## Problem 1 One-by-one Feature Selection

- Method:

In this problem, we use the Similarity-Based: **Fisher Score** method, which refers to lecture slide: DSLecture 5 Feature Selection p.25.

Hence, we follow the formula of Fisher score on lecture slide and use it as our score function for each feature.

Hence, our implementation is: For each feature, calculate its score function, and sort these features based on their score function.

- Experiment Result:

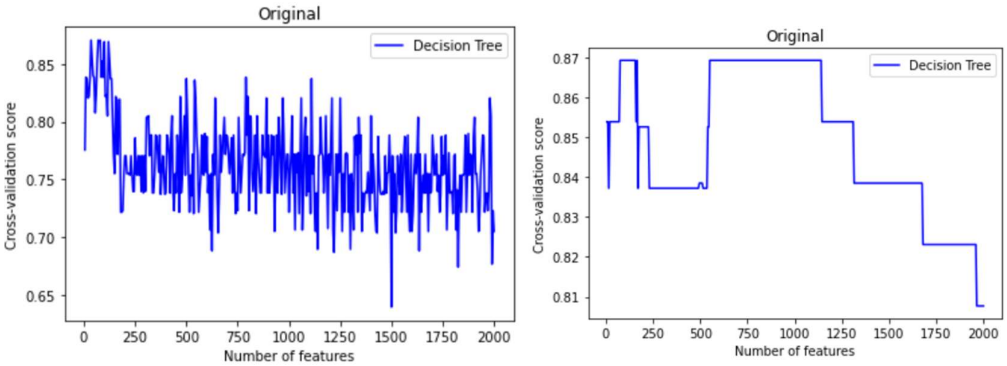
At first, to ensure that our implementation is right, we sort the features in the ranking_idx **randomly**. And get the better result of score/# of feature = 0.8538461538461538 / 460 than SVC. So we can use it as our baseline.

Then we implement prob.1 and use the sample code in feature evaluation part. We get results:

| | **DecisionTree** | **SVC** |
|---|---|---|
| **scores** | 0.8705128205128204 | 0.8692307692307694 |
| **Number of feature** | 35 | 75 |

Visualization:

Decision Tree: SVC:



Comment:

As above, we can see that our implementation indeed has improved, whose results are better than that of the baseline.

However, although the **features are cut significantly**, we can see that the **improvement in the score is slight**

Moreover, we can see the visualization of two methods, both of **can't see a positive trend** of the curve. It means **the order of features in ranking_idx is not exactly representing the significance of feature evaluation**, else we should observe a positive trend since as we add more "significant"(at the front of ranking_idx) features into evaluation, we should get a better score.

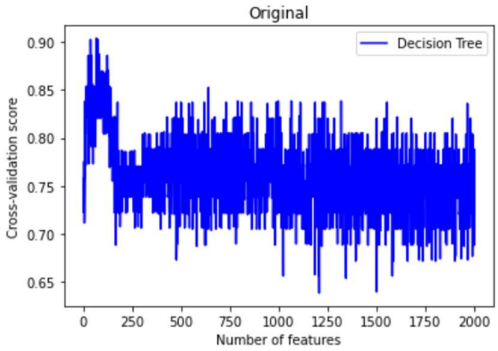
Or discussing more precisely, ideally, ranking_idx sort features based on the significance of feature evaluation of each feature. We can imagine that the front part of features is good for feature evaluation, the medium part of features can be seen as no help on evaluation, and the end part of features may worsen the evaluation. Hence, the visualization curve should have a positive trend, flat trend, and negative trend when we take more features into consideration. However, we observe the curve differs far away from the ideal, so we can say that one-by-one selection is not a good method.

Next, we observe that the sample code only examines the score that takes the first 5*n features into evaluation and ignores other possibilities. Hence, we next try to **examine the score that takes the first n features into evaluation**. We get the results:

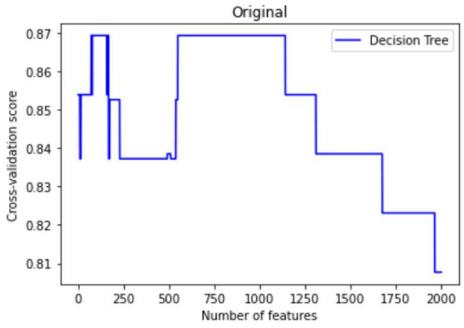
| | **DecisionTree** | **SVC** |
|---|---|---|
| **scores** | 0.9038461538461539 | 0.8692307692307694 |
| **Number of feature** | 67 | 72 |

Visulization:

Decision Tree:




SVC:




Comment:

As above, we can see that we improve the score to about 0.9 and cut #_of_features to 67. However, the problem discussed above still exists since we use the same score function in this experiment.

## Problem 2 Subset-Based Feature Selection

The metaheuristic we choose here is **PSO (particle swarm optimization)**.

- Idea of the algorithm:

Following the idea of PSO, we construct a search place with 2000-D(since the dataset has 2000 features) and boundaries [0,1] at each dimension. Each particle in the search space represents a “method” of feature selection. Given a particle x, we have the selection method:

**For i=1~2000, feature fi is chosen if x[i]>0.5, else we don’t choose fi.**

Where x[i] means the position of x along the i-th dimension.

Hence, we run the PSO algorithm to find the best global position, which gives the best feature selection that achieves a minimal cost function during training.

- Algorithm:

The objective function(/cost function) we use is:

**f(x) = a * (1 - P) + (1-a) *gamma*(#_selected_features/#_features).**

Where P is the **feature evaluation score**. We use the method in the sample code and choose the **Decision Tree with 5-fold validation** method.

Since we want to **maximize P and minimize the number of features** we choose, we need to **minimize f(x)**.

And “a” is a tunable parameter between 0-1 that we can tune if we prefer a **better evaluation score** or a **fewer number of features selection**.

And “gamma” is a scaling factor that balances the importance of “P” and “#_selected_features”.

As for the implementation of PSO algorithm, we refer to the lecture slide: DSLecture 5 Feature Selection p.50 and implement the PSO algorithm.

However, there are some differences in my algorithm.

At first, we add a tunable parameter **V_max** so that the velocity won’t be too large to make particles move out of search space.

And we also add security to ensure that after a particle updates its position, it won’t go beyond the search space.

Moreover, we make “**w**”, the weight of the original velocity of particles, linearly decay to gradually slow down the pace during training.

The tunable parameters are as below:

1. a : a tunable parameter in the objective function. a = 0 – 1
2. gamma: a scaling factor in the objective function. Default set gamma=1
3. w: weight of particle’s original velocity
4. phi_p: weight of particle’s probabilistic velocity towards local best-known position

5. phi_g: weight of particle's probabilistic velocity towards global best-known position
6. lr: the proportion at which the velocity affects the movement of the particle. Here, we **don't restrict lr=0~1** since we have no idea to tune w, phi_p, and phi_g parameters.
7. Iteration: the round of PSO algorithm would run
8. S: number of particles in the search space
9. K: Vmax = k*((b_up-b_low)/2) = 0.5*k , k = 0~1. Default set k=1.
10. w_decay: Boolean to decide if "w" is a constant or would linearly decay during training.

