Machine Learning Engineer Nanodegree

Capstone Proposal

Carl Martin
October 22nd, 2016

Domain Background

Predicting stock markets is hard. The efficient market hypothesis states that stock prices fully reflect all available information and prices are only influenced by new information. Since news happens randomly, stock prices follow a random walk pattern. This hypothesis has been extensively studied in the past. Researchers agree, that the behavior of stock prices is approximately close to the random walk process (Qian & Rasheed, 2006).

There are two major schools of thought of stock market analysis: fundamental and technical analysis. Fundamental analysis looks to determine a company's intrinsic value, while technical analysis tries to identify market trends. There are traders from both schools of thought that are very successful, proving that traders can exploit small market inefficiencies to increase portfolio value.

Since stock returns follow approximately a random walk model, it makes sense to treat stock market prediction as a classification problem. The goal is to classify whether a given stock will go up or down. Many studies have applied machine learning to this problem and achieved accuracy up to 65% for individual stocks using multiple classifiers (Qian & Rasheed, 2006) and up to 76% for indicies exploiting temporal correlations (Shen, Jian & Zhang, 2012).

Problem Statement and Evaluation Metrics

Stock market prediction in this capstone project is treated as a classification problem, where the objective is to predict whether the cumulative returns of a stock over a period of one day will be greater than zero (up) or smaller (down). The final algorithm has to return (1) the predicted label and (2) a confidence level.

The final classification algorithm will be used in a long/short equity trading strategy. Using a ranking scheme based on both the predicted label and the algorithms confidence the strategy goes long (buys) the top x% of equities of the ranking and goes short (sells) on the bottom x% of the ranking, while maintaining equal dollar volume between the long and short position (Granizo-Mackenzie). The implementation of the machine learning algorithm is out of the scope of this capstone project, yet relevant for the evaluation metric.

Given the algorithms future use, the evaluation metric has to penalize the algorithm heavily for false positives and false negatives, for which the algorithm assigned a high confidence level. Given the

requirements, the classification algorithm's performance will be evaluated using *logarithmic loss*, which can be calculated using the following formula:

$$-\frac{1}{N}\sum_{i-1}^{N}\sum_{j-1}^{M} y_{ji} \cdot log(p_{ji})$$

where N is the number of samples, M the number of possible labels, y_{ji} indicates whether label j is the correct classification for instance i, and p_{ji} is the probabilty yielded from the algorithm for label j. The objective will be to achieve a log_loss value as close to zero as possible.

Datasets and Inputs

Training and Testing Dates
Data will be split sequentially (not randomly) — the first 80% of the timeseries will be used for training the algorithm, which will be evaluated on the last 20% of the available data. For stocks that existed before 01–01–2003 all avaible from then until now will be used.

Stock Universe
Since the machine learning algorithm will be used for trading eventually, the stocks the algorithm is trained on have to be selected carefully. I'll draw from the TradableUS Methodology (Wassermann, 2016) to select a set of 500 stocks that are tradable, liquid and financially viable at the beginning of the training time to avoid survivor bias.

Pricing Data
Using the Yahoo finance API I'll be loading adjusted close, volume, high and low as needed. The API connection code is provided to reviewer. Data will be loaded automatically as needed.

Fundamental Data
I'll be using fundamental data and index prices from Intrinio, which is provided to the reviewer with in SQLite database.

- Market Capitalization
- Earnings before Interest and Taxes
- Earnings before Interest, Taxes, Depreciation and Amortization
- Sector Information
- S&P500, DAX, DASDAQ, DIJA, FTSE HENGSHENG, ASX and NIKKEI historical pricing

Should the algorithm not perform sufficiently I might consider additional free foreign exchange rate data from Quandl, which will be provided to the reviewer in a SQLite database as well.

