec(signal, wavelet, mode='per')
    coefficients[1:] = [pywt.threshold(i, value=scale*signal.max(), mode='soft') for i in co
    reconstructed_signal = pywt.waverec(coefficients, wavelet, mode='per')
    signal.plot(color='b', alpha=0.5, label='original signal', lw=2, title = f'Threshold Sca
    pd.Series(reconstructed_signal, index=signal.index).plot(c='k', label='DWT smoothing', l
```

4.2.1. Zipline alpha factor research workflow (offline)

```
from zipline.api import attach_pipeline, pipeline_output, record
from zipline.pipeline import Pipeline, CustomFactor
from zipline.pipeline.factors import Returns, AverageDollarVolume
from zipline import run_algorithm
MONTH, YEAR = 21, 252
```

```
N_LONGS = N_SHORTS = 25
VOL_SCREEN = 1000
class MeanReversion(CustomFactor):
    inputs = [Returns(window_length = MONTH)]
    window_length = YEAR
    def compute(self, today, assets, out, monthly_rerturns):
        df = pd.DataFrame(monthly_returns)
        out[:] = df.iloc[-1].sub(df.mean()).div(df.std())
def compute_factors():
    mean_reversion = MeanReversion()
    dollar_volume = AverageDollarVolume(window_length = 30)
    return Pipeline(columns={'longs': mean_reversion.bottom(N_LONGS),
        'shorts': mean_reversion.top(N_SHORTS),
        'ranking': mean_reversion.rank(ascending=False)},
        screen = dollar_volume.top(VOL_SCREEN))
```

We can place long and short orders. And then, `initialize()` method registers `compute_factors()` pipeline, and `before_trading_start()` method ensures the pipeline runs daily. The `record()` function adds the pipeline's ranking column and current asset prices, to performance DataFrame returned by `run_algorithm()` function. Defind start and end timestamp objects in UTC terms, set a capital base and execute `run_algorihtm()` with references to key execution methods. Subsequent data access is easier when stored in pickle format.

```
def initialize(context):
    attach_pipeline(compute_factors(),'factor_pipeline')
def before_trading_start(context, data):
    context.factor_data = pipeline_output('factor_pipeline')
    record(factor_data = context.factor_data.ranking)
    assets = context.factor_data.index
    record(prices = data.current(assets, 'price'))
start, end = pd.Timestamp('2020-01-01', tz='UTC'), pd.Timestamp('2021-01-01', tz='UTC')
capital_base = 1e7
performance = run_algorithm(start = start,
                            end = end,
                            initialize = initialize,
                            before_trading_start = before_trading_start,
                            capital_base = capital_base)
performance.to_pickle('single_factor.pickle')
```

4.3. Info Coefficient

IR is similar to Sharpe ratio but uses a benchmark (S&P500) rather than risk-free rate. $IR = \frac{\alpha}{TrackingError}$

Information Ratio (IR) measures average excess return per unit of risk taken by dividing alpha by the tracking risk. It's better to use non-parametric Spearman rank correlation coefficient, which measures how well relationship between 2 variables be described using monotomic function, as opposed to Pearson correlation, which measures the strength of a linear relationship.

```
from alphalens.performance import factor_information_coefficient
from alphalens.plotting import plot_ic_ts
ic = factor_information_coefficient(alphalens_data)
plot_ic_ts(ic[['5D']])
ic = factor_information_coefficient(alphalens_data)
ic_by_year = ic.resample('A').mean()
ic_by_year.index = ic_by_year.index.year
ic_by_year.plot.bar(figsize=(14,6))
```

5. Portfolio Optimization and Performance Evaluation

5.1. Size your bets: Kelly criterion

How much to stake on each bet in an infinite sequence of bets with varying but favorable odds to maximize terminal wealth?

Kelly has connection to Shannon's info theory.

- b: odds defining the amount won for \$1 bet. Odds = 5/1 implies \$4 gain if bet wins plus recovery of \$1 capital.
- p: prob defining likelihood of a favorable outcome
- f: share of current capital to bet
- V: value of capital as a result of betting
- $G = \lim_{N\to\infty} \frac{1}{N} \log \frac{V_N}{V_0}$: values' growth rate, we should maximize, where W, L are number of wins and losses
- $V_N = (1 + b * f)^w (1 - f)^L V_0 \Rightarrow G = p \log(1 + b * f) + (1 - p) \log(1 - f)$

Optimal share of capital to bet (Kelly Criterion): $f^* = \frac{b*p+p-1}{b}$

5.1.1. Optimal investment: Multiple assets

```
mean_returns = monthly_returns.mean()
cov_matrix = monthly_returns.cov()
precision_matrix = pd.DataFrame(inv(cov_matrix), index = stocks, columns = stocks)
kelly_wt = precision_matrix.dot(mean_returns).values
```

The Kelly portfolio can be applie dto multi-asset application (E.Chan, 2008), it's equivalent to the potentially levered max Sharpe ratio portfolio from mean-variance optimization. Many investors prefer to reduce Kelly weights to reduce strategy's vol, and Half-Kelly is popular.

Markowitz curse: when diversification is more important because investments are correlated, conventional portfolio optimizers will likely produce an unstable solution.

Hierarchical risk parity (HRP), leverages unsupervised ML to achieve superior out-of-sample portfolio allocations. Portfolio optimization leverages graph theory an hierarchical clustering to construct a portfolio ins 3 steps (Lopez de Prado, 2015):

1. Define distance metric so that correlated assets are close to each other, and apply single-linkage clustering to identify hierarchical relationships.
2. Use hierarchical correlation structure to quasi-diagonalize cov matrix.
3. Apply top-down inverse-variance weighting using recursive biisectional search to treat clustered assets as complements, rather than substitutes, in portfolio construction and to reduce number of degrees of freedom.

Hierarchical Clustering Portfolios (HCP) (Raffinot 2016) says correlation matrices lack notion of hierarchy, which allows weights to vary freely in potentially unintended ways.

Both HRP and HCP have been tested by JP Morgan (2012), and HRP produced equal or superior risk-adjusted returns and Sharpe ratios compared to naive diversification, the max-diversified portfolios, of GMV portfolios.

5.1.2. Signal generation and trade execution

`before_trading_start()` ensures daily execution of the pipeline an the recording of resullts, including current prices. We define slippage, the cost of adverse change in price between trade decision and execution.

```
def compute_factors():
    mean_reversion = MeanReversion()
    dollar_volume = AverageDollarVolume(window_length = 30)
    return Pipeline(colums={'longs': mean_reversion.bottom(N_LONGS),
            'shorts': mean_reversion.top(N_SHORTS),
            'ranking': mean_reversion.rank(ascending=False)},
            screen = dollar_volume.top(VOL_SCREEN))

def before_trading_start(context, data):
    context.factor_data = pipeline_output('factor_pipeline')
    record(factor_data = context.factor_data.ranking)
```

```
    assets = context.factor_data.index
    record(prices = ddata.current(assets, 'price'))

