ML_approaches _to_crypto

May 27, 2021

0.1 Random forests, SVMs and Linear Models in the cryptocurrency space

0.1.1 Introduction

This analysis looks at the predictability of 3 major cryptocurrencies: Bitcoin, Ethereum and Litecoin through standard ML algorithms.

This is part of a broader project available here: https://github.com/IanLDias/algo-trading

Data Sources: Two main data sources were used in this analysis, one to collect data on historical prices and the other to collect relevant IVs. - Coinmetrics: 10 IV's regarding blockchain data - Transaction information (count, size, fees, difficulty) - Market cap - Active addresses

https://coinmetrics.io/

- Cryptocompare
  - Historical prices (OHLCV)

https://www.cryptocompare.com/

0.1.2 Data preprocessing

- Historical price data is saved on a local postgresql server
- Data from coinmetrics is saved onto a local csv file
- All IVs and price data are collected and separated out into their individual dataframes.

[1]:
```
import pandas as pd
import re
import numpy as np
import plotly.express as px
#Gets data from postgressql server
from helper_funcs import get_data, convert_unix_to_datetime, separate_symbols

from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier, RandomForestRegressor
from sklearn.linear_model import LinearRegression, LogisticRegression
from sklearn.svm import SVC, SVR

from sklearn.metrics import accuracy_score, precision_score, f1_score,␣
 ↪recall_score
```

```python
from sklearn.metrics import mean_squared_error as MSE, mean_absolute_error as␣
↪MAE
```

[2]:
```python
tickers = ['BTC', 'ETH', 'LTC']
def get_tickers(tickers):
    """
    Given a list of tickers, return separate dataframe
    """
    data = get_data(tickers)
    df = pd.DataFrame(data, columns = ['id', 'symbol', 'date', 'high', 'low',␣
↪'open', 'close', 'volumeto', 'volumefor'])
    df = df.drop('id', axis=1)
    df['date'] = convert_unix_to_datetime(df['date'])
    return df

df = get_tickers(tickers)
#Separate out the coins into individual dataframes
btc, eth, ltc = separate_symbols(df)

#Indepedent variables. Has blockchain data for BTC/ETH/LTC
coinmetric_df = pd.read_csv('coin_metrics_btc_data.csv', encoding='utf-16')
```

```
Finding data for: 'BTC', 'ETH', 'LTC'
```

[3]:
```python
def _get_coin_cols(coin):
    """
    Used in preprocess function. Returns relevant columns for a given coin
    """
    cols = []
    for col in coinmetric_df.columns:
        if re.match(coin, col):
            cols.append(col)
    time_df = pd.DataFrame(coinmetric_df['Time'])
    time_df.rename(columns={"Time": "date"}, inplace=True)
    return time_df.join(coinmetric_df[cols])

def _take_diff(column_list, df):
    """
    Used in preprocess function. Returns the difference for a given list of␣
↪columns and dataframe.
    """
    for col in column_list:
        df[col] = df[col].diff()
    return df

def preprocess(df, symbol):
    """
```

```
    Given a dataframe and a symbol (i.e. 'BTC'), returns a clean dataset with␣
↪the relevant columns.
    Computes relative price change, parkinson volatility and adds 7 lags to the␣
↪closing price and volatility.

    Returns a dataframe
    """
    # Natural log of closing price is taken
    df['close'] = np.log(df['close'])

    # Need to use current data to predict 1 step ahead
    # df['close'] is shifted one step back to achieve this

    df.loc['close'] = df['close'].shift(-1)
    df['rel_price_change'] = 2 * (df['high'] - df['low']) / (df['high'] +␣
↪df['low'])
    df['parkinson_vol'] = np.sqrt((np.log(df['high']/df['low'])**2)/4*np.log(2))
    df = df[['date', 'close', 'volumeto', 'volumefor', 'rel_price_change',␣
↪'parkinson_vol']]

    for i in range(8):
        lag_close_col = df['close'].shift(i)
        lag_park_col = df['parkinson_vol'].shift(i)
        df['close_lag'+str(i)] = lag_close_col
        df['parkinson_lag'+str(i)] = lag_park_col

    df = df.merge(_get_coin_cols(symbol), on='date')
    df = df.set_index('date')
    df.columns =[re.sub(symbol+' / ', '', col) for col in df.columns]

    column_list = ['Market Cap (USD)', 'Tx Cnt', 'Active Addr Cnt',
                   'Mean Difficulty', 'Block Cnt', 'Xfer Cnt']
    btc_df = _take_diff(column_list, df=df)
    return df
```

