Analysis of the Data

From the provided dataset details, I decided to build a model which trains against **_1s_side** in order to predict the various changes in price (0,1 or 2).

Preprocessing of data:

Looking at df.head(), I deduced that the timestamp would not contribute much to the prediction as a feature, hence it is used as an index instead.
After plotting a pairplot on the dataframe, correlation between the bid_price,ask_price and trade_price is very high (Figure 1). Hence, I only kept bid_price as features should be independent of one another.

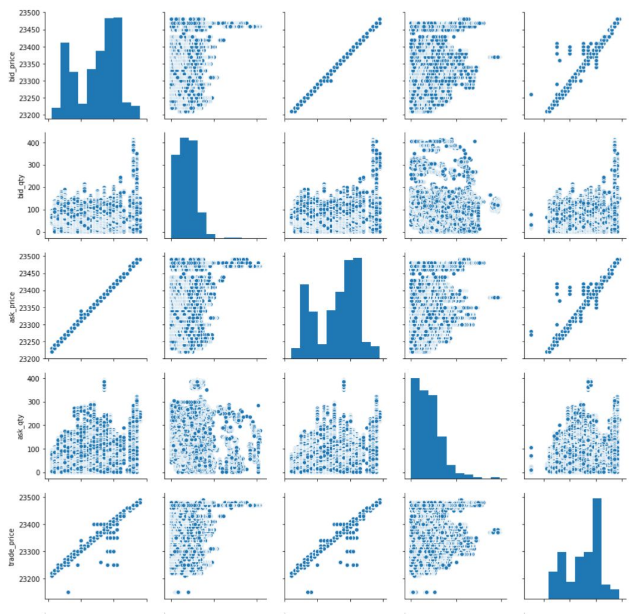
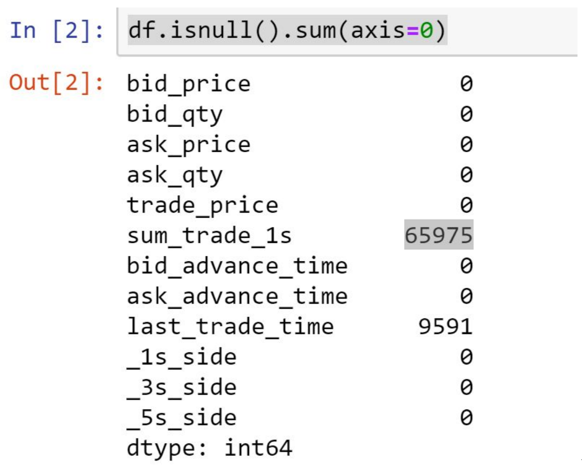