def initialize(context):
    attach_pipeline(compute_factors(), 'factor_pipeline')
    schedule_function(rebalance,
                      date_rules.week_start(),
                      time_rules.market_open(),
                      calendar = calendars.US_EQUITIES)
    set_commission(us_equities = commission.PerShare(cost = 0.00075, min_tradde_cost = .01))
    set_sllippage(us_equities = slippage.VolumeShareSlippage(volume_limit = 0.0025, price_im
```

6. Machine Learning Process

Manifold learning identifies a nonlinear transformation that yields a lower-dimensional representation of data.

Root-mean-square of log of error (RMSLE) is loss function appropriate when target is subject to exponental growth.

Mutual Info (MI) between a feature and outcome is a measure of mutual dependence between 2 variables.

Bias-Variance trade-off

- Error due to bias: too simple to capture complexity, it's underfitting
- Error due to variance: overly complex, overfitting

6.1. Purging, embargoing, combinatorial CV

Financial data labels are derived from overlapping data points, because returns are computed from prices across multiple periods.

- Purging: Eliminate training data points where evaluation occurs after prediction of point-in-time data point in validation set to avoid look-ahead bias.
- Embargoing: Further eliminate training samples that follow a test period.
- Combinatorial CV: Walk-forward CV limits historical paths to be tested. Instead, given T observations, compute all possible train/test splits for $N < T$ groups that each maintain their order, and purge and embargo potentially overlapping groups. Then, train model on all combination of $N - k$ groups while testing the model on remaining k groups. The result is a much larger number of possible historical paths.

7. Linear Models for Return Forecasts

- Generalized linear models (GLM) allows for response variables that imply an error distribution other than normal distribution. GLMs include probit/logistic models for categorical response variables in classification problems.
- Robust estimation: statistical inference when ddata violates baseline assumptions due to correlation over time.
- Shrinkage: improve predictive performance of linear models, use penalty to reduce variance and improve out-of-sample predictive performance.

7.1. Baseline: Multiple linear regression

A linear functional relationship between one continuous outcome variable and p input variables, which is regression of multiple outputs on multiple input variables.

Ways to correct OLS estimates for heteroskedasticity:

- **Robust standard errors** (White standard errors) take heteroskedasticity into account when computing error variance using **sandwich estimator**.
- **Clustered standard errors** assume distinct groups in data that are homoscedastic, but error variance differs between groups. These groups could be different asset classes or equities from different industries.
- Geeralized Least Squares (GLS): for arbitrary convariance matrix structure, yields efficient and unbiased estimates in presence of heteroskedasticity or serial correlation.
- Feasible generalized least squares (GLSAR), for autocorrelated errors that follow an autoregressive AR(p) process.

Multicollinearity occurs when ≥ 2 independent variables are highly correlated.

- Difficult to determine which factors influence dependent variable.
- Individual p-value can be misleading, a high p-value but variable is important
- Confidence intervals for regression coefficients will be too wide and even including 0.

8. Backtesting

8.1. Optimal Stopping for backtests

Optimal rule is to reject the first n/e candidates and then select the first candidate that surpasses all previous options. There's $1/e$ prob of selecting the

best candidate, irrespective of size n. Similarly, we test a random sample of $1/e$ (roughly 37%) of reasonable strats outperforms those tested before, and choose a near-best as soon as possible while minimizing the risk of a false positive.

8.1.1. Vectorized vs. Event-driven backtesting

Broker handles order execution and may reject trades if not enough cash.

9. Time Series Vol Forecasts and StatArbs

How to diagnose model fit? Ljung-Box Q-statistics can test hypothesis that the residual series follows white noise.

9.1. Adding features - ARMAX

Autoregressive moving-average model with exogenous inputs (ARMAX) adds input variable or covariate on the RHS of ARMA.

Adding seasonal differencing - SARIMAX: we include AR and MA terms that captures seasonality's periodicity.

9.2. Time series to forecast vol

Impulse-response function produced by multivariate model can examine cross-series dependencies (like how policy change to IR will affect other variables over different horizons), and can simulate how 1 variable responds to sudden change in other variables. Granger causality analyzes whether 1 variable is useful in forecasting another.

Vector Autoregressive VAR(p) model can be expressed in matrix form. If some or all of k series are unit-root non-stationary, they may be cointegrated. This extension of unit root to multiple time series means that a linear combination of ≥ 2 time series is stationary and mean-reverting. So we use Vector Error Correction Model (VECM, 1990) to explore cointegration because this can form pairs-trading strat.

9.3. Cointegration - time series with a shared trend

Conintegration differs from correlation, two series can be highly correlated but not conintegrated [1].

[1] 2 growing series are const multiples of each other, their correlation is high, but any linear combination will grow rather than revert to stable mean.

If ≥ 2 asset price series revert to a common mean, we can leverage deviations from the trend, because they imply future price moves in the opposite direction. We can use **Johansen likelihood-ratio test** to see if cointegration exists, it's long term relationship.

9.4. Pairs Trading

Distance approach is simple and less computationally intensive than cointegration tests. For 150 stocks with 4 years daily data, it takes 30ms to compute correlation with returns of ETF, but 18 secs for a suite of cointegration tests.

To balance tradeoff between computational cost and quality of resulting pairs, Krauss (2017) provides a procedure that combines both approaches:

- Select pairs with a stable spread that shows little drift to reduce number of candidates
- Test remaining pairs with highest spread variance for cointegration

This can select cointegrated pairs with lower divergence risk, while ensuring more volatile spreads that generate high profit opportunities.

Many tests will lead to **data snooping bias**,since multiple testing can increase number of false positives that mistakenly reject null hypothesis of no cointegration.

We first estimate optimal number of lags that we need to specify for Johansen test. For both tests (plus Engle-Granger test), we assume the cointegrated series (spread) may have intercept $\neq 0$ but no trend.

```
def compute_pair_metrics(security, candidates):
    security = security.div(security.iloc[0])
    ticker = security.name
    candidates = candidates.div(candidates.iloc[0])
    # compute heuristics
    spreads = candidates.sub(security, axis=0)
    n, m = spreads.shape
    X = np.ones(shape = (n,2))
    X[:,1] = np.arange(1, n+1)
    drift = ((np.linalg.inv(X.T @ X) @ X.T @ spreads).iloc[1].to_frame('drift'))
    vol = spreads.std().to_frame('vol')
    corr_ret = (candidates.pct_change()
                .corrwith(security.pct_change()).to_frame('corr_ret'))
    corr = candidates.corrwith(security).to_frame('corr')
    metrics = drift.join(vol).join(corr).join(corr_ret).assign(n=n)
    tests = []
    # compute cointegration tests
```

```
    for candidate, prices in candidates.items():
        df = pd.DataFrame({'s1':security, 's2':prices})
        var = VAR(df)
        lags = var.select_order()
        k_ar_diff = lags.selected_orders['aic']
        # Johansen Test with const term and estd. lag order
        cj0 = coint_johansen(df, det_order = 0, k_ar_diff = k_ar_diff)
        # Engle-Granger Tests
        t1, p1 = coint(security, prices, trend='c')[:2]
        t2, p2 = coint(prices, security, trend='c')[:2]
        tests.append([ticker, candidate, t1, p1, t2, p2, k_ar_diff, *cj0.lr1])
    return metrics.join(tests)
```

Apply a rolling Kalman Filter to remove noise and smooth prices. Obtain dynamic hedge ratio with KF for rolling linear regression.

```
def KFSmoother(prices):
    # estimate rolling mean
    kf = KalmanFilter(transition_matrices = np.eye(1),
        observation_matrices = np.eye(1),
        initial_state_mean = 0,
        initial_state_covariance = 1,
        observation_covariance = 1,
        transition_covariance = .05)
    state_means, _ = kf.filter(prices.values)
    return pd.Series(state_means.flatten(), index=prices.index)

def KFHedgeRatio(x,y):
    delta = 1e-3
    trans_cov = delta/(1-delta)*np.eye(2)
    obs_mat = np.expand_dims(np.vstack([[x], [np.ones(len(x))]]).T, axis=1)
    kf = KalmanFilter(n_dim_obs=1, n_dim_state=2,
        initial_state_mean = [0,0],
        initial_state_covariance = np.ones((2,2)),
        transition_matrices = np.eye(2),
        observation_matrices = obs_mat,
        observation_covariance = 2,
        transition_covariance = trans_cov)
    state_means, _ = kf.filter(y.values)
    return -state_means
```

9.5. Backtesting strategy with backtrader

9.5.1. Tracking pairs with a custom DataClass

`dataclass` is a data structure called `pair` that allows us to store the pair components, number of shares and hedge ratio, and compute current spread and the return among other things.

```
@dataclass
class Pair:
    period:int
    s1:str
    s2:str
    size1: float
    size2:float
    long: bool
    hr: float
    p1: float
    p2: float
    entry_date: date = None
    exit_date: date = None
    entry_spread: float = np.nan
    exit_spread: float = np.nan
    def compute_spread(self, p1, p2):
        return p1 * self.size1 + p2 * self.size2
    def compute_spread_return(self, p1, p2):
        current_spread = self.compute_spread(p1, p2)
        delta = self.entry_spread - current_spread
        return (delta / (np.sign(self.entry_spread) * self.entry_spread))
```

10. Bayesian ML - Dynamics Sharpe Ratios and Pairs Trading

- Stochastic techniques based on **MCMC** sampling is widely used. They have property to converge to the exact result. Sampling method can be computationally demanding and is limited to small-scale problems.
- Deterministic technique: **variational inference or variational Bayes** are based on analytical approximations to posterior distribution and can scale well to large applications. They make simplifying assumptions that the posterior factorizes in particular way or has specific parametric form, such as Gaussian. So they don't generate exact results and can be used as complements of sampling methods.

10.1. MCMC Sampling

Draw samples $X = (x_1, ..., x_n)$ from given distribution $p(x)$. Monte Carlo methods is to repeatedly random sample to approximate results that may be deterministic but that don't permit an exact analytic solution.

Steps.

- Start at current position
- Draw new position from distribution
- Evaluate prob of new position from data and prior distribution
  - If sufficiently likely, move to new position
  - Otherwise remain at current position
- Repeat Step 1
- After many iterations, return all accepted positions

MCMC key property is that the process should forget about its initial position after some iteratins, initial steps are typically discarded as burn-in samples.

10.1.1. Gibbs Sampling

It simplifies multivariate sampling to a sequence of 1-dim draws. It iteratively holds $n-1$ variables const while sampling the n-th variable. It incorporates this sample and repeats it. The sequential nature prevents parallellization.

10.1.2. Metropolis-Hasting Sampling

It randomly proposes new locations based on current state, to effectively explore the sample space and reduce correlation of samples relative to Gibbs sampling.

It samples from posterior, evaluates the proposal using the product of prior and likelihood, which is proportional to posterior.

Benefits: works with a proportional rather than an exact evaluation of posterior. But take long time to converge, because random movements not related to posterior can reduce acceptance rate, so large number of steps produces only a small number of samples. Acceptance rate can be tuned by reducing the variance of proposal distribution, but smaller steps imply less exploration..

10.1.3. Hamiltonian Monte Carlo (HMC) - going NUTS

A hybrid method that leverages the first-order derivative info of gradient of likelihood. It proposes new states for exploratin and overcomes MCMC challenges.

It incorporates momentum to efficiently jump around the posterior. So it converges faster to high-dim target distribution than simpler random walk Metropolis or Gibbs sampling.

No U-Turn Sampler (NUTS) (Hoffman, Gelman 2011) is a self-tuning HMC extension that adaptively regulates the size and # of moves around the posterior before selecting a proposal. It works well on high-dim and complex posterior distributions, and allows many complex models to be fit without specialized knowledge about the fitting algorithm itself. It's the default sampler in **PyMC3**.

10.1.4. Variational Inference and Automatic Differentiation

Variational Inference (VI) approximates prob densities by optimization.

- Select a parametrized family oof prob distributions
- Find member of this family closest to target by KL divergence

Compared to MCMC, **variational Bayes converges faster and scales better to large data**. So, variational inference is the need for model-specific derivations and the implementation of a tailored optimization routine, which slows down widespread adoption.

- MCMC approxiames posterior with samples from the chain that will converge arbitrarily close to target.
- Variational algorithms approximates posterir and will have optimized results that's not guaranteed to coincide with target.

We can use **Automatic Differentiation Variational Inference (ADVI)** to automate so that user only specifies the model, expressed as program, andd ADVI automatically generates a variational algorithm. (PyMC3 supports this)

10.2. Probabilistic Programming with PyMC3 (2017)

PyMC3 uses Theano as its computational backend for dynamic C compilation and automatic differentiation. Theano is matrix-focused and GPU-enabled optimization library at MILA, which inspired TensorFlow. PyMC4 is released in Dec 2019, uses Tensorflow instead of Theano.

10.2.1. Data and indicators

Federal Reserves' Economic Data (FRED)

- Long-term spread of treasury yield curve (difference of 10-year and 3-month Treasury yields)
- UMich consumer sentiment indicator
- National Financial Conditions Index (NFCI)
- NFCI nonfinancial leverage subindex

Will US economy be in recession x months in the future? We do Bayesian inference for logistic regression. Logistic regression models the prob that economy will be in recession 12 months after month i based on k features. Likelihood combines params with data according to logistic regression. Outcome is Bernoulli random variables with success prob given by likelihood.