- First 7 lags of the closing price and parkinson's volatility. Found through ACF and PACFs
- First diff of market cap, # transactions, active address, average difficulty, number of blocks, block size, number of payments

[4]:
```
#pd.options.mode.chained_assignment = None
btc = preprocess(btc, 'BTC')
eth = preprocess(eth, 'ETH')
ltc = preprocess(ltc, 'LTC')
```

[21]:
```
def split_data(df, return_test = False):
    if isinstance(df, tuple):
        df = df[0]
```

```
    df = df.dropna()
    train = df[:int(len(df) * 0.75)]
    test = df[int(len(df) * 0.75):]
    if return_test:
        return test

    y = train['close']
    X = train.drop('close', axis=1)

    X_train, X_valid, y_train, y_valid = train_test_split(X, y, shuffle=False)
    return X_train, X_valid, y_train, y_valid
```

0.1.3 Model building - Initial Data Split

- First 50% used to train the model. Training sample
- Next 25% each close is forecasted. Used to choose variables/hyperparameters. Validation sampl
- Use the models that showed the best performance in the validation sample. Test sample

[6]:
```
def plot_graph_sets(column, title=None):
    """
    Returns a graph for a pandas Series with date as the index.
    Colours in the training and validation sets used
    """
    p50 = int(len(column) * 0.5)
    p75 = int(len(column) * 0.75)

    fig = px.scatter(data_frame=column, range_color=(0,1000), title=title)

    fig.add_vrect(x0 = column.index[0], x1=column.index[p50:p50+1][0],␣
 ↪annotation_text="Training_set",
                  annotation_position="top right", fillcolor="blue", opacity=0.
 ↪25, line_width=0)

    fig.add_vrect(x0 = column.index[p50:p50+1][0], x1=column.index[p75:
 ↪p75+1][0], annotation_text="Validation_set",
                  annotation_position="top right", fillcolor="green", opacity=0.
 ↪25, line_width=0)

    fig.add_vrect(x0 = column.index[p75:p75+1][0], x1=column.index[-1],␣
 ↪annotation_text="Test_set",
                  annotation_position="top right", fillcolor="orange", opacity=0.
 ↪25, line_width=0)
    return fig
```