- Intuitive parameter setting:

To find an appropriate "a", under the condition: gamma=1 and other parameters are set "normally"(weights of velocity≒1, iteration=10, S=100, lr=0.8), we have tried a=0.1 and a=0.9, getting results:

| | **a=0.1** | **a=0.9** |
|---|---|---|
| **P** | 0.8679487179487179 | 0.9012820512820513 |
| **#_of_selected_feature** | 876 | 1015 |

As above, we can see that if we choose too small "a", "P" may be even worse than the baseline. To see if this method can achieve a better P than that in problem1, we should choose a = 0.9.

To find appropriate velocity weights, which contain "w","phi_p", and "phi_g", we can see the formula in the PSO algorithm that says:

**vi,d = w * vi,d + phi_p*rp*(pi,d-xi,d) + phi_g*rg*(gd - xi,d)**

Hence, since the search base has boundary [0,1], we should want **vi,d has an average value = 0.2**. Moreover, since rp,rg ~U(0,1), which has an average value = 0.5, and we estimate that (pi,d-xi,d),(gd - xi,d) have an average value = 0.5 since both pi,d/gd and xi,d are bound in [0,1]. Finally, we want three

velocities to have the same contribution to vi,d, we should **set “w”,”phi_p”, and “phi_g” to a ratio about 1:1:1** since in average, vi,d = w*0.25+phi_p*0.5*0.5+ phi_g*0.5*0.5. (here we assume vi,d has average value=0.25 for convenience)

To find an appropriate “S”, since the search space is 2000-D, we should set S to an order of 2000. However, considering the training time, we choose S=1000.

Hence, we initialize parameters as below:

a = 0.9

gamma=1

w = 0.9 / 3

phi_p = 0.8 / 3

phi_g = 0.8 / 3

lr = 1

iteration = 20

S = 100

k = 1

w_decay = False

- Experiment for parameters tuning:

Exp1: find an appropriate lr

In Exp1, we tune lr to find which lr can achieve better performance. It will determine the “exact” average velocity during training.

Exp2: find appropriate velocity weights ratio

In Exp2, we use the best lr we found in Exp1, and we tune different velocity weights ratios to see which ratio can achieve better performance.

Exp3: find appropriate “a”

In Exp3, we use the best parameters found in Exp2, and we tune different “a” to see which one can achieve better performance.

Exp4: see the effect of w_decay

In Exp4, we use the best parameters found in Exp3, and we turn w_decay = True. Then test bigger and smaller “w” to see if it helps on boosting performance.

- Experiment Result:

Exp1:

We set

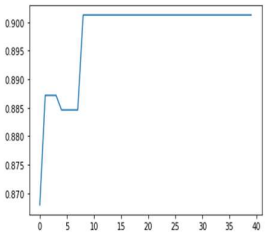
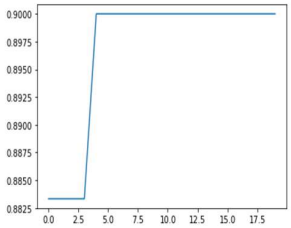
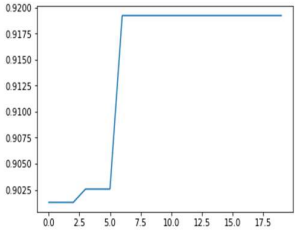
a=0.9 gamma=1 w=0.9/3 phi_p=0.8/3 phi_g=0.8/3 iter=20/40 S=1000 k=1

and tune different lr to results:

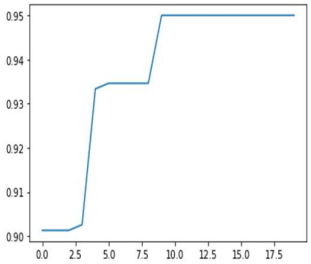
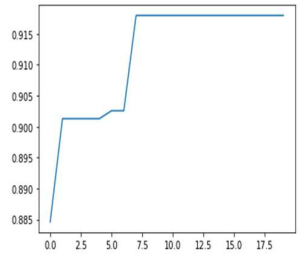
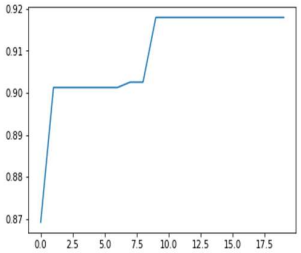
| | P | #_selected_feature |
|---|---|---|
| lr=0.2 (iter=40) | 0.9013 | 975 |
| lr=1 (iter=20) | 0.9000 | 981 |
| lr=2.8 (iter=20) | 0.9192 | 977 |
| lr=3 (iter=20) | 0.9500 | 988 |
| lr=3.2 (iter=20) | 0.9179 | 922 |
| lr=5 (iter=20) | 0.9179 | 859 |
| lr=8 (iter=20) | 0.8833 | 1038 |

Visualization: (x-axis: iteration times, y-axis: P(evaluation score))

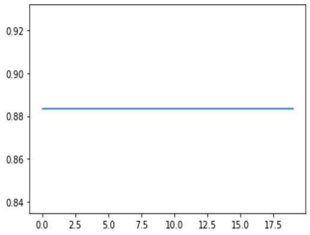
lr=0.2 lr=1 lr=2.8

lr=3 lr=3.2 lr=5

lr=8



Comment:

As above, we have observations below:

1. At smaller lr(=0.2), it takes more rounds to achieve its best performance.
2. At too big or too small lr, performance gets worse.
3. lr = 3 has the best performance.

Exp2:

Using the result of Exp1, we set

a=0.9 gamma=1 lr = 3 iter=20 S=1000 k=1

and tune the ratio of three velocity weights while keeping the average velocity at 0.2. That is, if w : phi_p : phi_g = 3 : 1 : 1, we set

w=0.8*3/5 phi_p=0.8*1/5 phi_g=0.8*1/5

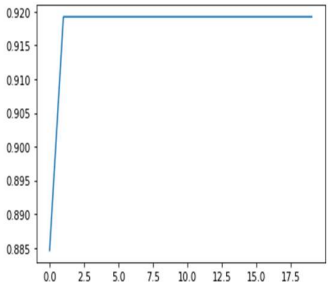
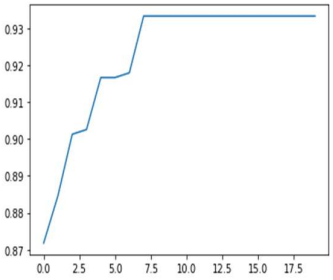
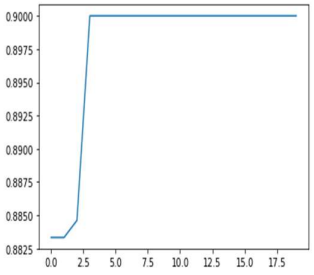
since vi,d = w*0.25+phi_p*0.5*0.5+ phi_g*0.5*0.5

Results:

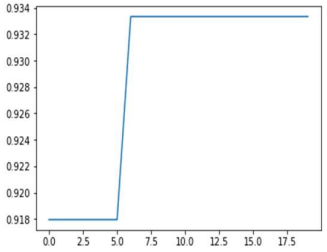
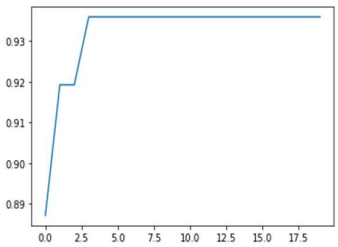
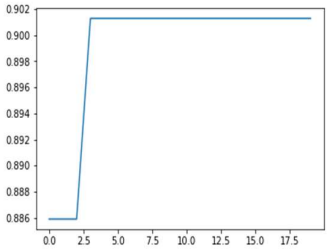
| w : phi_p : phi_g | P | #_selected_feature |
|---|---|---|
| 3 : 1 : 1 (w big) | 0.9192 | 891 |
| 1 : 3 : 3 (w small) | 0.9333 | 957 |
| 1 : 3 : 1 (phi_p big) | 0.9000 | 909 |
| 3 : 1 : 3 (phi_p small) | 0.9333 | 764 |
| 1 : 1 : 3 (phi_g big) | 0.9359 | 948 |
| 3 : 3 : 1 (phi_g small) | 0.9013 | 825 |

Visualization:

w big: w small: phi_p big

phi_p small: phi_g big: phi_g small:

Comment:

As above, we have observations below:

1. If we make ”w” or “phi_p” bigger, performance gets worse. Since the global best position is the really best, whose contribution should not be less.
2. However, the experiment results told us that setting the ratio of three velocity weights to 1:1:1 may be the best, which achieves the highest P.
3. There are some explanations for observation 2.. Maybe it is True that the ratio 1:1:1 is the best. Or it is just luck that we see it the best. Or we may have misestimated the formula of average velocity, which may need more rigorous probabilistic provement.

Exp3:

Based on the result of Exp1 and Exp2, we set:

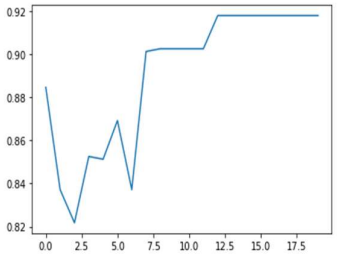
gamma= 1  w=0.9/3  phi_p=0.8/3  phi_g=0.8/3  lr=3 iter=20  S=1000  k=1

and tune “a” to get results:

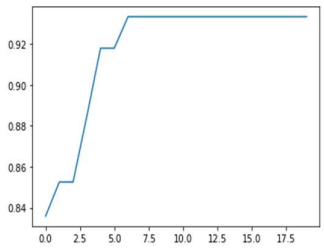
| | P | #_selected_feature |
|---|---|---|
| a=0.1 | 0.9179 | 543 |
| a=0.3 | 0.9333 | 684 |
| a=0.5 | 0.9192 | 843 |
| a=0.7 | 0.9000 | 939 |
| a=0.8 | 0.9000 | 1006 |
| a=1 | 0.8859 | 994 |

Visualization:

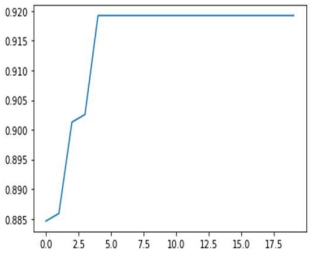
a=0.1



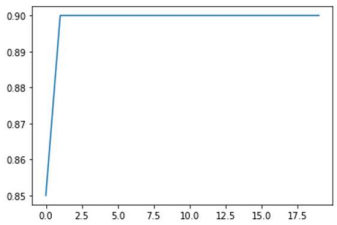

a=0.3



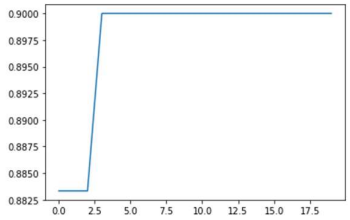

a=0.5



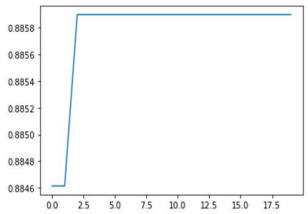

a=0.7




a=0.8




a=1




Comment:

As above, we have observations below:

1. It is indeed that when we have “a” smaller, we minimize the #_selected_feature, while when “a” is bigger, P doesn’t correspondingly get bigger. We guess it’s because the contribution of P and that of #_selected_feature to f(x) are not the same since the former has an average of **0.9** and the latter has an average of **0.5**. We guess that tuning “gamma” can improve it.
2. At small “a”, since the y-label of the plot is “P”, instead of f(x), the curve may not monotonically increase when P’s contribution is small.

Exp3.5:

To validate comment 1. In Exp3, we try to tune **gamma=1.8**

(since 0.9=1.8*0.5) and do Exp3 again. We get the results:

| | P | #_selected_feature |
|---|---|---|
| a=0.1 | 0.8679 | 547 |
| a=0.3 | 0.9026 | 542 |
| a=0.5 | 0.9013 | 709 |
| a=0.7 | 0.9167 | 846 |
| a=0.8 | 0.9167 | 886 |
| a=1 | 0.9346 | 934 |

As above, we can see **the statement in our comment 1. In Exp3 is True**.

Exp4:

Based on the result of Exp1, Exp2, and Exp3, we set:

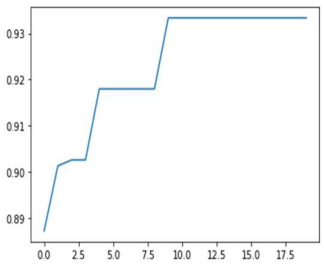
gamma= 1 w=0.9/3 phi_p=0.8/3 phi_g=0.8/3 lr=3 iter=20 S=1000 k=1

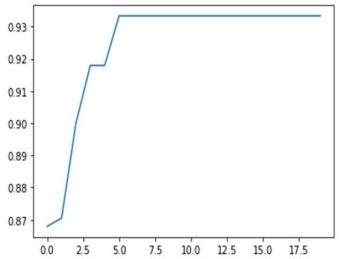
, set w_decay = True and tune “w” to get results:

| | P | #_selected_feature |
|---|---|---|
| w=0.3/3 | 0.9333 | 1009 |
| w=0.9/3 | 0.9026 | 986 |
| w=1.5/3 | 0.9333 | 750 |

Visualization:

w=0.3/3: w=1.5/3:




Comment:

As above, we have observations below:

1. “w_decay” doesn’t affect the performance too much, even may worsen the performance.
2. “w_decay” keeps a high P even we tune the ratio of velocity weights just as we do in Exp2.