Figure 1: part of the dataframe pairplot

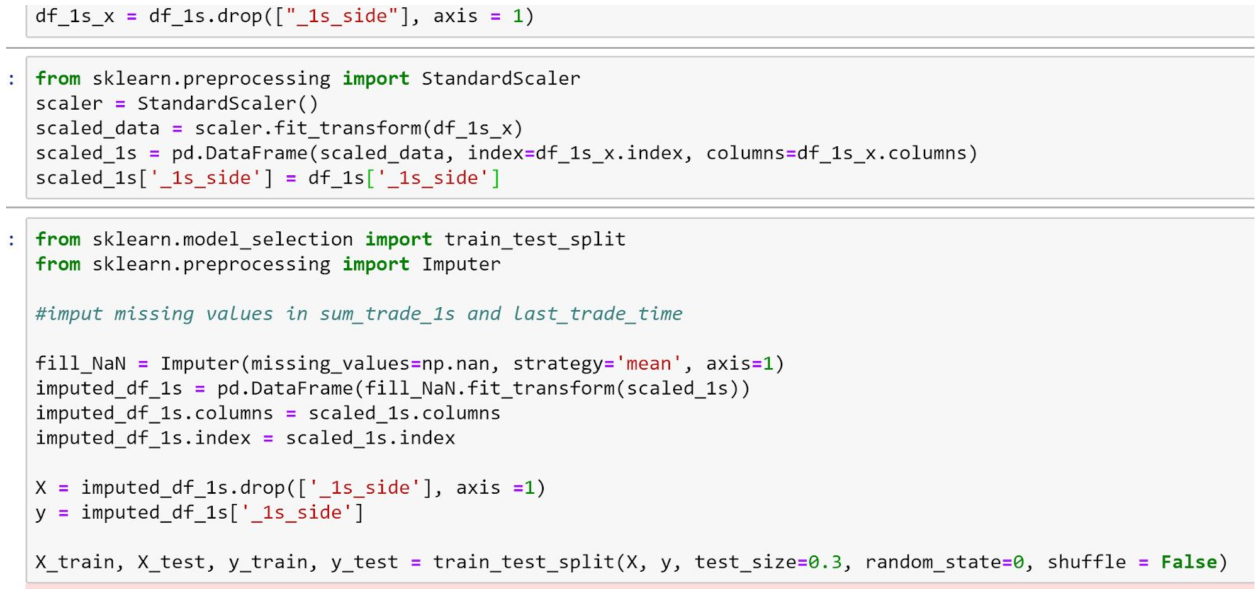
Using df.isnull().sum(axis=0), I found out that there are missing values aka NaNs in the data(Figure 2):



```
In [2]: df.isnull().sum(axis=0)

Out[2]: bid_price               0
        bid_qty                 0
        ask_price               0
        ask_qty                 0
        trade_price             0
        sum_trade_1s        65975
        bid_advance_time        0
        ask_advance_time        0
        last_trade_time      9591
        _1s_side                0
        _3s_side                0
        _5s_side                0
        dtype: int64
```

Figure 2: No. of NaNs in the data

This needs to be addressed by imputing. However, as I am imputing using mean, I scaled the data using StandardScaler before imputing as the order did not matter due to the fact that imputing will still take into account the mean of the whole scaled dataset. After that, I proceeded to split the data into train and test without shuffle as we should only be using past data to predict the future price change(due to presence of timestamp)(Figure 3).



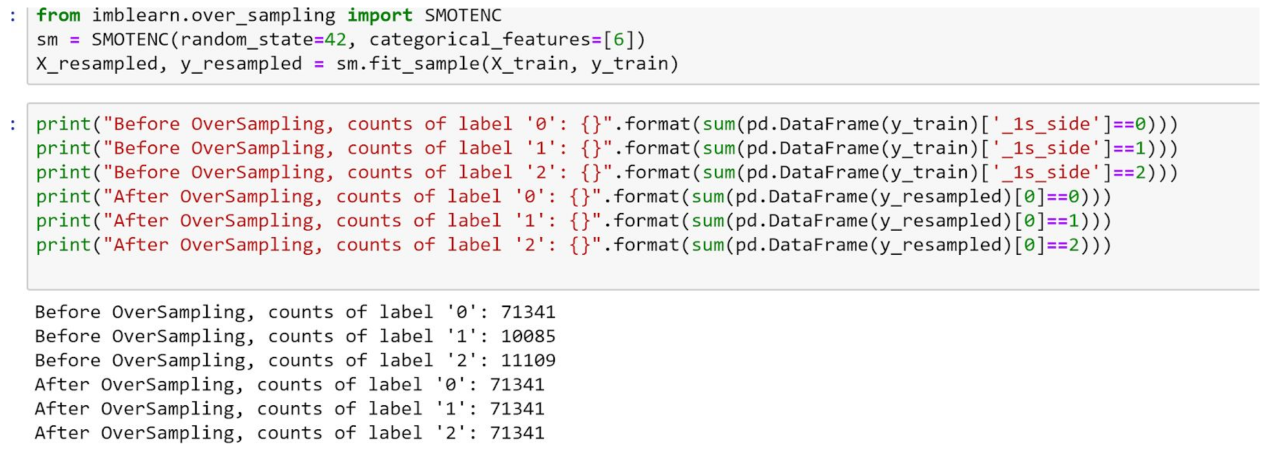
```
df_1s_x = df_1s.drop(["_1s_side"], axis = 1)
```

```
: from sklearn.preprocessing import StandardScaler
  scaler = StandardScaler()
  scaled_data = scaler.fit_transform(df_1s_x)
  scaled_1s = pd.DataFrame(scaled_data, index=df_1s_x.index, columns=df_1s_x.columns)
  scaled_1s['_1s_side'] = df_1s['_1s_side']
```

```
: from sklearn.model_selection import train_test_split
  from sklearn.preprocessing import Imputer

  #imput missing values in sum_trade_1s and last_trade_time

  fill_NaN = Imputer(missing_values=np.nan, strategy='mean', axis=1)
  imputed_df_1s = pd.DataFrame(fill_NaN.fit_transform(scaled_1s))
  imputed_df_1s.columns = scaled_1s.columns
  imputed_df_1s.index = scaled_1s.index

  X = imputed_df_1s.drop(['_1s_side'], axis =1)
  y = imputed_df_1s['_1s_side']

  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0, shuffle = False)
```