[7]:
```
plot_graph_sets(btc['close'], title='Bitcoin')
```

[8]:
```
class model_pipeline:
    """
    Full model pipeline.
    Requires cleaned dataframe from preprocess.
    """
    def __init__(self, df, verbose=False):
        self.df = df,
        self.verbose=verbose,
        self.models = ['LogisticRegression', 'RandomForestClassifier', 'SVC',␣
↪'EnsembleClf',
            'LinearRegression', 'RandomForestRegressor', 'SVR', 'EnsembleReg'],
        self.error = ['MAE', 'MSE', 'f1', 'precision', 'recall', 'accuracy'],
        self.model_list_clf = ['RandomForestClassifier', 'SVC',␣
↪'LogisticRegression'],
        self.model_list_reg = ['LinearRegression', 'RandomForestRegressor',␣
↪'SVR']

    def _class_or_reg(self, mod_type, sk_model, X_train, y_train, X_valid,␣
↪y_valid):
        """
        Helper function for fit_model
        """
        sk_model.fit(X_train, y_train)
        y_pred = sk_model.predict(X_valid)
        if mod_type == 'regression':
            errors = MAE(y_valid, y_pred), MSE(y_valid, y_pred)
            return sk_model, y_pred, errors

        elif mod_type == 'classification':
            errors = [f1_score(y_valid, y_pred), precision_score(y_valid,␣
↪y_pred),
                      recall_score(y_valid, y_pred), accuracy_score(y_valid,␣
↪y_pred)]
            return sk_model, y_pred, errors

    def fit_model(self, model_type):
        """

        Parameters
        ----------

        df : dataframe
            Full cleaned dataframe for a given coin

        model_type: str
            'RandomForestClassifier',
            'RandomForestRegressor',
```

```
            'SVC',
            'SVR',
            'LinearRegression',
            'LogisticRegression'

        Returns
        --------
        trained_model : sklearn model
            The model trained on the training set and validated on validation␣
↪set. Unseen to test set

        y_prediction: np.array
            A 1-D array that the model has predicted on the validation set.

        errors : list
            if regression model:
                returns [MAE, MSE]
            if classification mode:
                returns [f1, precision, recall, accuracy]

        """
        X_train, X_valid, y_train, y_valid = split_data(self.df)

        #Convert to binary dependent variables for classification models
        clf_y_train = y_train.diff() > 0
        clf_y_valid = y_valid.diff() > 0

        if model_type == 'RandomForestClassifier':
            clf = RandomForestClassifier()
            return self._class_or_reg(mod_type = 'classification',␣
↪sk_model=clf, X_train=X_train, y_train = clf_y_train,
                        X_valid=X_valid, y_valid=clf_y_valid)

        elif model_type == 'RandomForestRegressor':
            reg = RandomForestRegressor()
            return self._class_or_reg(mod_type = 'regression', sk_model=reg,␣
↪X_train=X_train, y_train=y_train,
                               X_valid=X_valid, y_valid=y_valid)

        elif model_type == 'SVC':
            clf = SVC()
            return self._class_or_reg(mod_type = 'classification',␣
↪sk_model=clf, X_train=X_train, y_train = clf_y_train,
                        X_valid=X_valid, y_valid=clf_y_valid)

        elif model_type == 'SVR':
            reg = SVR()
```

```
            return self._class_or_reg(mod_type = 'regression', sk_model=reg,␣
↪X_train=X_train, y_train=y_train,
                              X_valid=X_valid, y_valid=y_valid)


        elif model_type == 'LinearRegression':
            reg = LinearRegression()
            return self._class_or_reg(mod_type = 'regression', sk_model=reg,␣
↪X_train=X_train, y_train=y_train,
                              X_valid=X_valid, y_valid=y_valid)

        elif model_type == 'LogisticRegression':
            clf = LogisticRegression()
            return self._class_or_reg(mod_type = 'classification',␣
↪sk_model=clf, X_train=X_train, y_train = clf_y_train,
                       X_valid=X_valid, y_valid=clf_y_valid)

    def make_dataframe(self, ensemble=True):
        """
        Summarizes the errors of all used models.
        Ensemble adds a combination model for both classification and regression
        Returns a dataframe with all relevant errors in self.error
        """
        df_summary = pd.DataFrame(index = self.models[0], columns = self.
↪error[0])
        classify_models = []
        self.model_list_clf = self.model_list_clf[0]
        for i in self.model_list_clf:
            clf, clf_pred, [f1, precision, recall, accuracy] = self.fit_model(i)
            classify_models.append((f1, precision, recall, accuracy))

        for i, vals in zip(self.model_list_clf, classify_models):
            df_summary.loc[i][2:] =  vals

        reg_models = []
        for i in self.model_list_reg:
            reg, reg_pred, [mae, mse] = self.fit_model(i)
            reg_models.append([mae, mse])

        for i, vals in zip(self.model_list_reg, reg_models):
            df_summary.loc[i][:2] = vals

        if ensemble:
            clf_errors, reg_errors = self.ensemble()
            df_summary.loc['EnsembleClf'][2:] = clf_errors
            df_summary.loc['EnsembleReg'][:2] = reg_errors
```

```
        return df_summary

    def ensemble(self):
        """
        Combines all classification methods listed in self.model_list_clf and
        regression methods in self.model_list_reg and returns the average␣
↪result.
        """
        ensemble_clf = []
        if isinstance(self.model_list_clf, tuple):
            self.model_list_clf = self.model_list_clf[0]
        for i in self.model_list_clf:
            clf, clf_pred, [f1, precision, recall, accuracy] = self.fit_model(i)
            ensemble_clf.append(clf_pred)
        mapping = {True: 1, False: -1}
        mapped_data = []

        for i in ensemble_clf:
            mapped_data.append([mapping[x] for x in i])
        model_1, model_2, model_3 = np.array(mapped_data)
        y_pred = model_1+model_2+model_3
        _, _, _, y_valid = split_data(self.df)
        y_valid_clf = y_valid.diff() > 0
        y_pred = y_pred > 0
        clf_errors = [f1_score(y_valid_clf, y_pred),␣
↪precision_score(y_valid_clf, y_pred),
                      recall_score(y_valid_clf, y_pred),␣
↪accuracy_score(y_valid_clf, y_pred)]

        ensemble_reg = []
        for i in self.model_list_reg:
            reg, reg_pred, [mae, mse] = self.fit_model(i)
            ensemble_reg.append(reg_pred)
        model_1, model_2, model_3 = ensemble_reg
        ensemble_reg = (np.array(model_1) + np.array(model_2) + np.
↪array(model_3))/3
        reg_errors = [MAE(y_valid, y_pred), MSE(y_valid, y_pred)]

        return clf_errors, reg_errors
```