```
with pm.Model() as manual_logistic_model:
    # coefficients as rvs with uninformative priors
    intercept = pm.Normal('intercept', 0, sd=100)
    beta_1 = pm.Normal('beta_1', 0, sd=100)
    beta_2 = pm.Normal('beta_2', 0, sd=100)
    # likelihood transforms rvs into prob p
    # logistic regression
    likelihood = pm.invlogit(intercept +
                        beta_1 * data.yield_curve +
                        beta_2 * data.leverage)
    # outcome: Bernoulli rv with success prob
    # given by sigmoid function conditioned on actual data
    pm.Bernoulli(name = 'logit',
                p = likelihood,
                observation = data.recession)
```

10.2.2. Generalized Linear Models (GLM)

```
with pm.Model() as logistic_model:
    pm.glm.GLM.from_formula(recession ~ yield_curve + leverage,
        data,
        family = pm.glm.families.Binomial())
```

10.2.3. Exact MAP inference

```
with logistic_model:
    map_estimate = pm.find_MAP()
print_map(map_estimate)
```

PyMC3 solves optimization problem of fiding posterior point with highest density using quasi-Newton **Broyden-Fletcher-Golldfarb-Shanno (BFGS)** algorithm.

10.2.4. Approximate Inferece - MCMC

If we're only interested in point estimates for model parameters, MAP estimate would be sufficient in this model.

MCMC inferece

```
formula = 'recession ~ yield_curve + leverage + financial_conditions + sentiment'
with pm.Model() as logistic_model:
    pm.glm.GLM.from_formula(formula = formula,
                    data = data,
                    family = pm.glm.families.Binomial())
# pymc3 uses y for outcome
logistic_model.basic_RVs
```

Variables on different scales slow down the sampling, so we apply `scale()` by scikit-learn to standardize all features. For MCMC sampling, can use `pm.sample()`.

Sampling can be parallelized for multiple chains with `cores` argument (except when using GPU)

```
with logistic_model:
    trace = pm.sample(draws = 100,
                      tune = 1000,
                      init = 'adapt_diag',
                      chains = 4,
                      cores = 4,
                      random_seed = 42)
plot_traces(trace, burnin=0)
# See figure 10.6
```

10.2.5. Approximate inference - variational Bayes

Use `fit()` instead of `sample()`, and use early stopping `CheckParametersConvergence` callback if distribution-fitting process converges up to a given tolerance.

```
with logistic_model:
    callback = CheckParametersConvergence(diff = 'absolute')
    approx = pm.fit(n=100000,
                    callbacks = [callback])
```

10.3. Bayesian Model Diagnostics

PyMC3 has `pm.summary()`. Highest posterior density (HPD) estimate the min width credible interval [2] and is computed at 95% level. $\hat{R}$ is Gelman-Rubin statstic, checks convergence by comparing variance between chains to variance within each chain. If sampler converged, these variances should be identical and the statistics ≈ 1.

For high-dim models, we can use NUTS energy plot to assess problems of convergence, which shows how efficiently random process explores posterior.

Posterior Predictive Checks (PPCs) examines how well a model fits data.

Bayesian rolling regression for pairs trading, see Figurre 10.18, a great picture combining price series and regression lines, where the hue indicates the timeline.

11. Random Forests - Long-Short trading strategy

Bootstrap aggregation = bagging: randomize the construction of individual models and reduce correlatin of prediction errors made by ensemble's components. Bagging reduces variance.

11.1. Decision Trees

Sequentially apply a rule that split data into subset and make prediction for each subset. It can capture interdependece among features that linear mdels can't capture out of the box.

- Classification trees: predict prob estimated from relative class frequencies or value of majority class
- Regressin trees: compute prediction from mean of outcome values of data points.

Tree learning takes top-down greedy approach: recursive binary splitting to overcome computational limitation.

```
from sklearn.tree import DecisionTreeRegressor
# configure regression tree
regression_tree = DecisionTreeRegressor(criterion = 'mse',
            max_depth = 6,
```

[2] Credible intervals is the Bayesian counterpart of confidence intervals, as percentiles of the trace. See figure 10.8.

```
            min_samples_leaf = 50)
# create training data
y = dadta.target
X = data.drop(target, axis=1)
X2 = X.loc[:,['t-1','t-2']]
# fit model
regression_tree.fit(X=X2, y=y)
# fit OLS model
ols_model = sm.OLS(endog = y, exog=sm.add_constant(X2)).fit()
```

Classification tree. Gini impurity, cross-entropy are more sensitive to node purity than classification error rate.

```
# randomize train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y_binary, test_size = 0.2, ranom_stat
# configure & train tree learner
clf = DecisionTreeClassifier(criterion = 'gini',
                max_depth = 5,
                random_state = 42)
clf.fit(X = X_train, y=y_train)
# output
DecisionTreeClassifier(class_weight = None,
            criterion = 'gini', max_depth = 5,
            max_features = None, max_leaf_noeds = None,
            min_impurity_decrease = 0.0, min_impurity_split = None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf = 0.0, presort=False, random_state=42,
            splitter = 'best')
```

Use Graphviz library to visualize the tree.

Advantages

- easy to understand, white-box
- require less data prep
- some decision tree handle categorical input, no need for creation of dummy variables to improve memory efficiency; can work with missing values
- prediciton is fast because of logarithmic.
- validate model with stat tests and see its realiability

Disadvantages

- can overfit, high generation error, so we need pruning and regularization use early-stop to limit tree growth

- sensitive to unbalanced class weights: biased trees. We can oversample the underrepresented classes or undersample more frequent class.
- high variance due to ability to closely adapt to training set. So, minor variations in data can produce wide swings in tree's structure and model predictions. We can use an ensemble of randomized decision trees that have low bias and produce uncorrelated predictin errors
- greedy approach optimizes local criteria that reduce prediction error at the current node and don't guarantee a global optimal result. We can use ensemble.

11.2. Random Forests - making trees more reliable

2 ensemble methods:

- Averaging methods: train several base estimators andd take average.
- Boosting methods: train base estimators sequentially with specific goal of reducing bias of combined estimator. Motivation is to combine several weak models into a poweful ensemble.

11.2.1. Bootstrap aggregation = Bagging

Bagging is random samples with replacement. Such a random sample has the same number of observations as original dataset but may contain duplicates due to replacement.

Bagging works best for complex models with low bias and high variance, to limit overfitting.

```
noise = .5 # noise relative to std(y)
noise = y.std() * noise
X_test = choice(x, size=test_size , replace = False)
X_test = choice(x, size=test_size, replace=False)
max_depth = 10
n_estimators = 10
tree = DecisionTreeRegressor(max_depth = max_depth)
bagged_tree = BaggingRegressor(base_estimator=tree, n_estimators = n_estimators)
learners = {'Decision Tree': tree, 'Bagging Regressor': bagged_tree}
predictions = {k: pd.DataFrame() for k,v in learners.items()}
for i in range(reps):
    X_train = choice(x, train_size)
    y_train = f(X_train) + normal(scale = noise, size= train_size)
    for label, learner in learners.items():
        learner.fit(X = X_train.reshape(-1,1), y=y_train)
        preds = pd.DataFrame({i: learner.predict(X_test.reshape(-1,1))},
```

```
            index = X_test)
        predictions[label] = pd.concat([predictions[label], preds], axis=1)
```

Sample size for features

- Classification: sample size $\approx \sqrt{\#features}$
- Regressin: 1/3 to all features, selected based on cross-validation

11.2.2. Random Forest Pros and Cons

Advantages

- Perform on par with the best supervised learning algorithms
- Reliable feature importance estimate
- Efficient estimates of test error without incurring the cost of repeated model training associated with cross validation

Disadvantages

- Less interpretable than individual decision tree
- Train many deep trees have high computational costs (but can be parallelized) and use much memory. Predictions slower, challenges for low latency

LightGBM: for gradient boosting, fast and memory-efficient. Can efficiently encode categorical variables as numeric features rather than using one-hot dummy encoding.