Solution Statement

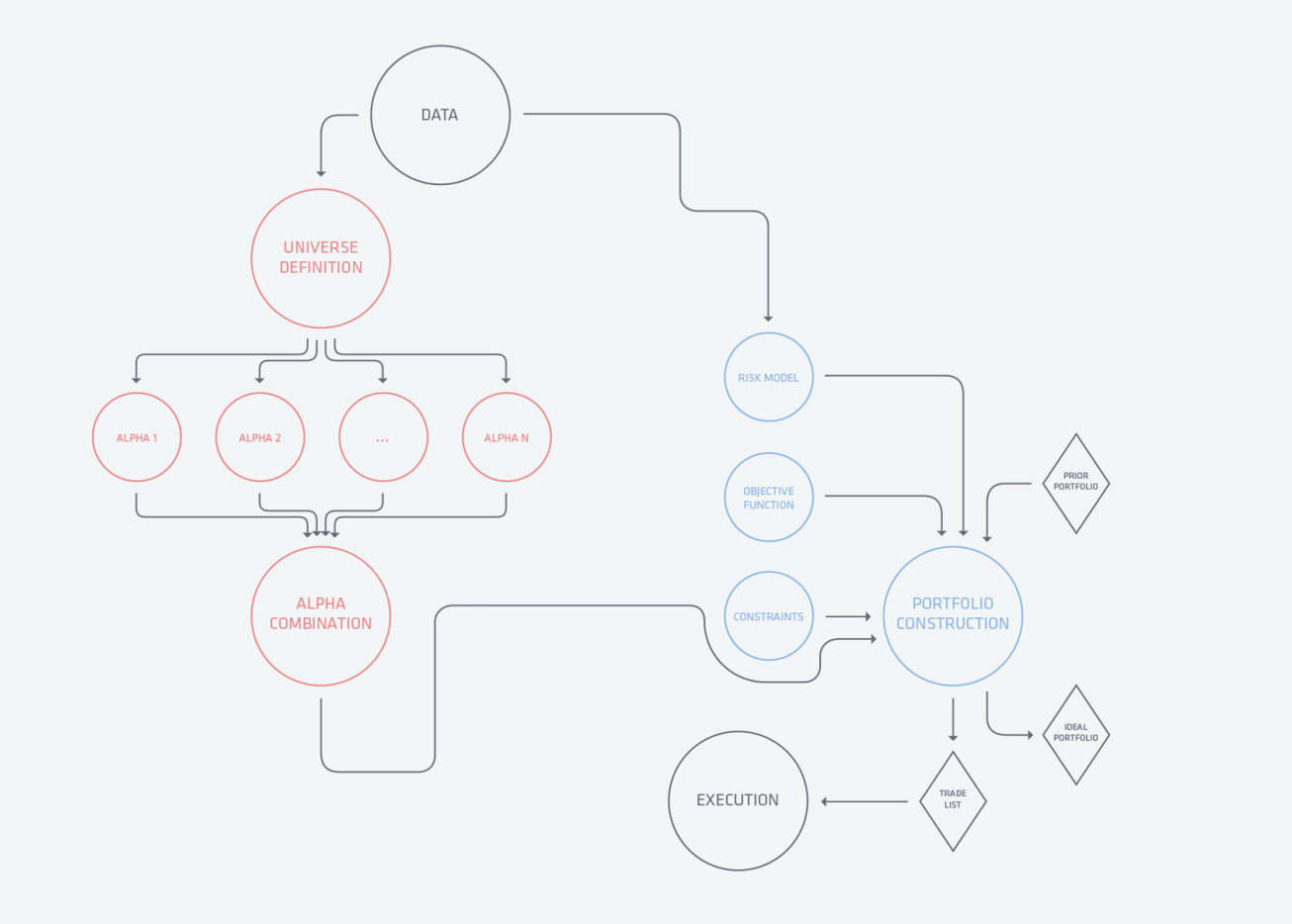
The final algorithms training interface will take a data range `(start_date, end_date)` and will train the algorithm on the preselected stock universe. The query interface will take `date` as an input and return a `DataFrame` with the stocks of the stock universe ranked by class (up/down) and algorithm confidence. The query date passed must be after the training `end_date` . Both, the training and querying interface, will be provided in an Jupyter Notebook as two respective Python 2.7 functions.

Benchmark Model

The benchmark model takes the scaled daily returns of the past four days for a given stock and uses the AdaBoost Classifier. The benchmark model achieved a logarithmic loss on average of 0.69. The specific implementation of the benchmark model can be found in the appendix (last section).

Project Design

My Capstone project will follow the Professional Quant Equity Workflow proposed by Jonathan Larkin, Chief Investment Officer at Quantopian. Having selected the data sources and stock universe above, I've already completed the first two steps of the workflow.



Having selected the trading 'universe', I'll move on to the feature engineering, or alpha discovery, stage of the workflow. I'll be calculating fundamental and technical indicators (alphas) with the data from the aforementioned datasets for the selected stock universe. Each alpha will be evaluated using the `alphalens` package to determine whether the alpha is predictive of future returns.

I intend to develop a system that is an ensemble of multiple machine learning classifiers. The goal of ensembles is that the performance of the entire system is better than the performance of each classifier individually. Another benefit of ensemble systems is that even if one underlying classifier fails, the system can continue to function, which makes it more robust (Qian & Rasheed, 2006).