[9]:
```
#---- Bitcoin ----
test_1 = model_pipeline(btc)
summary_df = test_1.make_dataframe()
summary_df
```

[9]:

```
                              MAE        MSE        f1  precision    recall  \
LogisticRegression            NaN        NaN  0.945455   0.968085  0.923858
RandomForestClassifier        NaN        NaN  0.948454   0.963351   0.93401
SVC                           NaN        NaN  0.806653   0.683099  0.984772
EnsembleClf                   NaN        NaN   0.94359   0.953368   0.93401
LinearRegression         0.000017        0.0       NaN        NaN       NaN
RandomForestRegressor    0.005262   0.000117       NaN        NaN       NaN
SVR                      1.168894   1.601073       NaN        NaN       NaN
EnsembleReg               8.32283  69.674009       NaN        NaN       NaN

                         accuracy
LogisticRegression        0.94385
RandomForestClassifier   0.946524
SVC                      0.751337
EnsembleClf              0.941176
LinearRegression              NaN
RandomForestRegressor         NaN
SVR                           NaN
EnsembleReg                   NaN
```

[10]:

```
#---- Ethereum----
test_2 = model_pipeline(eth)
summary_eth = test_2.make_dataframe()
summary_eth
```

[10]:

```
                              MAE        MSE        f1  precision    recall  \
LogisticRegression            NaN        NaN  0.955556   0.955556  0.955556
RandomForestClassifier        NaN        NaN  0.952646   0.955307      0.95
SVC                           NaN        NaN  0.097087   0.384615  0.055556
EnsembleClf                   NaN        NaN  0.952646   0.955307      0.95
LinearRegression         0.000727   0.000001       NaN        NaN       NaN
RandomForestRegressor    0.010088   0.000213       NaN        NaN       NaN
SVR                      1.373301   2.470815       NaN        NaN       NaN
EnsembleReg              4.679408  22.187443       NaN        NaN       NaN

                         accuracy
LogisticRegression       0.957219
RandomForestClassifier   0.954545
SVC                      0.502674
EnsembleClf              0.954545
LinearRegression              NaN
RandomForestRegressor         NaN
SVR                           NaN
EnsembleReg                   NaN
```

[11]:

```
#----Litecoin----
test_3 = model_pipeline(ltc)
```

```
summary_ltc = test_3.make_dataframe()
summary_ltc
```

[11]:

```
                             MAE        MSE        f1  precision    recall  \
LogisticRegression           NaN        NaN  0.957219   0.937173  0.978142
RandomForestClassifier       NaN        NaN  0.970027   0.967391  0.972678
SVC                          NaN        NaN  0.914729   0.867647  0.967213
EnsembleClf                  NaN        NaN  0.965147   0.947368  0.983607
LinearRegression             0.0        0.0       NaN        NaN       NaN
RandomForestRegressor   0.004425   0.000049       NaN        NaN       NaN
SVR                     1.005269   1.490075       NaN        NaN       NaN
EnsembleReg             3.649711   13.72644       NaN        NaN       NaN

                        accuracy
LogisticRegression      0.957219
RandomForestClassifier  0.970588
SVC                     0.911765
EnsembleClf             0.965241
LinearRegression             NaN
RandomForestRegressor        NaN
SVR                          NaN
EnsembleReg                  NaN
```