```
for train_length, test_length in test_params:
    n_splits = int(2 * YEAR / test_length)
    cv = MultipleTimeSeriesCV(n_splits = n_splits,
                    test_period_length = test_length,
                    lookahead = lookahead,
                    train_period_length = train_length)
    label = label_dict[lookahead]
    outcome_data = data.loc[:, features + [label]].dropna()
    lgb_data = lgb.Dataset(data=outcome_data.drop(label, axis=1),
            label= outcome_data[label],
            categorical_feature = categoricals,
            free_raw_data = False)

for p, (bagging_fraction, feature_fraction, min_data_in_leaf)\
         in enumerate(cv_params_):
```

```
params = base_params.copy()
params.update(dict(bagging_fraction = bagging_fractin,
            feature_fraction = feature_fraction,
            min_data_in_leaf = min_data_in_leaf))
start = time()
cv_preds, nrounds = [], []
for i, (train_idx, test_idx) in \
        enumerate(cv.split(X=outcome_data)):
    lgb_train = lgb_data.subset(train_idx.tolist()).construct()
    lgb_test = lgb_data.subset(test_idx.tolist()).construct()
    model = lgb.train(params=params,
                      train_set = lgb_train,
                      num_boost_round = num_boost_round,
                      verbose_eval = False)
    test_set = outcome_data.iloc[test_idx, :]
    X_test = test_set.loc[:, model.feature_name()]
    y_test = test_set.loc[:, label]
    y_pred = {str(n): model.predict(X_test, num_iteration=n)
        for n in num_iterations}
    cv_preds.append(y_test.to_frame('y_test')
        .assign(**y_pred).assign(i=i))
    nrounds.append(model.best_iteration)
```

12. Boosting Trading Strategy

12.1. Adaptive Boosting (AdaBoost)

Ensemble algorithm to iteratively adapt to cumulative learning progress when fitting an additional ensemble member. Adaboost changed weights on training data to reflect cumulative errors of current ensemble on training set, before fitting a new, weak learner. It's the most accurate classification algorithm at the time.

Adaboost grows shallow trees as weak learners, often producing superior accuracy with stumps: trees formed by a single split. We start with equally weighted training set, and successively alters sample distribution. After each iteration, AdaBoost increases the weights of incorrectly classfied observations and reduces weights of correctly predicted samples so that subsequent weak learners focus more on difficult cases. Once trained, the new decision tree is incorporated into ensemble with a weight that reflects its contribution to reducing training error.

Adaboost for ensemble of base learners $h_m(x)$ predicts discrete classes $y \in [-1, 1]$, N training observationns:

- Initialize sample weights $w_i = 1/N$ for observations $i = 1, ..., N$.

- For each base classifier h_m:
  - Fit $h_m(x)$ to training data, weighted by w_i.
  - Compute base learner's weighted error rate ϵ_m on training set.
  - Compute base learner's ensemble weight $\alpha_m = \log(\frac{1-\epsilon_m}{\epsilon_m})$ as function of its error rate.
  - Update weights for misclassified samples by $w_i * \exp(\alpha_m)$
- Predict positive class when weighted sum of ensemble members is positive, and negative otherwise: $H(x) = \text{sign}\left(\sum_{\text{m=1}}^{\text{M}} \alpha_{\text{m}} \text{h}_{\text{m}}(\text{x})\right)$

Advantages

- Ease of implementation and fast computation
- Can combine with any method for identifying weak learners
- No need to tuen hyperparameters
- Can identify outliers because samples receiving highest weights are consistently misclassified and inherently ambiguous, which is typical for outliers

Disadvantages

- Adaboost on given dataset depends on ability of weak learner to adequately capture relationship between features and outcme.
- Boosting not perform well when there is insufficient data, or when complexity of ensemble members is not a good match for the complexity of data

12.2. Gradient Booosting - Ensembles for most tasks

AdaBoost is a stagewise forward approach to minimize an exponential loss function for a binary outcome y that identifies a new base learner h_m, at each iteration m with weight α_m, and adds t ensemble

$$\arg\min_{\alpha,h} \sum_{i=1}^{N} \exp\left(-y_i\left(f_{m-1}\left(x_i\right)\right) + \alpha_m h_m\left(x_i\right)\right)$$

In exp, there is current ensemble + new member. AdaBoost is a gradient descent algorithm that minimizes exponential loss function.

Gradient boosting applies boosting method t a wider range of loss functions.

Gradient Boosting machines (GBMs) is training the base learners to learn negative gradients of current loss function of ensemble. So each addition to

ensemble directly contributes to reducinng overall training error, given errors made by prior ensemble members.

Gradient boosting optimizes over the functions h_m in an additive fashion. Its success is based on the ability to learn complex functional relationships in an incremental fashion. But we need to pay attention to risk of overfitting by tuning hyperparameters to contrain the model learning noise.

12.3. Train and Tune GBM models

We can use shrinkage to regularize, by scaling the contribution of each new ensemble member down by a factor $\in [0, 1]$. This factor is learning rate of boosting ensemble. Reducing learning rate increases shrinkage, because it lowers the contribution of each new decision tree to ensemble.

Lower learning rates + larger ensembles = reduce test error inn regression / prob estimation.

We can use adaptive learning rates to lower impact of trees added later in process.

12.3.1. Subsampling and Stochastic gradient boostinng

Stochastic gradient boosting samples training data withouot replacement at each iteration to grow the next tree (but bagging with replacement). Benifit: lower computation due to smaller sample and better accuracy, but should combine with shrinkage.

12.4. XGBoost, LightGBM and CatBoost

Random forests can be trained in parallel by growing individual trees on independent boostrap samples. **Sequantial approach of gradient boosting** slows down training, and have large number of hyperparameters to tune. Computational cost during training $\propto$ time it takes to evaluate potential split points for each feature.

Regularization penalty help avoid overfitting by favoring a model that uses simple yet predictive regression trees. In XGBoost, penalty for regression tree h depends on # of leaves per tree T, regression tree scores for each terminal noed w, and hyperparameters γ, λ. At each step, algorithm greedily adds the hypothesis h_m that most improves regularized objective.

12.4.1. Simpliefied split-finding algorithms

Approximate split-finding reduces # of split points by assigning feature values to user-determined set of bins, which can greatly reduce memory requirements

during training, because only a single value needs to be stored for each bin. XGBoost is a quantile sketch algorithm that divides weighted training samples into percentile bins to achieve uniform distribution. XGBoost introduce ability to handle sparse dadta cause by missing values, frequent zero-gradient stats, and one-hot encooding, and can learn optimal default direction for a given split. So the algorithm only needs to evaluate non-missing values.

In contrast, LightGBM uses gradient-based 1-side sampling (GOSS) to exclude significant proportion of samples with small gradients, and only uses remainder to estimate info gain and select a split value accordingly. Samples with larger gradients require more training and tend to contribute more to info gain.

LightGBM uses exclusive feature bundling to combine features that are exclusive, they rarely take nonzero values simultaneously, and can reduce # of features. So, LightGBM is the fastest implementation.

12.4.2. Depth-wise vs. Leaf-wise growth

LightGBM differs from XGBoost and CatBoost in how it prioritizes which nodes to split. LightGBM decides n splits leaf-wise (split leaf node that maximizes info gain, even when this lead to unbalanced trees). This increases model complexity and speed up cnvergence, and may overfit.

But XGBoost and CatBoost expand all nodes depth-wise and first split all nodes at a given level of depth, before adding more levels. They expand nodes in different order and prduce different results except for complete trees.

12.4.3. DART (dropout for additive regression trees)

Rashmi Gilad-Bachrach (2015): train gradient boosting trees to address over-specialization problem: trees added during later iterations only affect predictin of a few instances, while making minor contribution to remaining instances. But the model's out-of-sample performance will suffer, and may be oversensitive to contributins of small number of trees. Nodes in higher layers can't rely on a few connections to pass info for prediction.

DART operates at the level of trees and mutes complete trees as opposed to individual features. Goal is for trees in ensemble generated using DART to contribute more evenly toward final prediction.

12.4.4. Categorical Features

CatBoost and LightGBM handles categorical variables directly without need for dummy encoding.

CatBoost implementation has several options to handle features, in addition to automatic one-hot encoding. It assigns either categories of individual features or combinations of categories for several features to numerical values.

CatBoost creates new categorical features from combinations of existing features. Numerical values associated with category levels of individual features or combinations of features depend on their relationship with outcome value. For classification, it's related to prob of observing positive class, computed cumulatively over the sample based on prior, and with smoothing factor.

LightGBM groups the levels of categorical features to maximize homogeneity (min variance) within groups w.r.t. outcome values. XGBoost doesn't handle categorical features directly and requires one-hot (dummy) encoding.

12.4.5. Additional features and optimizations

XGBoost optimizes computation for multithreading. It keeps data in memory in compressed column blocks, each column is sorted by corresponding feature value. It computes input layout once befoore training and reuses it throughouot to amortize upfront cost. So, the search for split stats over columns is linear scan of quantiles that can be donn in parallel and supports column subsampling.

LightGBM further accelerated training throuogh optimized threading and reduced memory usage.

XGBoost supports monotonicity contraints. These contraints ensure the values for a given feature are only positively or negatively relate dto outcome over entire range.

12.5. Long Short Trading Strat with boosting

LightGBM and CatBoost are written in C++ and translate Python objects into binary data formats before precomputing feature stats to accelerate search for split points.