I'll use the following algorithms as my base-learners (subject to change):

- `sklearn.svm.SVC` — Support Vector Classifier
- `sklearn.neighbors.KNeighborsClassifier` — K-Nearest Neighbors Classifier

- `sklearn.neural_network.MLPClassifier` — Neural Network Classifier
- `sklearn.ensemble.RandomForest` — Random Forest Classifier
- `sklearn.ensemble.AdaBoost` — AdaBoost Classifier

As the meta-learner, which combines the predictions of the individual classifiers, I'll be using the AdaBoost Classifier.

The classification system will be implemented using sklearn's `Pipeline` and `FeatureUnion` objects. The final Pipeline will be implemented the following way:

```
from sklearn.pipeline import Pipeline, FeatureUnion
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn import preprocessing

# Transforms sklearn models to be used as features within pipelines
class ModelTransformer(BaseEstimator, TransformerMixin):

    def __init__(self, model):
        self.model = model

    def fit(self, *args, **kwargs):
        self.model.fit(*args, **kwargs)
        return self

    def transform(self, X, **transform_params):
        return pd.DataFrame(self.model.predict(X))

pipeline = Pipeline([
  ('features', FeatureUnion([
    ('feature_1', Pipeline([
      ('calculation', someCustomTransformer()),
      ('scaling', preprocessing.RobustScaler())
    ])),

    ...

  ])),
  ('estimators', FeatureUnion([
    ('estimator_1', ModelTransformer(someClassifier())),
```

```
    ...

  ])),
  ('estimator', AdaBoost())
])
```

The advantage of using sklearn Pipelines is that running GridSearch and Crossvalidation on the entire pipeline is feasible and the final model will be easier to understand.

References

Qian, Bo, and Khaled Rasheed. "Stock Market Prediction with Multiple Classifiers." *Applied Intelligence* 26.1 (2006): 25-33. Web.

Shunrong Shen, Haomiao Jiang, and Tongda Zhang., “Stock market forecasting using machine learning algorithms”. *Stanford University*, 2012

Granizo-Mackenzie, Delaney. "Quantopian Lecture Series: Long-Short Equity Strategies." N.p., n.d. Web. 22 Oct. 2016.

Wassermann, Gil. "Quantopian - The Tradeable500US Is (almost) Here!" N.p., 6 July 2016. Web. 22 Oct. 2016.

Appendix

I provided the implementation of the Benchmark model below:

```
import pandas as pd
import numpy as np
from datetime import datetime
import requests
import json
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.pipeline import Pipeline, FeatureUnion
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
from sklearn import preprocessing
from sklearn.ensemble import AdaBoostClassifier
from sklearn.metrics import log_loss

#############
Necessary Classes
```

```
#############

class Database(object):

    def __init__(self):

        self.engine = create_engine('sqlite:///datalake.db')

    def save(self, tablename, data, behavior='fail', index=False):

        '''behavior options:
        fail: If table exists, do nothing.
        replace: If table exists, drop it, recreate it, and insert data.
        append: If table exists, insert data. Create if does not exist.'''

        data.to_sql(tablename, self.engine, if_exists=behavior, index=index)

    def search_ticker(self, query):
        return pd.read_sql("SELECT security_name, ticker FROM 'Master Data' WHERE security_name LIKE '%{}%' OR ticker LIKE '%{}%'".format(query, query), self.engine)

    def retrieve_prices(self, ticker):
        ticker = ticker.upper()
        prices = pd.read_sql("SELECT * from '{}'".format(ticker), self.engine)
        prices = prices.set_index('date')
        return prices

# Main Dataprovider class
class Intrinio(object):

    def __init__(self):

        self.user = 'b8f16afb02891a7fc1085118fd64ad47'
        self.password = '5ad4db89b891c478803e731c8b3ff2d0'

    def request(self, url, payload={}):
```

```
50
51          r = requests.get(url, params=payload, auth=(self.user, self.passwor
    d))
52
53          if r.status_code == 200:
54
55              return json.loads(r.text)
56
57          else:
58              raise RuntimeError('An error has occurred while requesting data.
    Status Code:{}'.format(r.status_code))
59
60      def extract_data(self, response):
61
62          return pd.DataFrame(response['data'])
63
64      def multiple_pages(self, url, parameters={}):
65          response = self.request(url, parameters)
66          data = self.extract_data(response)
67
68          if response['total_pages'] > 1:
69              i = 2
70
71              while i <= response['total_pages']:
72
73                  parameters['page_number'] = i
74                  data = data.append(self.request(url, parameters)['data'], ig
    nore_index=True)
75                  i += 1
76
77          return data
78
79      def request_new_prices(self, ticker):
80
81          response = self.request('https://api.intrinio.com/prices', {'identif
    ier': ticker, 'sort_order':'asc'})
82          return self.extract_data(response).set_index('date')
83
```

```
# This class wraps repeatedly used functionality of assets into a Stock object
class Stock(object):

    def __init__(self, ticker, data_provider=intrinio, database=database, validate=True):

        self.data_provider = data_provider
        self.database = database
        self.ticker = ticker.upper()

        if validate == True:
            self.company_name = self.retrieve_company_name(ticker)

        try:
            # TODO: Check for current date
            self.prices()
        except:
            self.download_prices()

    def retrieve_company_name(self, ticker):

        try:
            company_name = pd.read_sql("SELECT security_name FROM 'Master Data' WHERE ticker = '{}'".format(ticker), self.database.engine).ix[0][0]
            return company_name

        except:
            raise ValueError('The ticker {} you entered is not valid.'.format(ticker))

    def compute_daily_ret(self, data):
        data = data['adj_close']
        daily_ret = data.copy()
        daily_ret[1:] = (data[1:] / data[:-1].values) - 1
        daily_ret.ix[0] = 0
        return daily_ret