0.1.4 Results

Linear Regression is seemingly the best model, with the value close to 0 (pandas rounds the the above table)

Using the Test data, which was separated at the very start. Unseen to any models

[29]:

```
test = split_data(btc, return_test=True)

y_test = test['close']
X_test = test.drop('close', axis=1)

lin_reg = model_pipeline(btc)
model_linreg, y_pred_lr, errors = lin_reg.fit_model('LinearRegression')
y_test_lr = model_linreg.predict(X_test)
```

[31]:

```
MAE(y_test, y_test_lr)
```

[31]: 2.857591268764808e-05

[32]:

```
MSE(y_test, y_test_lr)
```

[32]: 1.6234436221290687e-09

0.1.5 The linear regression model is still the best.

0.1.6 Cross-validation

[12]:
```
def params(model_cv):
    if model_cv == 'RandomForestClassifier' or model_cv ==␣
↪'RandomForestRegressor':
        n_estimators = [500, 1000, 1500]
        min_samples_leaf = [1, 2, 5]
        max_features = ['auto', None]
        min_impurity_decrease = [0, 0.1, 0.3]
        param_grid = dict(n_estimators=n_estimators, min_samples_leaf =␣
↪min_samples_leaf, max_features=max_features,
                              min_impurity_decrease = min_impurity_decrease)
        return param_grid
    if model_cv == 'SVC':
        C = [0.8, 1, 1.5, 5]
        kernel = ['rbf', 'poly']
        degree = [3, 5]
        class_weight = [None, 'balanced']
        param_grid = dict(C=C, kernel=kernel, degree=degree,␣
↪class_weight=class_weight)
        return param_grid

    if model_cv == 'SVR':
        C = [0.8, 1, 1.5, 5]
        kernel = ['rbf', 'poly']
        degree = [3, 5]
        epsilon = [0, 0.1, 0.2]
        param_grid = dict(C=C, kernel=kernel, degree = degree, epsilon =␣
↪epsilon)
        return param_grid

    if model_cv == 'LogisticRegression':
        C = [0.8, 1, 1.5]
        class_weight = ['None', 'balanced']
        param_grid = dict(C=C, class_weight=class_weight)
        return param_grid
```

[13]:
```
X_train, X_valid, y_train, y_valid = split_data(btc)
clf_y_train = y_train.diff() > 0
clf_y_valid = y_valid.diff() > 0

def cross_validate(model, name_of_model, classify=False):
    param_grid = params(name_of_model)
    grid_search = GridSearchCV(estimator=model, param_grid=param_grid, cv=3,␣
↪n_jobs=-1, verbose=1)
    if classify:
```

```
        grid_search.fit(X_train, clf_y_train)
    else:
        grid_search.fit(X_train, y_train)
    return grid_search
```

[14]:
```
grid_search_log = cross_validate(LogisticRegression(), 'LogisticRegression',␣
 ↪classify=True)
grid_search_log.best_params_
```

```
Fitting 3 folds for each of 6 candidates, totalling 18 fits
```

[14]:
```
{'C': 0.8, 'class_weight': 'balanced'}
```

[15]:
```
grid_search_rfc = cross_validate(RandomForestClassifier(),␣
 ↪'RandomForestClassifier', classify=True)
grid_search_rfc.best_params_
```

```
Fitting 3 folds for each of 54 candidates, totalling 162 fits
```

[15]:
```
{'max_features': None,
 'min_impurity_decrease': 0.1,
 'min_samples_leaf': 1,
 'n_estimators': 1500}
```

[16]:
```
grid_search_rfr = cross_validate(RandomForestRegressor(),␣
 ↪'RandomForestRegressor', classify=True)
grid_search_rfr.best_params_
```

```
Fitting 3 folds for each of 54 candidates, totalling 162 fits
```

[16]:
```
{'max_features': 'auto',
 'min_impurity_decrease': 0.1,
 'min_samples_leaf': 1,
 'n_estimators': 1000}
```