```
data = (pd.read_hdf('data.h5', 'model_adta')
        .sort_index()
        .loc[idx[:,:'2015'],:])
labels = sorted(data.filter(like='fwd').columns)
features = data.columns.difference(labels).tolist()
categoricals = ['year','weekday','month']
for feature in categoricals:
    data[feature] = pd.factorize(data[feature], sort=True)[0]

import lightgbm as lgb
outcome_data = data.loc[:, features + [label]].dropna()
```

```
lgb_data = lgb.Dataset(data=outcome_data.drop(label, axis=1),
        label=outcome_data[label],
        categorical_feature = categoricals,
        free_raw_data = False)

cat_cols_idx = [outcome_data.columns.get_loc(c) for c in categoricals]
catboost_data = Pool(label = outcome_data[label],
        data = outcome_data.drop(labels, axis=1),
        cat_features = cat_cols_idx)

for i, (train_idx, test_idx) in enumerate(cv.split(X=outcome_data)):
    lgb_train = lgb_data.subset(train_idx.tolist()).construct()
    train_set = catboost_data.slice(train_idx.tolist())
```

Note: CatBoost implementation needs feature columns to be identified using indices rather than labels.

12.6. Evaluation

Cross Validation: LightGBM vs. CatBooost: LightGBM performs slightly better than CatBoost, for long horizons, because we ran more configurations for LightGBM.

12.6.1. Feature Importance

- Gain (contribution of a feature to reduce loss)
- Split count (count how often a feature is used to make a plist decision, based on selection of features based on info gain)
- Permutation (randomly permutes feature values in a test set and measure how much model's error change)

Partial dependence plots (3D plot see Figure 12.14)

12.6.2. SHapley Additive exPlanatins (SHAP)

Provide granular feature attribution at the level of each individual prediction andd enable much richer inspection of complex models through interactive visualization. (See also Kaggle Course Notes under https://github.com/junfanz1/CS-Online-Course-Notes/)

12.7. Boosting for Intraday Strat

Use AlgoSeek NASDAQ 100 dataset, minute bar frequency. Strat is driven by ensemble of gradient boosting models profitable before significant trading costs. See notebook.

12.8. Summary

Gradient boosting builds ensembles in a sequential manner, adding a shallow decision tree with small number of features to improve on predictions.

13. Data-Driven Risk Factors & Asset Allocation w/ Unsupervised Learning

13.1. PCA vs. ICA

Dimensionality reduction

- Linear algo. PCA/ICA constrain new variables to be linear combinations of original features. PCA requires new features to be uncorrelated, ICA goes further and imposes stats independence, implying absence of both linear and nonlinear relationships.
- Nonlinear algo. t-distributed stochastic neighbor embedding (t-SNE), Uniform Manifold Approximation and Projection (UMAP) for high-dim data.

PCA Assumptions

- High variance implies high signal-to-noise ratio
- Standardized data, so variance is comparable across features
- Linear transformations capture relevant aspects of data
- High-order stats, beyond first/second moments don't matter, which implies data has normal distribution.

Independent Component Analysis (ICA) is a linear algorithm that can deal with blind source separation. n original signals, and unknown square matrix A with n-dim set of m observations. Our goal is to find matrix $W = A^{-1}$ that untangles the mixed signals to recover sources.

ICA Assumptions

- Source of signals are statistically independent

- Linear transformations are sufficient to capture the relevant info
- Independent components don't have normal distribution
- Maxing matrix A can be inverted.
- Data should be centered and whitened (mutually uncorrelated with unit variance), so we can preprocessinng data by PCA.

`FastICA` by sklearn.

13.2. Manifold learning - nonlinear dim reduction

A manifold is a space locally resembles Euclidean space. Manifold hypothesis maintains that high-dim adta often resides in a lower-dim space, if identified, permites a faithful representation of data in this subspace.

Technique to approximate a lower-dim manifold: locally linear embedding (LLE, 2000), which identifies a given number of nearest neighbors and computes weights that represent each point as a linear combination of its neighbors. It finds lower-dim embeding by linearly projecting each neighborhood on global internal coordinates on lower-dim manifold and can be thought of as a sequence of PCA applications.

13.2.1. t-distributed Stochastic Neighbor Embedding

t-SNE (2008) detects patterns in high-dim data. By converting high-dim distances into conditional prob, where high prob imply low distance and reflect likelihood of sampling 2 points based on similarity. It first position a normal distribution over each point and compute density for a point and each neighbor. Second, it arranges points in low dim and uses similarly computed low-dim prob to match high-dim distribution by KL divergence, which puts high penalty on misplacing similar points in low dim. The low-dim prob use t-distribution with 1 degree of freedome, because it has fatter tails that reduces penalty of misplacing points that are more distant in high dim to manage crowing problem.

Weakness

- computational complexity scales quadratically in n points, as it evaluates all pairwise distances, but a subsequent tree-based implementation can have $n \log n$.
- cannot facilitate projection of new data points into low-dim space. The compressed output is not useful input for distance- or density-based cluster algo, because t-SNE treats small and large distances differently.

13.2.2. Uniform Manifold Approximation and Projection (UMAP)

Assumption

- Data is uniformly distributed on locally connected manifold and looks for closest low-dim equivalent usign fuzzy topology.

Faster and scales better to large datasets than t-SNE, and preserves global structure better than t-SNE.

13.3. Clustering

13.3.1. Density-based spatial clustering of applications with noise (DBSCAN, 1996)

Identiy clusters that differ in shape significantly from k-means.

13.3.2. Gaussian Mixture Models (GMM)

Generative models that assume data has been generated by a mix of various multivariate normal distributions.

GMM generalizes k-means: GMM adds covariance among features so that clusters can be ellipsoids rather than spheres, while the centroids are represented by means of each distributionn. It performs soft assignments because each point has a prob of being a member of any cluster.

13.3.3. Hierarchical clustering for optimal portfolios

Hierarchical risk parity (HRP, by Prado 2016) levarages hierarchical clustering to assign position sizes to assets based on risk characteristics of subgroups.

14. Text Data - Sentiment Analysis with NLP

N-grams

Bag of words: represent documents as word vectors

TF-IDF

`spaCy`

15. Topic Modeling - Financial News

15.1. LDA

Reduce dimensionality of DTM (document-term matrix) with latent semantic indexing (LSI)

Extract topics with prob latent semantic annalysis (pLSA): equivalent to Non-negative matrix factorization (NMF) using KL divergence

Latent Dirichlet allocation (LDA) improves pLSA by adding a generative process for topics (2003). It's a hierarchical Bayesian model that assumes topics are prob distributions over words, and documents are distribution over topics following sparse Dirichlet distribution, which implies that documents reflect only a small set of topics and topics use only a limited number of terms frequently.

LDA describes: % contribution of each topic to a document; and prob association of each word with a topic.

LDA solves Bayesian inference problem of recovering distributions from body of documents, and words they contain by reverse engineering the assumed content generation process.

15.2. Evaluation on LDA

- Perplexity: evaluates how well topic-word prob distribution recovered by model predicts a sample of unseen documents, based on entropy $H(p)$ of distribution.
- Topic coherence: evaluates semantic consistency of uncovered patterns, whether human perceive the words and prob associated with topics meaningful.

UCI metric (2012): a word pair's score = pointwise mutual info (PMI) between 2 distinct pairs of topic words and a smoothing factor. Prob is computed from word co-occurrence frequencies in sliding window over external corpus, so that it's a comparison to semantic ground truth.

We can visualize LDA results by `pyLDAvis`. See figure 15.13, impact of LDA hyperparameter settings on topic quality.

16. Word Embeddings for Earnings Calls and SEC Filings

Attention mechanism produces more context-sensitive sentence representatins, can be in transformer architectures like BERT family of models.

GloVe - Global vectors for word representation

Visualizing embeddings with TensorBoard

Sentiment analysis using doc2vec embeddings `gensim.models.Doc2Vec`

16.1. Pretrained transformer

Attention mechanism can learn more contextualized word embeddings (2017).

- Bidirectional language models, process next both left-to-right and right-to-left sequentially in RNN [3] for a richer context representation
- Semi-supervised pretraining onn large generic corpus to learn universal language aspects in the form of embeddinng and network weights, can be used and finetuned for specific tasks (transfer learning)

Transformer model uses encoder-decoder architecture with several layers, each using several attentin mechanisms (called heads) in parallel. 2018 Google's Bidirectional Encoder Representation from Transformers (BERT).

Key innovation: deeper attentin and pretraining

- **transformer architecture**. Takes attention mechanism t new deeper level by 12 or 24 layers, each with 12 oor 16 attention heads. So 24 * 16 = 384 attention mechanism to learn context-specific embeddings.
- **unsupervised, bidirectional pretraining**: doesn't need to be trained from scratch for each new task, but weights are fine-tuned. It has 2 tasks: masked language modeling (predict a missing word given left and right) and next sentence prediction (predict whether one sentence follow another)

17. Deep Learning for Trading

- L1 regularization produces sparse parameter estimates by reducing weights all the way to 0.
- L2 regularization preserves direction along which the params significantly reduce the cost function.

[3] In contrast, Transformer requires onlly a const # of steps to identify semantically related words. It has self-attention mechanism capturing links between all words in a sentence regardless of relative position. The model learns representation of a word by assigning an attention score to every other word in the sentence that determines how much each of other words should contribute to representation. These scores inform a weighted average of all words' representations, which is fed into a fully connected network to generate new representation for the targeted word.

Dropout: randomized omission of individual units with a given prob during forward or backward propagation. So these omitted units don't contribute to training error or receive updates. It reduces risk of overfitting by preventing units from compensating for mistakes by other units during training.

17.1. Stochastic Gradient Descent

Too high learning rate can lead to repeated overshooting and oscillation around or divergence from minimum.

Momentum can accelerate cnvergence to local min. Momentum can track recent directions and adjust params by a weighted average of most recent gradient and current value with momentum term γ to weight contribution ooff latest adjustment to iteration's update $v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$.

Nesterov momentum (2013) is simple change to normal momentum. Gradient term is not computed at current parameter space position θ_t but instead from an imtermediate position. Goal is to correct foor the momentum term overshooting or pointing in wrong direction.

Adam = adaptive moment deviation. Hyperparameters:

- α = learning rate/step size, determine how much weights are updated so that larger values speed up learning before the rate is updated, by default 0.001.
- β_1 = exponential decay rate for first moment estimates, by default 0.9
- β_2 = exponential decay rate for secondn moment estimates, by default 0.999
- ϵ = small number to prevent division by 0, by default 1e-8.

TensorBoard is visualization tool.

17.2. PyTorch 1.4

It has more granular control than Keras through lower-level API.

Difference to Tensorflow: it emplys eager execution, but Tensorfloow use static computatin graphs. PyTorch has autograd package for automatic differentiation of tensor operations - compute gradients on the fly, so that network structures can be partially modified more easily, which is called **define-by-run**: backpropagatin is defined by how your code runs, which implies that every single iteration can be different.

We convert NumPy or pandas input data to `torch` tensors. Use PyTorch tensors to instantiate first `TensorDataset` and second a `DataLoader` that includes info about `batch_size`.