Conclusion:

After the above experiments, we have set

**a = 0.9  gamma=1  w = 0.9 / 3  phi_p = 0.8 / 3  phi_g = 0.8 / 3  lr = 3**

**iteration = 20  S = 100  k = 1  w_decay = False**

to achieve a high evaluation score of P = 0.95, with #_selected_feature = 988

And the index of selected features are:

[0, 1, 4, 6, 7, 8, 9, 10, 11, 15, 17, 18, 23, 25, 26, 27, 28, 30, 32, 34, 35, 36, 38, 41, 42, 45, 53, 58, 62, 67, 68, 70, 75, 80, 81, 84, 86, 93, 94, 96, 103, 104, 105, 108, 109, 110, 113, 114, 115, 118, 120, 121, 122, 123, 124, 125, 127, 128, 135, 138, 139, 140, 143, 144, 145, 146, 147, 148, 150, 152, 153, 156, 157, 159, 162, 163, 165, 166, 170, 171, 172, 173, 174, 175, 176, 177, 179, 180, 181, 182, 183, 185, 187, 198, 199, 200, 202, 203, 204, 205, 206, 208, 211, 214, 215, 217, 218, 221, 224, 225, 227, 228, 229, 230, 232, 233, 235, 237, 239, 240, 241, 243, 244, 247, 251, 258, 260, 261, 262, 265, 269, 270, 271, 273, 274, 275, 277, 284, 294, 295, 298, 299, 300, 301, 303, 306, 307, 308, 314, 316, 318, 321, 322, 325, 330, 332, 334, 336, 337, 338, 339, 340, 341, 342, 344, 345, 346, 351, 352, 353, 354, 356, 357, 358, 369, 371, 372, 374, 376, 377, 380, 381, 383, 384, 386, 387, 390, 391, 392, 393, 394, 395, 398, 399, 401, 402, 403, 405, 406, 408, 410, 411, 413, 414, 417, 418, 419, 422, 424, 426, 427, 428, 429, 430, 431, 432, 433, 435, 437, 440, 448, 449, 453, 455, 456, 459, 463, 467, 468, 469, 470, 471, 472, 473, 474, 476, 477, 479, 484, 485, 488, 489, 491, 492, 496, 498, 502, 503, 504, 505, 506, 508, 509, 510, 511, 513, 516, 517, 520, 521, 524, 525, 526, 527, 532, 533, 534, 535, 538, 540, 541, 543, 544, 545, 547, 553, 555, 557, 558, 563, 564, 566,

568, 569, 570, 572, 574, 575, 576, 577, 578, 583, 584, 585, 588, 590, 591,
593, 596, 597, 598, 599, 600, 602, 607, 610, 612, 613, 614, 615, 617, 620,
621, 624, 626, 627, 629, 631, 632, 635, 638, 641, 642, 644, 645, 648, 649,
650, 651, 652, 654, 656, 657, 659, 660, 662, 664, 665, 672, 673, 674, 677,
679, 684, 686, 687, 689, 690, 693, 694, 695, 697, 698, 699, 700, 701, 703,
704, 707, 709, 710, 714, 715, 718, 720, 724, 725, 726, 727, 729, 731, 734,
735, 736, 738, 739, 742, 745, 755, 760, 762, 764, 767, 768, 770, 771, 772,
773, 774, 776, 778, 779, 781, 784, 789, 792, 795, 797, 800, 801, 811, 813,
814, 815, 819, 821, 823, 824, 825, 828, 829, 830, 831, 832, 835, 836, 837,
841, 843, 844, 846, 847, 849, 850, 852, 855, 857, 860, 861, 863, 864, 869,
870, 871, 873, 874, 876, 878, 879, 880, 881, 883, 886, 887, 888, 890, 891,
892, 894, 896, 897, 900, 901, 902, 903, 904, 905, 906, 908, 909, 911, 916,
920, 921, 923, 925, 927, 929, 931, 933, 935, 939, 940, 944, 948, 949, 951,
952, 953, 963, 964, 965, 966, 969, 970, 971, 975, 977, 979, 980, 981, 982,
983, 984, 990, 991, 993, 994, 998, 999, 1000, 1001, 1003, 1009, 1010, 1013,
1014, 1016, 1017, 1018, 1019, 1021, 1024, 1026, 1027, 1028, 1030, 1031,
1034, 1042, 1045, 1047, 1051, 1053, 1054, 1058, 1059, 1063, 1065, 1067,
1071, 1072, 1077, 1082, 1083, 1086, 1090, 1096, 1098, 1099, 1100, 1102,
1103, 1104, 1105, 1106, 1107, 1108, 1110, 1112, 1113, 1118, 1120, 1122,
1123, 1124, 1126, 1129, 1131, 1133, 1134, 1140, 1141, 1142, 1145, 1150,
1151, 1152, 1156, 1157, 1158, 1161, 1163, 1164, 1165, 1167, 1168, 1170,
1171, 1173, 1177, 1181, 1182, 1183, 1184, 1185, 1188, 1189, 1197, 1199,
1200, 1202, 1204, 1205, 1214, 1215, 1216, 1217, 1219, 1220, 1224, 1225,
1226, 1228, 1230, 1231, 1232, 1233, 1235, 1237, 1238, 1240, 1242, 1248,
1249, 1251, 1252, 1253, 1254, 1257, 1258, 1259, 1264, 1271, 1273, 1274,
1275, 1282, 1283, 1285, 1289, 1291, 1294, 1297, 1298, 1302, 1305, 1307,
1310, 1314, 1319, 1322, 1324, 1325, 1326, 1327, 1328, 1329, 1332, 1336,
1338, 1343, 1345, 1351, 1355, 1358, 1360, 1363, 1365, 1367, 1369, 1371,
1372, 1373, 1375, 1376, 1379, 1381, 1382, 1383, 1384, 1386, 1390, 1391,
1395, 1396, 1397, 1399, 1401, 1402, 1403, 1404, 1406, 1407, 1408, 1410,
1411, 1413, 1415, 1419, 1421, 1423, 1424, 1425, 1427, 1431, 1432, 1433,
1438, 1439, 1440, 1441, 1443, 1444, 1445, 1446, 1448, 1449, 1450, 1451,
1455, 1456, 1457, 1458, 1459, 1460, 1463, 1466, 1468, 1470, 1472, 1473,
1474, 1476, 1477, 1478, 1480, 1482, 1487, 1488, 1489, 1491, 1492, 1494,
1498, 1500, 1502, 1505, 1506, 1507, 1508, 1510, 1512, 1513, 1514, 1515,
1517, 1519, 1520, 1522, 1523, 1524, 1525, 1526, 1527, 1528, 1530, 1532,