Figure 3: Scaling, Imputing and Train Test Split

While checking for imbalanced classes, I realised that class 1 and 2 are the minority class while 0 is the majority. Hence, SMOTENC resampling was applied to equalise the classes. SMOTE was not used as the values were continuous values due to scaling(Figure 4).



```
from imblearn.over_sampling import SMOTENC
sm = SMOTENC(random_state=42, categorical_features=[6])
X_resampled, y_resampled = sm.fit_sample(X_train, y_train)
```

```
print("Before OverSampling, counts of label '0': {}".format(sum(pd.DataFrame(y_train)['_1s_side']==0)))
print("Before OverSampling, counts of label '1': {}".format(sum(pd.DataFrame(y_train)['_1s_side']==1)))
print("Before OverSampling, counts of label '2': {}".format(sum(pd.DataFrame(y_train)['_1s_side']==2)))
print("After OverSampling, counts of label '0': {}".format(sum(pd.DataFrame(y_resampled)[0]==0)))
print("After OverSampling, counts of label '1': {}".format(sum(pd.DataFrame(y_resampled)[0]==1)))
print("After OverSampling, counts of label '2': {}".format(sum(pd.DataFrame(y_resampled)[0]==2)))
```

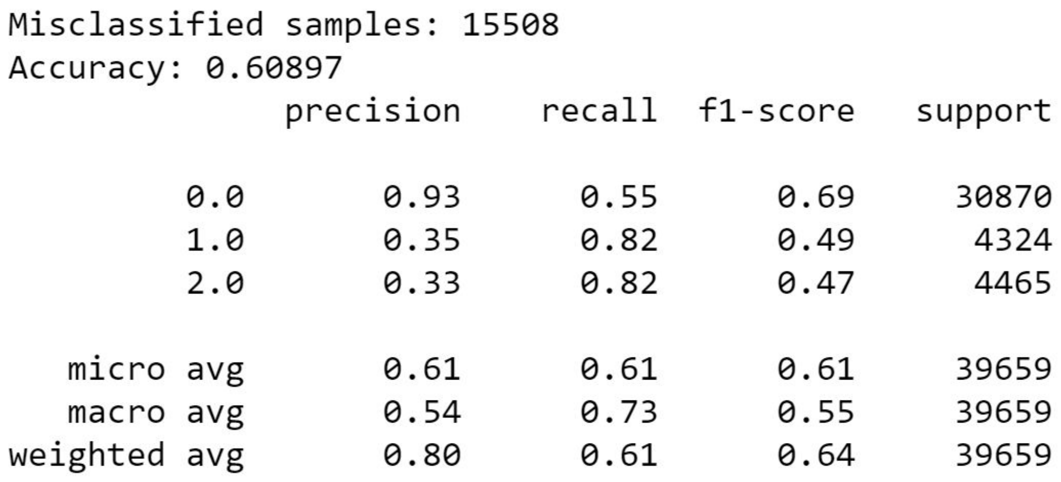
```
Before OverSampling, counts of label '0': 71341
Before OverSampling, counts of label '1': 10085
Before OverSampling, counts of label '2': 11109
After OverSampling, counts of label '0': 71341
After OverSampling, counts of label '1': 71341
After OverSampling, counts of label '2': 71341
```