```
import torch
X_tensor = torch.from_numpy(X)
y_tensor = torch.from_numpy(y)
X_tensor.shape, y_tensor.shape
(torch.Size([50000,2]), torch.Size([50000]))

import torch.utils.data as utils
dataset = utils.TensorDataset(X_tensor, y_tensor)
dataloader = utils.DataLoader(dataset,
                              batch_size = batch_size,
                              shuffle=True)
```

17.2.1. Network Architecture

```
import torch.nn as nn
class Net(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(Net, self).__init__() # inherit frm nn.Module
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.logistic = nn.LogSigmoid()
        self.fc2 = nn.Linear(hidden_size, num_classes)
        self.softmax = nn.Softmax(dim=1)
    def forward(self, x):
        # forward pass: stacking each layer together
        out = self.fc1(x)
        out = self.logistic(out)
        out = self.fc2(out)
        out = self.softmax(out)
        return out
# ...
```

18. CNN for Financial Time Series and Satellite Images

Satellite data can signal future commodity trends, including supply of certain crops or raw materials via aerial images of agricultural areas, mines, transport networks like oil tankers. Surveillance camera footage from shopping malls can track and predict consumer activity.

CNN learn to model grid-like data

LeNet5

18.1. Transfer Learning - faster learning with less data

Alternative approach to transfer learning: CNN's last convolutional layer's output feeds into fully connected part is the **bottleneck features**. Bottleneck features of pretrained network as inputs can be a new fully connected network, after ReLU. So, we freeze the convolutinal layers and replace dense part of network, and we can use inputs of different sizes because it's the dense layers that constrain input size.

VGNNet: more depth and smaller filters

GoogLeNet: fewer params through inception

ResNet: shortcut connections beyond human performance

Object detection and segmentation: YOLO

18.2. Summary

CNN can extract meaningful info from time series of alpha factors converted into 2-dim grid. Result is not robust and slight modification yields significantly worse performance. Tuning surface the notorious difficulties in successfully training deep NN, especially when signal-to-noise ratio is low, too complex network or wrong optimizer can lead CNN to local optimum where it predicts const value.

19. RNN for Multivariate Time Series Sentiment Analysis

Unfolding a computational graph with cycles.

Bidirectional RNN combine an RNN that moves forward with another RNN scanning sequence in the opposite direction.

Encoder-decoder architectures = seq2seq (encoder: RNN that maps the input space to different space = laten space; decoder: complementary RNN that maps the encoded input to target space)

Attention mechanism addresses a limitation of using fixed-size encoder inputs when input sequences themselves vary in size. A recent transformer architecture (2017) dispenses with recurrence and convolutions and exclusively relies on this attentionn mechanism to learn input-output mappings.

19.1. Design Deep RNN

We can stack recurrent layers on top of each other, so that they learn a hierarchical temporal representation of input data. A lower layer may capture

higher-frequency patterns, synthesized by a higher layer into lower frequency characteristics that prove useful for classification or regression.

Repeated multiplication oon gradients during backpropagation over many time steps will cause **gradients vanish/explorde**.

19.1.1. LSTM - learning how much to forget

Combines 4 parameterized layers that interact with each other and the cell state, by transforming and passing along vectors.

- Forget gate controls how much cell's state should be voided to regulate network's memory.
- Input gate computes sigmoid activation that produces update candidates.
- Output gate filters updated cell state using sigmoid activation, and multiplies it by cell state normalized to $[-1, 1]$ range by tanh activation.

19.1.2. GRU - Gated Recurrent Units

GRU simplifies LSTM units by omitting output gate. They acheive similar performance on language modeling tasks, but do better on smaller datasets.

GRU aim for each recurrent unit to adaptively capture dependencies of different time scales. Similar to LSTM unit, GRU has gating units that modulate flow of info inside unit, but discard separate memory cells.

20. Autoencoders for Conditional Risk Factors and Asset Pricing

20.1. VAE

Autoencoder is a NN trained to reproduce the input while learning a new representation of data, encoded by parameters of a hidden layer. It can be used for nonlinear dimensionality reduction, manifold learning.

Variatinal autoencoders (VAE) is generative modeling, which solves the more general problem of learning a joint prob distribution over all variables. Learning the data generatin process reveals underlying causal relatinships and supports semisupervised learning to generalize froom a small laveled dataset to a large unlabeled one.

VAE can learn latent (unobserved) variables of model responsiblle for input data. If successful, you can generate new data points by sampling from distribution learned by VAE.

We can use t-SNE to visualize the encoding.

20.2. Convolutional Autoencoders

On CNN we can incorporate convolutinal layers into autoencoder to extract info characteristic of grid-like structure of image data.

20.3. Conditional Autoencoder for traing

Gu, Kelly and Xiu (GKX, 2019): asset pricing model based on exposure oof securities to data-driven risk factors. Asset characteristics used by factor models to capture systematic drivers of anomalies are just proxies for time-varying exposure to risk factors that can't be directly measured. Anomalies are returns in excess of those explained by exposure to aggregate market risk. GKX treats risk factors as latent, non-observable drivers of covariance among many assets large enough to prevent investors from avoiding exposure through diversification. So, investors require a reward that adjusts like any price to achieve equilibrium, providing in turn an economic rationale for return differences that are no longer anomalous. So, risk factors are purely statistical in nature while the underlying economic forces can be of arbitrary and varying origin.

Kelly, Pruitt, Su (2019): a linear model: instrumental pricinpal component analysis (IPCA) to estimate latent risk factors and the assets' factor loading from dadta. IPCA extends PCA to include asset characteristics as covariates and produce time-varying factor loadings. By conditining asset exposure to factors on observable asset characteristics, IPCA aims to answer whether there's a set oof common latent risk factors that explain an observedd anomaly rather than whether there's a specific observable factor.

GKX creates a conditional autoencoder to reflect nonlinear nature of return dynamics ignoredd by linear Fama-French models and IPCA approach. Result: deep NN simultaneously leanrs the premia on given number of unobservable factors using autoencoder, and factor loadings for a large universe of equities based on broad range of time-varying asset characteristics using feedforward network. So this model can explain and predict asset returns, with Sharpe ratio when doing long-short decile spread strategy.

```
from pandas_datareader.nasdaq_trader import get_nasdaq_symbols
traded_symbols = get_nasdaq_symbols()
import yfinance as yf
tickers = yf.Tickers(traded_symbols[~traded_symbols.ETF].index.to_list())
info = []
for ticker in tickers.tickers:
    info.append(pd.Series(ticker.info).to_frame(ticker.ticker))
info = pd.concat(info, axis=1).dropna(how='all').T
info = info.apply(pd.to_numeric, errors='ignore')
price_adj = []
with pd.HDFStore('chunk.h5') as store:
```

```
    for i, chunk in enumerate(chunks(tickers, 100)):
        print(i, end=' ', flush=True)
        prices_adj.append(yf.download(chunk,
                                      period = 'max',
                                      auto_adjust = True).stack(-1))
price_adj = (pd.concat(prices_adj)
                .dropna(how = 'all', axis=1)
                .rename(columns=str.lower)
                .swaplevel())
price_adj.index.names = ['ticker', 'date']
df = prices_adj.close.unstack('ticker')
pmax = df.pct_change().max()
pmin = df.pct_change().min()
to_drop = pmax[pmax>1].index.union(pmin[pmin<-1].index)
# ...