1534, 1535, 1539, 1541, 1542, 1543, 1545, 1546, 1549, 1550, 1553, 1554, 1557, 1558, 1563, 1565, 1569, 1570, 1572, 1573, 1574, 1576, 1578, 1581, 1582, 1583, 1585, 1587, 1588, 1589, 1591, 1595, 1597, 1603, 1607, 1608, 1609, 1610, 1611, 1612, 1618, 1619, 1623, 1624, 1625, 1628, 1630, 1632, 1633, 1637, 1638, 1639, 1641, 1642, 1643, 1644, 1646, 1649, 1650, 1652, 1653, 1655, 1657, 1660, 1664, 1665, 1668, 1669, 1673, 1674, 1675, 1678, 1679, 1680, 1681, 1684, 1685, 1687, 1689, 1691, 1693, 1695, 1696, 1701, 1702, 1703, 1705, 1707, 1709, 1710, 1711, 1716, 1717, 1718, 1719, 1724, 1726, 1727, 1728, 1730, 1731, 1732, 1734, 1735, 1736, 1737, 1739, 1740, 1741, 1742, 1743, 1744, 1748, 1749, 1751, 1752, 1754, 1756, 1759, 1761, 1762, 1763, 1767, 1768, 1770, 1773, 1775, 1776, 1777, 1779, 1780, 1783, 1789, 1792, 1793, 1794, 1796, 1805, 1806, 1809, 1810, 1816, 1817, 1820, 1821, 1827, 1828, 1829, 1832, 1836, 1837, 1839, 1840, 1843, 1845, 1846, 1847, 1848, 1850, 1851, 1858, 1863, 1865, 1868, 1870, 1871, 1872, 1873, 1877, 1881, 1883, 1885, 1887, 1888, 1889, 1892, 1893, 1895, 1899, 1901, 1904, 1906, 1911, 1916, 1917, 1918, 1921, 1922, 1925, 1926, 1929, 1930, 1931, 1933, 1936, 1940, 1942, 1943, 1944, 1946, 1947, 1948, 1953, 1954, 1955, 1956, 1957, 1958, 1960, 1963, 1964, 1965, 1968, 1970, 1972, 1973, 1976, 1979, 1982, 1985, 1986, 1987, 1988, 1989, 1990, 1995, 1996, 1997, 1998, 1999]

, total of 988 features.

Alternatively, if 988 features are too much, we can set

a=0.1 gamma=1 w=0.9/3 phi_p=0.8/3 phi_g=0.8/3 lr=3 iter=20

S=1000 k=1

to **achieve a normal evaluation score of P = 0.918**, which is still better than the baseline and the best result in problem 1, with **#_selected_feature = 543**

And the index of selected features are:

[0, 4, 5, 7, 8, 11, 12, 14, 20, 27, 30, 32, 33, 36, 41, 45, 53, 54, 55, 58, 59, 60, 70, 75, 79, 81, 86, 88, 90, 94, 103, 106, 107, 108, 118, 122, 125, 127, 135, 139, 145, 147, 149, 151, 152, 162, 163, 164, 167, 175, 177, 179, 188, 190, 191, 197, 198, 205, 213, 215, 216, 217, 219, 222, 224, 233, 234, 235, 248,

249, 254, 255, 272, 273, 280, 282, 285, 291, 296, 297, 298, 303, 305, 310,
312, 313, 314, 315, 316, 317, 319, 320, 327, 328, 334, 335, 344, 357, 360,
361, 362, 363, 366, 368, 370, 371, 373, 374, 378, 382, 384, 387, 406, 408,
409, 415, 424, 426, 432, 433, 435, 441, 442, 443, 445, 447, 454, 461, 463,
468, 469, 471, 477, 478, 485, 504, 505, 506, 508, 514, 516, 517, 522, 525,
537, 543, 550, 553, 554, 555, 557, 559, 567, 570, 572, 574, 577, 588, 591,
593, 599, 600, 603, 605, 606, 613, 617, 621, 626, 628, 629, 632, 642, 646,
648, 650, 651, 656, 659, 663, 675, 676, 678, 685, 698, 714, 717, 718, 722,
725, 730, 732, 734, 737, 741, 743, 745, 751, 752, 759, 770, 779, 781, 788,
789, 791, 798, 799, 801, 803, 809, 810, 813, 818, 819, 820, 824, 827, 828,
835, 836, 862, 864, 865, 867, 870, 878, 884, 887, 890, 895, 898, 900, 909,
912, 913, 925, 931, 935, 936, 950, 953, 955, 958, 965, 966, 968, 973, 980,
989, 990, 992, 993, 1008, 1016, 1017, 1018, 1027, 1028, 1035, 1037, 1043,
1044, 1047, 1051, 1052, 1056, 1059, 1072, 1073, 1074, 1077, 1078, 1079,
1085, 1092, 1094, 1096, 1100, 1102, 1105, 1109, 1111, 1112, 1113, 1117,
1120, 1125, 1126, 1127, 1129, 1133, 1136, 1137, 1139, 1142, 1148, 1160,
1167, 1168, 1170, 1174, 1178, 1185, 1186, 1188, 1190, 1192, 1194, 1197,
1199, 1201, 1204, 1207, 1213, 1216, 1223, 1224, 1225, 1228, 1234, 1236,
1238, 1250, 1252, 1253, 1254, 1255, 1256, 1259, 1260, 1262, 1265, 1269,
1270, 1272, 1274, 1280, 1287, 1292, 1295, 1305, 1310, 1313, 1314, 1316,
1317, 1322, 1323, 1324, 1326, 1331, 1340, 1345, 1347, 1348, 1352, 1355,
1357, 1358, 1364, 1368, 1374, 1378, 1384, 1386, 1390, 1391, 1398, 1402,
1411, 1413, 1415, 1417, 1418, 1420, 1433, 1441, 1443, 1448, 1449, 1452,
1453, 1459, 1465, 1472, 1474, 1475, 1491, 1494, 1497, 1500, 1502, 1504,
1511, 1514, 1522, 1523, 1525, 1528, 1529, 1530, 1531, 1536, 1539, 1541,
1546, 1547, 1551, 1557, 1563, 1566, 1569, 1570, 1571, 1575, 1576, 1583,
1585, 1589, 1593, 1599, 1604, 1606, 1607, 1609, 1615, 1616, 1617, 1619,
1625, 1626, 1627, 1629, 1632, 1636, 1637, 1638, 1642, 1644, 1646, 1648,
1650, 1651, 1654, 1655, 1663, 1664, 1666, 1667, 1705, 1707, 1710, 1711,
1718, 1724, 1728, 1729, 1736, 1737, 1744, 1748, 1750, 1751, 1752, 1754,
1755, 1759, 1760, 1763, 1768, 1771, 1776, 1781, 1786, 1788, 1794, 1795,
1798, 1802, 1809, 1810, 1811, 1814, 1815, 1818, 1820, 1821, 1822, 1823,
1828, 1829, 1837, 1843, 1844, 1845, 1854, 1860, 1862, 1863, 1867, 1869,
1873, 1875, 1876, 1882, 1885, 1891, 1892, 1895, 1896, 1901, 1902, 1907,
1915, 1917, 1926, 1927, 1929, 1931, 1932, 1934, 1943, 1946, 1948, 1952,

1953, 1954, 1956, 1957, 1961, 1966, 1973, 1975, 1977, 1979, 1981, 1982, 1984, 1986, 1988, 1992, 1993]

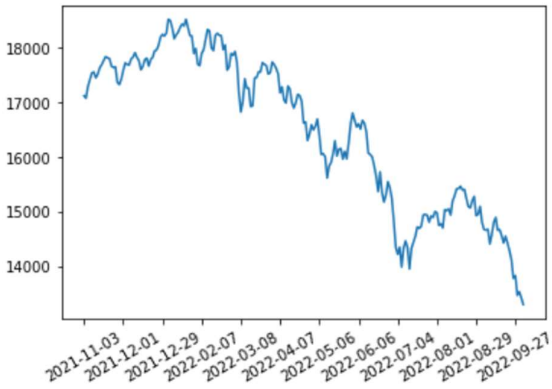
, total of 543 features.