```

```
118     def download_prices(self):
119 
120         prices = self.data_provider.request_new_prices(self.ticker)
121 
122         if len(prices) < 1:
123             raise RuntimeError('An error occurred, while downloading the pri
    ces.')
124 
125         prices['daily_ret'] = self.compute_daily_ret(prices)
126         self.database.save(self.ticker, prices, 'replace', 'date')
127 
128     def prices(self):
129         prices = self.database.retrieve_prices(self.ticker)
130         return prices
131 
132 class Windowing(BaseEstimator, TransformerMixin):
133     '''
134     http://stockcharts.com/school/doku.php?st=roc&id=chart_school:technical_
    indicators:rate_of_change_roc_and_momentum
135     '''
136     def __init__(self, t=5, feature='daily_ret'):
137 
138         self.t = t
139         self.feature = feature
140 
141     def fit(self, X, y=None, **fit_params):
142         return self
143 
144     def transform(self, X):
145         X = X.copy()
146         columns = ['{}_{}'.format(self.feature, 0)]
147         X.loc[:,'{}_{}'.format(self.feature, 0)] = X[:][self.feature].values
148         for i in np.arange(1, self.t+1):
149             X.loc[i:,'{}_{}'.format(self.feature, i)] = X[:-i][self.featur
    e].values
150             X.loc[:i,'{}_{}'.format(self.feature, i)] = 0
151             columns.append('{}_{}'.format(self.feature, i))
152         return X[columns]
```

```

def train_test_split(X, y, test_size=0.2, target='daily_ret'):
        index = int(len(X) * (1 - test_size))
        return X[:index], X[index:], y[:index][target].values, y[index:][tar
get].values
```

```
#############
Algorithm
#############

intrinio = Intrinio()
# Set validate to false, because I didn't provide the Master Data class
stock = Stock('AAPL', validate=False)

X = stock.prices()[['adj_low', 'adj_high', 'adj_close','adj_volume', 'daily_
ret']]

# Time delay between X and y - Experimenting with time delay lead to improve
d scores. Likely becuase it reduces noise. TODO: Investigate
time_delay = 1

# Calculate the price difference between current price and time_delay days p
rior
target_transformation = Momentum(time_delay) #
y = target_transformation.transform(X)
y.columns = ['target']
y['target'] = y['target'].apply(lambda x: 1 if x > 0 else -1)

# Training, testing split
X_train, X_test, y_train, y_test = train_test_split(X[:-time_delay], y[time_
delay:], target='target')

# Pipeline for feature engineering
pipeline = Pipeline([
        ('features', FeatureUnion([
            ('daily_ret', Pipeline([
                ('windowing', Windowing(feature='daily_ret')),
                ('scale', preprocessing.RobustScaler())
```

```
            ]))
        ], n_jobs=1)),
        ('estimator', AdaBoostClassifier(random_state=0))
    ])

params = {
    'estimator__n_estimators': np.arange(100,200, 10),
    'features__daily_ret__windowing__t': np.arange(4,10)
}

# Basic hyperparameter optimization of algorithm and features
grid_search = GridSearchCV(pipeline, param_grid=params, n_jobs=12)
grid_search.fit(X_train, y_train)
classifier = grid_search.best_estimator_

# Evaluation of classifier with log_loss metric
y_proba = classifier.predict_proba(X_test)
log_loss(y_test, y_proba)
```