# stock momentum, 11-month cumulative stck returns ending 1 month before current date
MONTH = 21
mom12m = (close
            .pct_change(periods=11*MONTH)
            .shift(MONTH)
            .resample('W-FRI')
            .last()
            .stack
            .to_frame('mom12m'))
# Amihud Illiquiddity measure = ratio of stock's absolute returns relative to ..
# .. its dollar volume, measrued as rolling 21-day average
dv = close.mul(volume)
ill = (close.pct_change().abs()
        .div(dv)
        .rolling(21)
        .mean()
        .resample('W-FRI').last()
        .stack()
        .to_frame('ill'))

# Idisyncratic vol: std of regression of residuals of weekly returns ..
# .. on the returns of equally weighted market index returns for prior 3 years

index = close.resample('W-FRI').last().pct_change().mean(1).to_frame('x')
def get_ols_residuals(y, x=index):
    df = x.join(y.to_frame('y')).dropna()
    model = sm.OLS(endog = df.y, exog = sm.add_constant(df[['x']]))
    result = model.fit()
    return result.resid.std()
idiovol = (returns.apply(lambad x: x.rolling(3 * 52)
```

```
                        .apply(get_ols_residuals)))

# market beta
def get_market_beta(y, x=index):
    df = x.join(y.to_frame('y')).dropna()
    model = RollingOLS(endog = df.y,
                       exog = sm.add_constant(df[['x']]),
                       window = 3*52)
    return model.fit(params_only = True).params['x']
beta = (returns.dropna(thresh = 3*52, axis=1)
        .apply(get_market_beta).stack().to_frame('beta'))
```

Conditional autoencoder by GKX allows for time-varying return distributions that consider changing asset characteristics. See Figure 20.7.

```
def make_model(hidden_units = 8, n_factors = 3):
    input_beta = Input((n_tickers, n_characteristics), name='input_beta')
    input_factor = Input((n_tickers,), name='input_factor')
    hidden_layer = Dense(units = hidden_units,
                         activation = 'relu',
                         name = 'hiden_layer')(input_beta)
    batch_norm = BatchNormalization(name='batch_norm')(hidden_layer)
    output_beta = Dense(units = n_factors, name='output_beta')(batch_norm)
    output_factor = Dense(units = n_factors, name = 'output_factor')(input_factor)
    output = Dot(axes=(2,1), name='output_layer')([output_beta, output_factor])
    model = Model(inputs = [input_beta, input_factor], outputs = output)
    model.compile(loss= 'mse', optimizer = 'adam')
    return model
```

21. Generative Adversarial Networks for Synthetic Time Series

21.1. GANs

GANs is a generative model like VAE, and can generate high-quality samples that mimic a range of input data. GANs can improve supervised learning performance.

Deep convolutional GANs (DCGANs) use CNN to do supervised learning for grid-like data (2016), which can be used for GANs for unsupervised learning by developing a feature extractor based on adversarial training.

Conditional GANs (cGANs) has additional label info into the training process, and it has better quality with some control over he output.

Generative adversarial what-where network (GAWWN, 2016) uses bounding box info not only to generate synthetic images, but also to place objects at a given location.

CycleGAN - unpaired image-to-image translation. CycleGAN pairs images that are not available and transforms images from one domain to match another. *Ex*: Synthetic paiting of horses as zebras or impressionistic photo.

StackGAN - text-to-photo image synthesis: uses a sentence as input and generates multiple images that match description.

SRGAN - photorealistic single image super-resolution

Synthetic time series with recurrent conditional GANs (RCGANs, 2017)

21.2. GAN Training

In the training, we use `@tf.function` decorator to speed up execution by compiling it to a TensorFlow operation rather than relying on eager execution.

```
@tf.function
def train_step(images):
    # generate random input for generator
    noise = tf.random.normal([BATCH_SIZE, noise_dim])
    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        # get generator output
        generated_img = generator(noise, training=True)
        # collect discriminator decisions regarding real and fake input
        true_output = discriminator(images, training=True)
        fake_output = discriminator(generated_img, training=True)
        # compute loss for each model
        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(true_output, fake_output)
        # compute gradients for each loss w.r.t. model variables
    grad_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
    grad_discriminator = disc_tape.gradient(disc_loss, discriminator.trianable_variables)
    # apply gradient to complete backpropagation step
    gen_optimizer.apply_gradients(zip(grad_generator,
                                      generator.trainable_variables))
    dis_optimizer.apply_gradients(zip(grad_discriminator,
                                      discriminator.trainable_variables))
```

21.3. TimeGAN for synthetic financial data

Time-seres Generative Aversarial Network (TimeGAN, NeurIPS 2019, Yoon *et al.*): learn temporal dynamics that shape how one sequence of observatins

follow another. Learn temporal correlations by combining supervised learning and unsupervised training. Model learns time-series embedding space while optimizing both supervised and adversarial objectives, which encourage it to adhere to the dynamics observed while sampling from historical data during training. We can test the model on various time series, including historical stock prices, and find the quality of synthetic data significantly outperforms that of available alternatives.

We want to capture both cross-sectional distribution of features at each point in time and the longitudinal relationships among these features over time. Like image, the model not only needs to learn what a realistic image looks like, but also how one image evolves from the previous as in a video.

21.3.1. Combining Adversarial and supervised training

TimeGAN incorporates autoregressive nature of time series by combining unsupervised adversarial loss on both real and synthetic sequences familiar from DCGAN example, with a *stepwise supervised loss* w.r.t. original data. Goal is to reward the model for learning *distribution over transitions* from one point in time to the next that are present in historical data.

TimeGAN includes an embedding network that maps the time-series features to a lower-dimensinal latent space to reduce complexity of adversarial space. We can capture the drivers of temporal dynamics that often have lower dimensionality.

Both generator and embedding (autoencoder) network in TimeGAN are responsible for minimizing the supervised loss that measures how well the model learns the dynamics. So, the model learns a latent space conditioned on facilitating the generator's task to faithfully reproduce temporal relationships observed in historical data.

21.3.2. 4 components of TimeGAN architecture

- Autoencoder: embedding and recovery networks
- Adversarial network: sequence generator and sequence discriminator components

By **joint training** of autoencoder and adversarial networks with 3 different loss functions, the reconstruction loss optimizes the autoencoder, the unsupervised loss trains the adversarial net, and supervised loss enforces the temporal dynamics. So, TimeGAN simultaneously learns to encode features, generate representations, and iterate across time. The embedding network creates the latent space, adversarial network operates within the space, and supervised loss synchronizes the latent dynamics of both real and sythetic data.

Embedding and recovery components of autoencoder map the feature space into latent space. This facilitates learning of temporal dynamics by adversariall

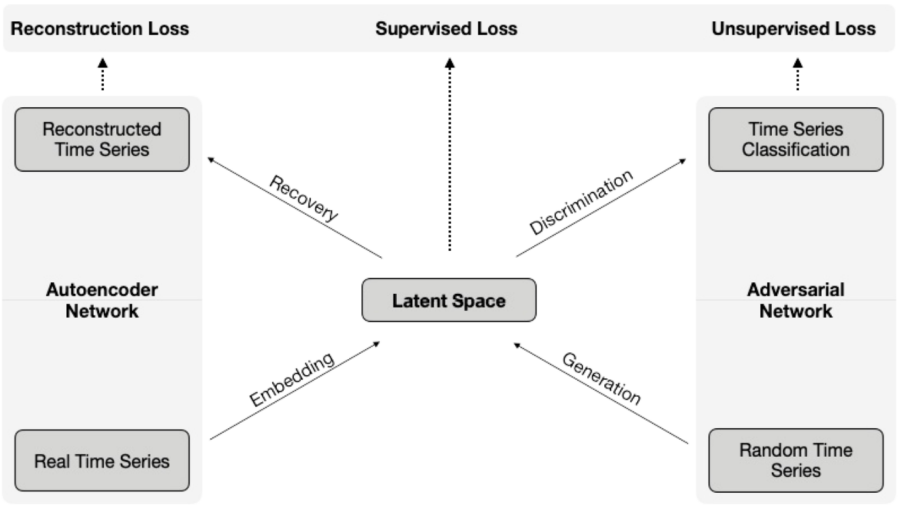

Figure 2: The components of TimeGAN architecture (Figure 21.4)

networok, which learns in lower-dimensional space. We can implement embedding and recovery network using stacked RNN and feedforward network.

Generator and Dicriminator differ from DCGAN because they operate on sequential data and synthetic features are generated in latent space and model learns simultaneously.

Unsupervised loss: competitive interaction between generator and discriminator in DCGAN, while generator can minimize prob that discriminator classifies its output as fake, discriminator can optimize correct classification or real and fake inputs.

Supervised loss: how well generator approximates the actual next time step in latent space when receiving encoded real data fr prior sequence.

21.4. TimeGAN Implementation using TensorFlow 2

We scale each series to $[0, 1]$ with Sklearn `MinMaxScaler` class