## Problem 3 ARIMA Forecast

- Analyze Data

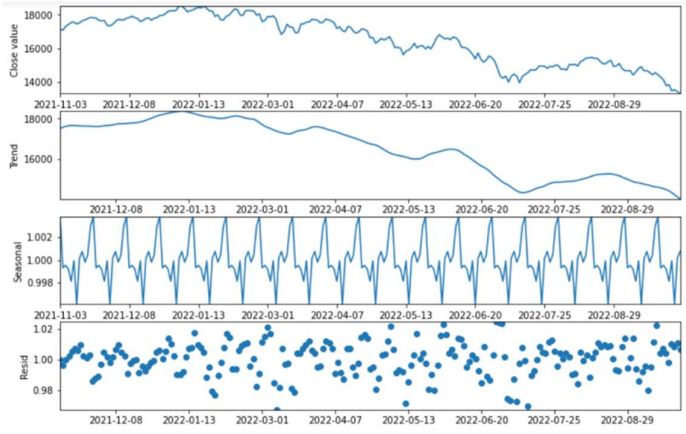
At first, we plot some properties of data to know more about the data.

The “close” value data is as below:




We can see that the “close” value has a trend of **getting smaller**.

Next, we use ETS(error-trend-seasonality) models to see more information in the data:



We can see the trend and seasonality of the data above. Looking at the third-row plot, the data has a period

= (data_size) / (number_of_repeat) ≒ 225/19 ≒ 12.

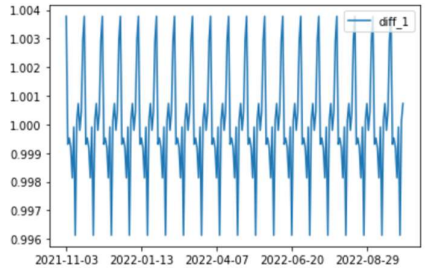
Then we use the **Dickey-Fuller test** to check its stationary and get the result:

```
Results of Dickey-Fuller Test
================================================
Test Statistic                   0.296756
p-value                          0.977195
#Lags Used                       0.000000
Number of Observations Used    224.000000
Criterical Value (1%)           -3.459885
Criterical Value (5%)           -2.874531
Criterical Value (10%)          -2.573694
dtype: float64
================================================
The data is non-stationary, so do differencing!
```

Since Test Statistic > Critical Value (10%), the data is non-stationary.

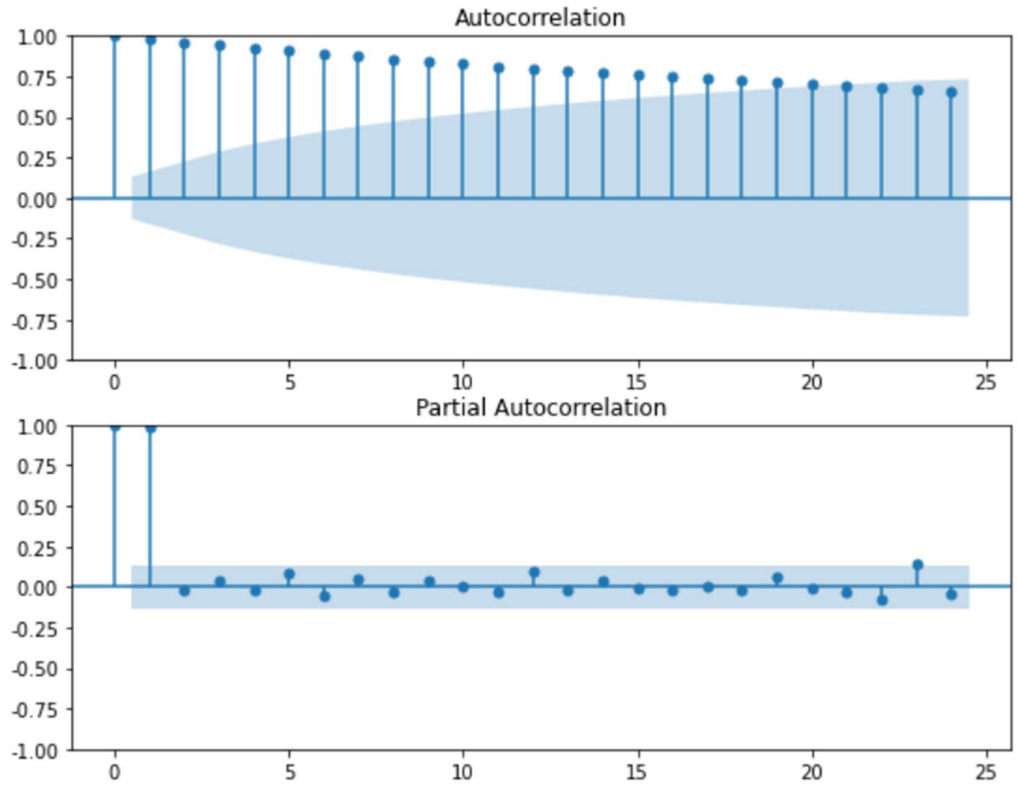
Next, we do the 1st-order differential on data and check its stationary again:

```
Results of Dickey-Fuller Test
================================================
Test Statistic                -25.283578
p-value                         0.000000
#Lags Used                      0.000000
Number of Observations Used   224.000000
Criterical Value (1%)          -3.459885
Criterical Value (5%)          -2.874531
Criterical Value (10%)         -2.573694
dtype: float64
================================================
The data is stationary. (Criterical Value 1%)
```




As above, the 1st-order differential on data becomes stationary and has a period = 12.

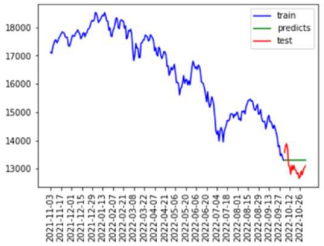
Then we plot the ACF(Autocorrelation Function)、PACF(Partial Autocorrelation Function) of data to check if it is appropriate to use AR, MA, AR(I)MA model for this data, which can help us on parameter choosing:




As above, from its PACF plot, combining what we have learned in class, we can see that an AR(I)MA model is the best.

- Build ARMIA model using auto_arima():

First, we refer to the example in the GitHub hyperlink but get the prediction to be a flat line, which means all 21 predictions after the last day in the train.csv are the same and equal to the last day's "close" value in the train.csv.



We think there must be some problem and start to tune other parameters in auto_arima().

After effort, we tune the parameters to be:

**model =pm.auto_arima(train_data,start_p=0,start_q=0, start_P=0, test='adf',D=1,trace=True ,seasonal=True,m=12 ,stepwise=False)**

We change start_p,start_q, and start_P from 1(default) to 0 to see if it may be better when they are zero.

We change the test method from ‘ocsb’ to ‘adf’ to see if it may be better.