Figure 4: SMOTENC oversampling

Results of Models:

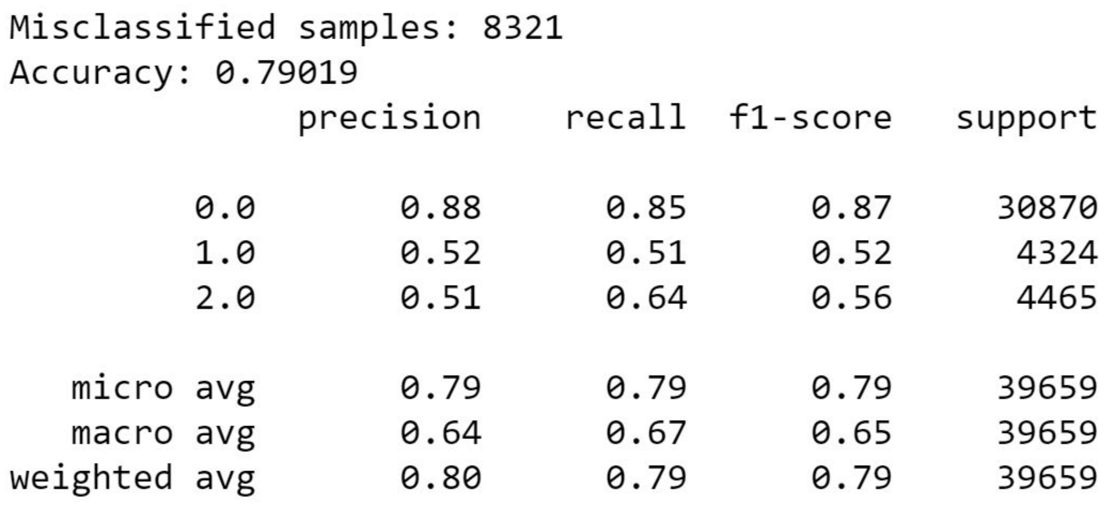
I tried out the following models: K-nearest neighbours, Decision Tree, Random Forest, kernelised Support Vector Machine with Radial Basis Function and an ensemble of a kernelised SVM, logistic regression and decision tree. The results are show below.

KNN



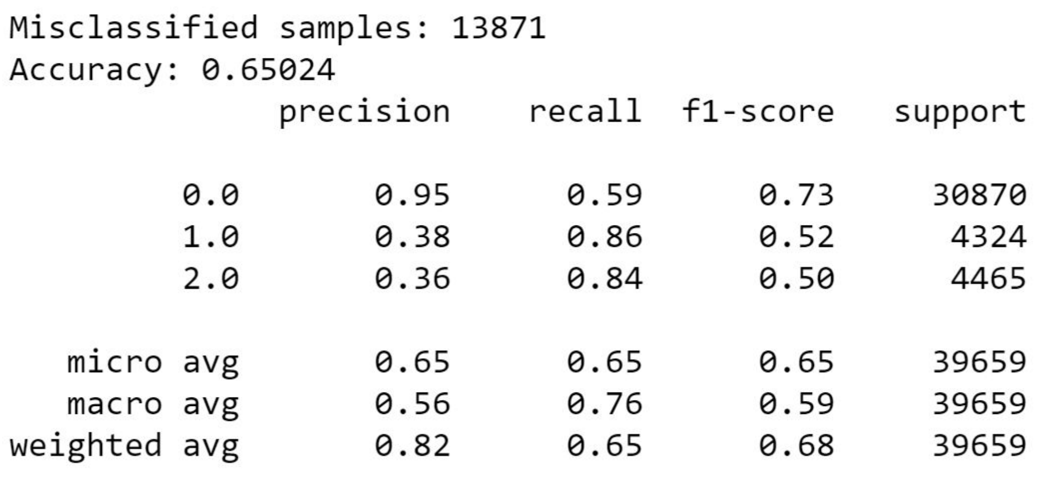
```
Misclassified samples: 15508
Accuracy: 0.60897
               precision    recall  f1-score   support

         0.0       0.93      0.55      0.69     30870
         1.0       0.35      0.82      0.49      4324
         2.0       0.33      0.82      0.47      4465

   micro avg       0.61      0.61      0.61     39659
   macro avg       0.54      0.73      0.55     39659
weighted avg       0.80      0.61      0.64     39659
```