```
df = pd.read_hdf(hdf_store, 'data/real')
scaler = MinMaxScaler()
scaled_data = scaler.fit_transform(df).astype(np.float32)
data = []
for i in range(len(df) - seq_len):
    data.append(scaled_data[i:i+seq_len])
n_series = len(data)
```

```
real_series = (tf.data.Dataset
                .from_tensor_slices(data)
                .shuffle(buffer_size = n_windows)
                .batch(batch_size))
real_series_iter = iter(real_series.repeat())

# generator to draw requisite data uniform at random and feeds result to ‘tf.data.Dataset‘ t
def make_random_data():
    while True:
        yield np.random.uniform(low=0, high=1, size=(seq_len, n_seq))
random_series = iter(tf.data.Dataset
                     .from_generator(make_random_data,
                                     output_types = tf.float32)
                     .batch(batch_size)
                     .repeat())
# ... a lot of functions

# Autoencoder with real data: Training phase 1
autoencoder_optimizer = Adam()
@tf.function
def train_autoencoder_init(x):
    with tf.GradientTape() as tape:
        x_tilde = autoencoder(x)
        embedding_loss_t0 = mse(x,x_tilde)
        e_loss_0 = 10 * tf.sqrt(embedding_loss_t0)
    var_list = embedder.trainable_variables + recovery.trainable_variables
    gradients = tape.gradient(e_loss_0, var_list)
    autoencoder_optimizer.apply_gradients(zip(gradients, var_list))
    return tf.sqrt(embedding_loss_t0)

# Supervised learning with real data: Training phase 2
supervisor_optimizer = Adam()
@tf.funnnction
def train_supervisor(x):
    with tf.GradientTape() as tape:
        h = embedder(x)
        h_hat_supervised = supervisor(h)
        g_loss_s = mse(h[:,1:,:], h_hat_supervised[:,1:,:])
    var_list = supervisor.trainable_variables
    gradients = tape.gradient(g_loss_s, var_list)
    supervisor_optimizer.apply_gradients(zip(gradients, var_list))
    return g_loss_s

# Joint training with real random data: Training phase 3
def get_generator_moment_loss(y_true, y_pred):
    y_true_mean, y_true_var = tf.nn.moments(x=y_true, axes=[0])
```

```
    y_pred_mean, y_pred_var = tf.nn.moments(x=y_pred, axes=[0])
    g_loss_mean = tf.reduce_mean(tf.abs(y_true_mean - y_pred_mean))
    g_loss_var = tf.reduce_mean(tf.abs(tf.sqrt(y_true_var + 1e-6)
            - tf.sqrt(y_pred_var + 1e-6)))
    return g_loss_mean + g_loss_var

# ...
# Use 4 loss functions and combine networks components
@tf.function
def train_generator(x,z):
    with tf.GradientTape() as tape:
        y_fake = adversarial_supervised(z)
        generator_loss_unsupervised = bce(y_true = tf.ones_like(y_fake),
                    y_pred = y_fake)
        y_fake_e = adversarial_emb(z)
        generator_loss_unsupervised_e = bce(y_true = tf.ones_like(y_fake_e),
                    y_pred = y_fake_e)
        h = embedder(x)
        h_hat_supervise = supervisor(h)
        generator_loss_supervise = mse(h[:,1:,:],
                                       h_hat_supervised[:,1:,:])
        x_hat = synthetic_data(z)
        generator_moment_loss = get_generator_moment_loss(x, x_hat)
        generator_loss = (generator_loss_unsupervised +
                    generator_loss_unsupervise_e +
                    100 * tf.sqrt(generator_loss_supervised)
                    + 100 * generator_moment_loss)
        var_list = generator.trainable_variables + supervisor.trainable_variables
        gradients = tape.gradient(generator_loss, var_list)
        generator_optimizer.apply_gradients(zip(gradients, var_list))
        return (generator_loss_unsupervised, generator_loss_supervise, generator_moment_loss
```

21.5. Assessing diversity - visualization with PCA and t-SNE

To visualize real and synthetic series with 24 time steps and 6 features, we reduce their dimensionality so that we plot them in 2 dim.

PCA is linear method that idenfities a new basis with mutually orthogonal vectors that successively capture directions of max variance in adta.

```
pca = PCA(n_components = 2)
pca.fit(real_sample_2d)
pca_real = (pd.DataFrame(pca.transform(real_sample_2d))
            .assign(Data='Real'))
```

```
pca_synthetic = (pd.DataFrame(pca.transform(synthetic_sample_2d))
            .assign(Data = 'Synthetic'))
```

t-SNE is nonlinear manifold learning for visualization of high-dim data, which converts similarities between data points to joint prob and to minimize KL divergence between joint prob of low-dim embedding and high-dim data.

```
tsne_data = np.concatenate((real_sample_2d,
                            synthetic_sample_2d), axis=0)
tsne = TSNE(n_components=2, perplexity=40)
tsne_result = tsne.fit_transform(tsne_data)
```

21.6. Summary on TimeGAN

Caveats:

- We generateed price data for small number of assets at daily frequency, but in reality, we are interested in returns for much larger number of assets at high frequency. Cross-sectional and temporal dynamics are more complex and may require adjustments to TimeGAN architecture.
- We need to expand scope to high-dim time series containing info other than prices and also to test usefulness for feature engineering.

22. Deep Reinforcement Learning - Building Trading Agent

22.1. Q-leaning

Bellman equations define a recursive relationship between value functions for all states s in S and any of their successor states s under a policy π, by decompsing the value function into immediate reward and discounted value of next state.

Solve MDP/Q-learning using `pymdptoolbox` library.

Q-learning uses learning rate α * Temporal difference to update the value function, this improvement is bootstrapping. Q-learning is part of temporal difference (TD) learning algorithms. TD learning doesn't wait until receiving final reward for an episode. Instead, it updates its estimates using values of intermediate states that are closer to final reeward, so intermediate state is one time step ahead.

22.2. Double Deep Q-learning (DDQN) - decoupling action and prediction

Q-learning overestimates the action values, because it purposely samples max estimated action values. This bias can negatively affect learning process and resulting policy, if not applying uniformly and alters action preferences. So we decouple the estimation of action vallues from selection of actions by DDQN.

We can use Inverse Reinforcement Learning, which aims to identify the reward function of an agent (trader) given its observed behavior (Roa-Vicens *et al.* 2019) on limit order book context.

23. Conclusions

23.1. Big Data - from Hadoop to Spark

Tools of Hadoop:

- Apache Pig: a data processing language for large-scale extract-transform-load (ETL) pipeline using MapReduce
- Apache Hive: de facto standard for interactive SQL queries over petabytes of data
- Apache HBase: NoSQL database for real-time read/write access that scales linearly to billions of rows and millions of columns.

Apache Spark has accelerated computation at scale due to resilient distributed data (RDD) structure, which allows highly optimized in-memory computation. It includes iterative computation for optimization, like gradient descent. Spark DataFrame interface is designed in pandas.

23.2. ML Tools to facilitate ML workflow

- H2O.ai: integrates cloud computing with ML automation. It can fit thousandsd of potential models to their data to explore patterns in data.
- Datarobot: automatets model dev process by providing a platform to rapidlly build and deploy predictive models in the cloud or on-premises
- Dataiku: collaborative data science platform to prototype, build and deliver data products
- **BeakerX project**: 2-sigma's quant tool
- Bloomberg's integrated Jupyter Notebok

23.3. Moving Average & other indicators

MA, EMA, Weighted-MA, Double exponential MA (DEMA), Triple exponential MA (TEMA), Triangular MA (TRIMA)

Kaufman adaptive MA (KAMA): takes into account the market volatility changes.

MESA adaptive MA (MAMA): exponential MA that adapts t market price movement based on the rate change of phase, as measured by Hilbert Transform discriminator (`TA_Lib` documentation). In addition t price series, MAMA accepts 2 additional params: *fastlimit* and *slowlimit* to control max and min alpha values that should be applied to EMA when calculating MAMA.

Parabolic SAR identifies trend reversals. It's a trend-following (lagging) indicator to trail stop loss or determine entry/exit points. This indicator is less reliable in flat/range-bound market. See `talib.SAR`.

Arooon Oscillator measures the time between highs and lows over a time period.

Stochastic Relative strength index (StochRSI) compares distance of current RSI to lowest RSI over a time period to max range f values the RSI has assumed for this period. `talib.STOCHRSI`

Stochastic oscillator is a momentum indicator that compares a particular closing price to a range of its prices over a period of time. `talib.STOCH`

Ultimate oscillator (ULTOSC) measures average difference between current close and previous lowest price over 3 timeframes - 7, 14, 28 days to avoid overreacting to short-term price changes and incorporate short/medium/long-term market trends.

Williams %R. On-balance volume. Normalize average true range (NATR)

Chaikin advance/decline (AD) = accumulation/distribution (AD) line: volume based indicator to measure cumulative flow of money into and out of an asset. It can signal a change in direction when the indicator diverges from security price. Chaikin A/D oscillator (ADOSC) is MACD indicator applied to Chaikin ADD line.

Rank correlation of performance metrics (Figure A.18)

𝔼nd - 𝕆f - 𝕋he - 𝔹ook - ℕotes