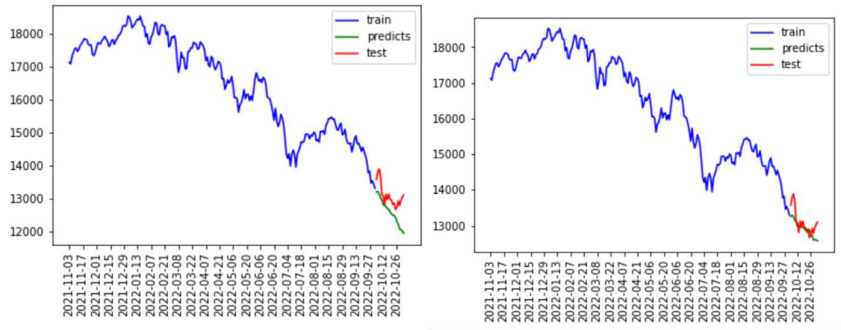
We change D from None to 1, since we have seen that the data has a seasonal differencing.

We set trace=True to see the status of the fits of each ARIMA model.

We set seasonal=True,m=12 since we have observed that the data has seasonality and had period = 12.

We set stepwise=False to make the algorithm not use the stepwise algorithm outlined in Hyndman and Khandakar (2008) to identify the optimal model parameters, which significantly improves the result.

stepwise=True stepwise=False




As above, we can see that the predicted curve fits more on the test curve if we set stepwise=False.

Using auto_arima(), we built an ARIMA model with parameters:

(p, d, q, P, D, Q, s) = (0, 1, 0, 0, 1, 1, 12)

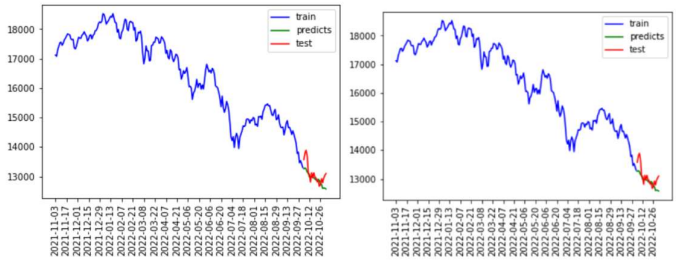
and get the MSE = 85245.55170136398.

It is not a good result but we have tried our best.

- .update() method in auto_arima():

Since we have seen an implementation method that split the training data into training part and validation part, and use .update() to update the ARIMA model. So we do an experiment to see if it really helps in our implementation. We split the last 20 data of training data as validation part.

without update: with update:




MSE= 85245.55170136398 MSE= 85477.65008679492

We can see that the result is almost the same, while “with update” even has a worse MSE. Hence, we guess that it isn’t necessary to use .update() in our implementation.

- Build ARMIA model using arima()

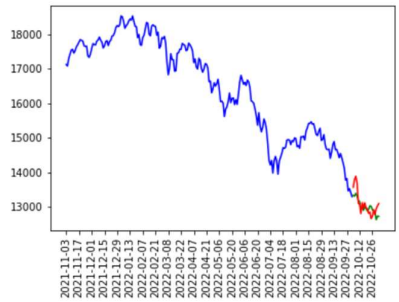
Method:

Since our goal is to minimize the MSE error of our forecasts, we use it as our cost function. What we do is **try every ARIMA model and see which has the minimum MSE forecasts**.

Since arima() function doesn’t have parameters as auto_arima(), first we only tune the parameters: **order,** and **sesonal_order** in arima().

We search for **p, P = 0~4, d, D=1~2, q, Q=0~4, and m=12** to see which parameter achieve the best performance.

Result:




model's table: (use model.summary())

SARIMAX Results

| | | | |
|---|---|---|---|
| **Dep. Variable:** | y | **No. Observations:** | 225 |
| **Model:** | SARIMAX(3, 1, 2)x(3, 2, [1], 12) | **Log Likelihood** | -1391.064 |
| **Date:** | Tue, 15 Nov 2022 | **AIC** | 2804.127 |
| **Time:** | 11:31:26 | **BIC** | 2840.409 |
| **Sample:** | 0 | **HQIC** | 2818.810 |
| | - 225 | | |
| **Covariance Type:** | opg | | |

| | coef | std err | z | P>\|z\| | [0.025 | 0.975] |
|---|---|---|---|---|---|---|
| **intercept** | 13.3558 | 13.739 | 0.972 | 0.331 | -13.572 | 40.283 |
| **ar.L1** | -1.3283 | 0.089 | -14.888 | 0.000 | -1.503 | -1.153 |
| **ar.L2** | -0.7899 | 0.126 | -6.260 | 0.000 | -1.037 | -0.543 |
| **ar.L3** | 0.1426 | 0.085 | 1.674 | 0.094 | -0.024 | 0.309 |
| **ma.L1** | 1.4734 | 0.606 | 2.430 | 0.015 | 0.285 | 2.662 |
| **ma.L2** | 0.9998 | 0.824 | 1.213 | 0.225 | -0.616 | 2.616 |
| **ar.S.L12** | -0.5230 | 0.080 | -6.515 | 0.000 | -0.680 | -0.366 |
| **ar.S.L24** | -0.4398 | 0.087 | -5.078 | 0.000 | -0.610 | -0.270 |
| **ar.S.L36** | -0.1991 | 0.080 | -2.491 | 0.013 | -0.356 | -0.042 |
| **ma.S.L12** | -0.9842 | 0.852 | -1.155 | 0.248 | -2.654 | 0.686 |
| **sigma2** | 5.238e+04 | 8.54e-05 | 6.13e+08 | 0.000 | 5.24e+04 | 5.24e+04 |

| | | | |
|---|---|---|---|
| **Ljung-Box (L1) (Q):** | 0.03 | **Jarque-Bera (JB):** | 3.78 |
| **Prob(Q):** | 0.87 | **Prob(JB):** | 0.15 |
| **Heteroskedasticity (H):** | 1.00 | **Skew:** | -0.22 |
| **Prob(H) (two-sided):** | 0.99 | **Kurtosis:** | 3.51 |

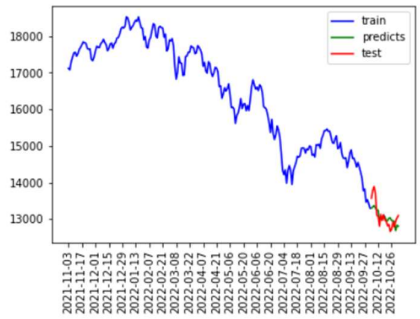
As above, we found that if we choose

(p, d, q, P, D, Q, s) = **(3, 1, 2, 3, 2, 1, 12)**,

we can achieve the best performance with MSE= **60895.79786192435**, which is better than auto_arima() but still isn't a good result.

Next, we try to tune the parameter: **out_of_sample_size** in arima(), which means the number of examples from the tail of the time series to hold out and use as validation examples. We tune out_of_sample_size=**20**(about 10% of training set) and do the experiment again.