Random Forest



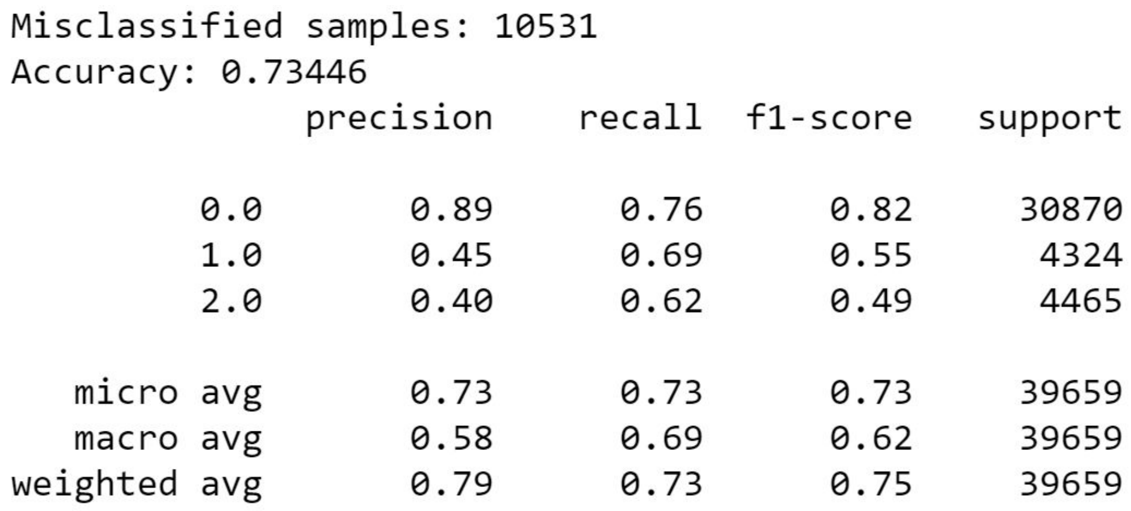
```
Misclassified samples: 8321
Accuracy: 0.79019
              precision    recall  f1-score   support

         0.0       0.88      0.85      0.87     30870
         1.0       0.52      0.51      0.52      4324
         2.0       0.51      0.64      0.56      4465

   micro avg       0.79      0.79      0.79     39659
   macro avg       0.64      0.67      0.65     39659
weighted avg       0.80      0.79      0.79     39659
```

Decision Tree:



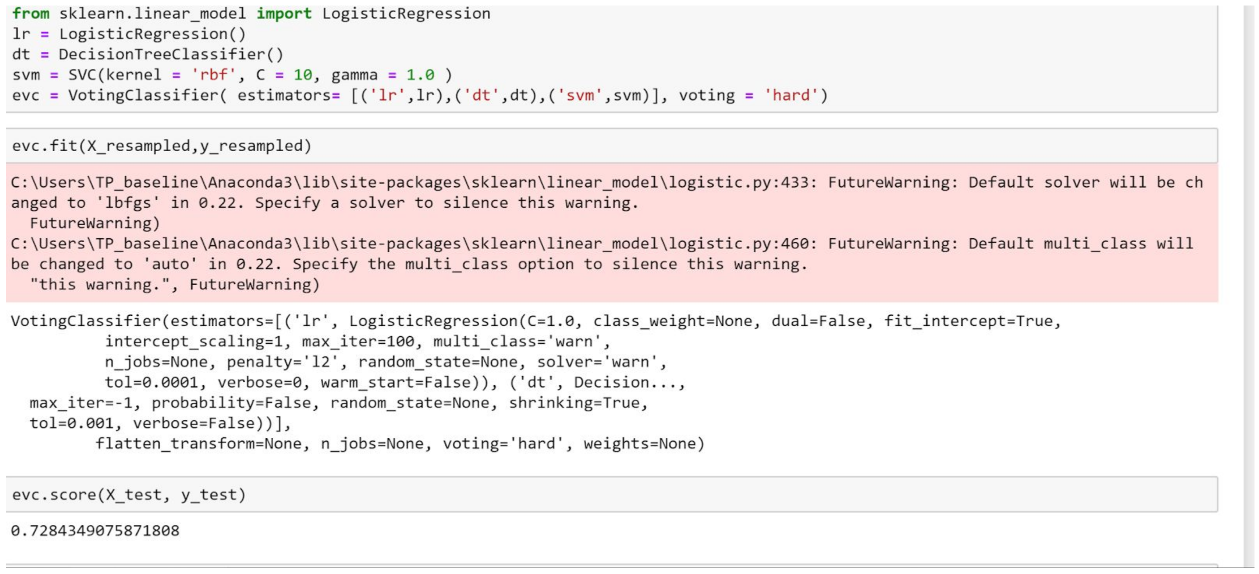
```
Misclassified samples: 13871
Accuracy: 0.65024
              precision    recall  f1-score   support

         0.0       0.95      0.59      0.73     30870
         1.0       0.38      0.86      0.52      4324
         2.0       0.36      0.84      0.50      4465

   micro avg       0.65      0.65      0.65     39659
   macro avg       0.56      0.76      0.59     39659
weighted avg       0.82      0.65      0.68     39659
```

SVM, rbf with C = 10, gamma = 1.0



```
Misclassified samples: 10531
Accuracy: 0.73446
              precision    recall  f1-score   support

         0.0       0.89      0.76      0.82     30870
         1.0       0.45      0.69      0.55      4324
         2.0       0.40      0.62      0.49      4465

   micro avg       0.73      0.73      0.73     39659
   macro avg       0.58      0.69      0.62     39659
weighted avg       0.79      0.73      0.75     39659
```

Ensemble of SVM rbf, Decision Tree and Logistic Regression using Voting



```
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression()
dt = DecisionTreeClassifier()
svm = SVC(kernel = 'rbf', C = 10, gamma = 1.0 )
evc = VotingClassifier( estimators= [('lr',lr),('dt',dt),('svm',svm)], voting = 'hard')
```

```
evc.fit(X_resampled,y_resampled)
```

```
C:\Users\TP_baseline\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:433: FutureWarning: Default solver will be ch
anged to 'lbfgs' in 0.22. Specify a solver to silence this warning.
  FutureWarning)
C:\Users\TP_baseline\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:460: FutureWarning: Default multi_class will
be changed to 'auto' in 0.22. Specify the multi_class option to silence this warning.
  "this warning.", FutureWarning)
```

```
VotingClassifier(estimators=[('lr', LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='warn',
          n_jobs=None, penalty='l2', random_state=None, solver='warn',
          tol=0.0001, verbose=0, warm_start=False)), ('dt', Decision...,
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False))],
         flatten_transform=None, n_jobs=None, voting='hard', weights=None)
```

```
evc.score(X_test, y_test)
```

```
0.7284349075871808
```

In conclusion, the random forest model with 1000 decision trees performed the best at classification, although it did not do a great job against the former minority class 1 and 2.