Results:




model's table:

| | | | | | | |
|---|---|---|---|---|---|---|
| **Dep. Variable:** | | | y | **No. Observations:** | | 225 |
| **Model:** | SARIMAX(2, 2, 3)x(2, 1, [], 12) | | | **Log Likelihood** | | -1439.836 |
| **Date:** | Tue, 15 Nov 2022 | | | **AIC** | | 2897.671 |
| **Time:** | 12:55:42 | | | **BIC** | | 2927.838 |
| **Sample:** | 0 | | | **HQIC** | | 2909.865 |
| | - 225 | | | | | |
| **Covariance Type:** | opg | | | | | |

| | coef | std err | z | P>\|z\| | [0.025 | 0.975] |
|---|---|---|---|---|---|---|
| **intercept** | 1.4734 | 1.052 | 1.401 | 0.161 | -0.588 | 3.534 |
| **ar.L1** | -1.1086 | 0.079 | -13.999 | 0.000 | -1.264 | -0.953 |
| **ar.L2** | -0.9245 | 0.080 | -11.529 | 0.000 | -1.082 | -0.767 |
| **ma.L1** | 0.1846 | 5.556 | 0.033 | 0.973 | -10.704 | 11.073 |
| **ma.L2** | -0.2399 | 6.557 | -0.037 | 0.971 | -13.091 | 12.611 |
| **ma.L3** | -0.9446 | 5.236 | -0.180 | 0.857 | -11.207 | 9.318 |
| **ar.S.L12** | -0.5594 | 0.075 | -7.474 | 0.000 | -0.706 | -0.413 |
| **ar.S.L24** | -0.2757 | 0.092 | -3.005 | 0.003 | -0.455 | -0.096 |
| **sigma2** | 5.136e+04 | 2.84e+05 | 0.181 | 0.856 | -5.04e+05 | 6.07e+05 |

| | | | |
|---|---|---|---|
| **Ljung-Box (L1) (Q):** | 3.61 | **Jarque-Bera (JB):** | 2.62 |
| **Prob(Q):** | 0.06 | **Prob(JB):** | 0.27 |
| **Heteroskedasticity (H):** | 1.08 | **Skew:** | -0.19 |
| **Prob(H) (two-sided):** | 0.76 | **Kurtosis:** | 3.39 |

As above, we found that if we choose

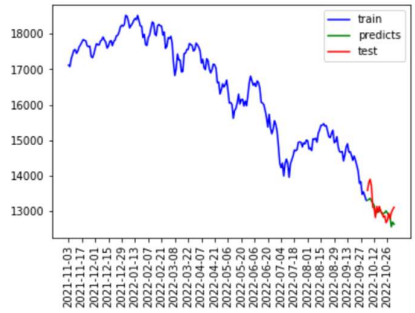
(p, d, q, P, D, Q, s) = (2, 2, 3, 2, 1, 0, 12),

we can achieve the best performance with MSE= 60449.52553029907, which has a little better result.

Moreover, we guess that the value of data may too "close", which has a minimum about 13000 and a maximum about 18000. Hence, we tried to do some linear data transformation before training. Since we want the value of data to has an order of 1 and the maximum is 2.5 times more than the minimum, we transform the data to be:

**New_data = (orginal_data/10000 - 0.966)*3**

Then we do the experiment again and get the results:




model's table:

SARIMAX Results

| | | | |
|---|---|---|---|
| **Dep. Variable:** | y | **No. Observations:** | 225 |
| **Model:** | SARIMAX(2, 1, 1)x(2, 2, [1, 2, 3], 12) | **Log Likelihood** | 240.236 |
| **Date:** | Tue, 15 Nov 2022 | **AIC** | -460.472 |
| **Time:** | 15:32:08 | **BIC** | -427.488 |
| **Sample:** | 0 | **HQIC** | -447.124 |
| | - 225 | | |
| **Covariance Type:** | opg | | |

| | coef | std err | z | P>\|z\| | [0.025 | 0.975] |
|---|---|---|---|---|---|---|
| **intercept** | 0.0008 | 0.001 | 0.671 | 0.503 | -0.002 | 0.003 |
| **ar.L1** | 0.4173 | 0.553 | 0.755 | 0.450 | -0.666 | 1.500 |
| **ar.L2** | -0.1416 | 0.090 | -1.579 | 0.114 | -0.317 | 0.034 |
| **ma.L1** | -0.2466 | 0.561 | -0.439 | 0.660 | -1.347 | 0.853 |
| **ar.S.L12** | -0.9119 | 1.776 | -0.514 | 0.608 | -4.392 | 2.569 |
| **ar.S.L24** | -0.0251 | 0.227 | -0.111 | 0.912 | -0.469 | 0.419 |
| **ma.S.L12** | -0.6187 | 1.789 | -0.346 | 0.730 | -4.126 | 2.889 |
| **ma.S.L24** | -0.7750 | 2.573 | -0.301 | 0.763 | -5.818 | 4.268 |
| **ma.S.L36** | 0.4617 | 0.862 | 0.536 | 0.592 | -1.228 | 2.151 |
| **sigma2** | 0.0036 | 0.001 | 5.968 | 0.000 | 0.002 | 0.005 |

| | | | |
|---|---|---|---|
| **Ljung-Box (L1) (Q):** | 0.28 | **Jarque-Bera (JB):** | 3.04 |
| **Prob(Q):** | 0.59 | **Prob(JB):** | 0.22 |
| **Heteroskedasticity (H):** | 1.41 | **Skew:** | -0.19 |
| **Prob(H) (two-sided):** | 0.16 | **Kurtosis:** | 3.47 |

As above, we found that if we choose

(p, d, q, P, D, Q, s) = (2, 1, 1, 2, 2, 3, 12),

after transforming the data back, we can achieve the best performance with MSE= 71914.75876944764, which isn't a better result.

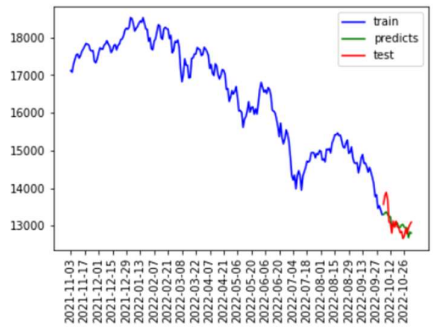
Hence, linear data transformation doesn't help with ARIMA model forecasting.

Conclusion:

After the above experiments, we found that if we choose

**(p, d, q, P, D, Q, s) = (2, 2, 3, 2, 1, 0, 12)**

, set stepwise=False, and directly fit the training data on the model, we can the result:




With minimum MSE = 60449.52553029907.

Reference:

- Lecture slide
- Reference in “Data Science HW3.pptx”

Problem2:

- https://journals.sagepub.com/doi/full/10.1155/2015/806954#fig2-2015-806954
- https://niapy.org/en/stable/tutorials/feature_selection.html
- https://ithelp.ithome.com.tw/articles/10231724?sc=pt

Problem3:

- https://adaptable-haze-butterfly-551.medium.com/arima%E6%99%82%E9%96%93%E5%BA%8F%E5%88%97%E6%A8%A1%E5%9E%8Bpython%E6%87%89%E7%94%A8-%E9%8A%85%E5%83%B9%E6%A0%BC%E9%A0%90%E6%B8%AC-%E4%B8%80-4f91693e3ec6

- https://towardsdatascience.com/time-series-forecasting-predicting-stock-prices-using-an-arima-model-2e3b3080bd70