02_Model

October 6, 2020

1 flats-in-cracow machine learning

1.1 Imports

[1]:
```
from datetime import datetime
from distutils.dir_util import copy_tree
from pathlib import Path

import joblib
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

from matplotlib.ticker import MaxNLocator
from pylab import rcParams
from sklearn.compose import ColumnTransformer, TransformedTargetRegressor
from sklearn.dummy import DummyRegressor
from sklearn.ensemble import (GradientBoostingRegressor, RandomForestRegressor,
                              VotingRegressor)
from sklearn.impute import KNNImputer
from sklearn.metrics import (mean_absolute_error, mean_squared_error,
                             mean_squared_log_error)
from sklearn.model_selection import GridSearchCV, train_test_split, KFold
from sklearn.neural_network import MLPRegressor
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import MinMaxScaler, OneHotEncoder
```

1.2 Setup

[2]:
```
# Create directory for images
Path("img").mkdir(parents=True, exist_ok=True)

# Set default figure size
rcParams['figure.figsize'] = (4, 4)

# Tell pandas how to display floats
pd.options.display.float_format = "{:,.2f}".format
```

1.3 Data loading

[3]:
```
path = '../flats-data/cleaned_data.csv'
```

[4]:
```
data = pd.read_csv(path, lineterminator='\n')
```

[5]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4592 entries, 0 to 4591
Data columns (total 17 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   District   4592 non-null   object
 1   Amount     4592 non-null   int64
 2   Seller     4592 non-null   object
 3   Area       4592 non-null   int64
 4   Rooms      4592 non-null   int64
 5   Bathrooms  4592 non-null   int64
 6   Parking    4592 non-null   object
 7   Garden     4592 non-null   bool
 8   Balcony    4592 non-null   bool
 9   Terrace    4592 non-null   bool
 10  Floor      4592 non-null   bool
 11  New        4592 non-null   bool
 12  Estate     4592 non-null   bool
 13  Townhouse  4592 non-null   bool
 14  Apartment  4592 non-null   bool
 15  Land       4592 non-null   bool
 16  Studio     4592 non-null   bool
dtypes: bool(10), int64(4), object(3)
memory usage: 296.1+ KB
```

[6]:
```
data.head()
```

[6]:

	District	Amount	Seller	Area	Rooms	Bathrooms	Parking	Garden
0	krowodrza	595000	realtor	78	4	2	no parking	False
1	podgorze	449000	realtor	61	3	1	no parking	False
2	nowa huta	449000	realtor	58	3	1	no parking	False
3	krowodrza	595000	realtor	78	4	2	no parking	False
4	krowodrza	430000	realtor	48	2	1	garage	False

	Balcony	Terrace	Floor	New	Estate	Townhouse	Apartment	Land	Studio
0	True	False	False	False	False	False	False	False	False
1	True	False	True	False	False	False	False	False	False
2	True	False	False	True	False	False	False	False	False
3	True	False	False	False	False	False	False	False	False

```
4      True     False     True   False     True        False        False   False     False
```

1.4 Feature engineering

The next step is to engineer features. We add columns describing the `Total Rooms` in the property, ratio of `Area to Rooms` and so on.

[7]:
```python
data['Log Area'] = np.round(np.log(data['Area']), 2)
data['Bool Sum'] = data.select_dtypes(bool).sum(axis=1)
# Avoid division by zero
data['Area to Bool Sum'] = round(data['Area'] / (data.select_dtypes(bool).
 ↪sum(axis=1) + 1), 2)
data['Rooms to Bool Sum'] = round(data['Rooms'] / (data.select_dtypes(bool).
 ↪sum(axis=1) + 1), 2)
data['Rooms to Bathrooms' ] = round(data['Rooms'] / data['Bathrooms'], 2)
data['Total Rooms'] = round(data['Rooms'] + data['Bathrooms'], 2)
data['Area to Rooms'] = round(data['Area'] / data['Rooms'], 2)
data['Area to Bathrooms'] = round(data['Area'] / data['Rooms'], 2)
data['Area to Total Rooms'] = round(data['Area'] / data['Total Rooms'], 2)
```

1.5 Data split

We decide to use 80% of the data to train the model and 20% to check performance. We make sure to remove the `Amount` column from the training data since this is our target and remove duplicates before training.

[8]:
```python
print(len(data))
data = data.drop_duplicates()
print(len(data))
```

```
4592
3998
```

[9]:
```python
X = data.drop(['Amount'], axis=1)
y = data['Amount']

split = train_test_split(X, y, train_size=.8,
                         random_state=123)

X_train, X_test, y_train, y_test = split
```

1.6 Models

Next step is to create the models and associated piplines. We apply one hot encoding to categorical features and use the `ColumnTransformer` parameter `passthrough` to allow the rest of the columns to remain unchanged.

[10]:
```
categorical = list(X.select_dtypes('object').columns)
continuous = list(X.select_dtypes('int64'))
continuous += list(X.select_dtypes('float64'))
```

1.6.1 Baseline model

For comparison purposes we create a model to give base predictions.

[11]:
```
dmr = DummyRegressor()

dmr_ohe = Pipeline(
    steps=[('onehot', OneHotEncoder(handle_unknown='ignore'))]
)

dmr = Pipeline(steps = [('preprocessor', dmr_ohe),
                        ('regressor', dmr)])

# dmr.fit(X_train, y_train)
```

1.6.2 Multi-layer Perceptron

For the neural network we apply the `MinMaxScaler` so that the continuous columns have values in $[0, 1]$ and then we apply `OneHotEncoder` to the categorical columns.

[12]:
```
mlp = MLPRegressor(hidden_layer_sizes=(100, 100, 100),
                   max_iter=2*10**4,
                   random_state=123)

mlp_ohe = Pipeline(
    steps=[('onehot', OneHotEncoder(handle_unknown='ignore'))]
)

mlp_scale = Pipeline(
    steps=[('scale', MinMaxScaler())]
)

mlp_pre = ColumnTransformer(
    transformers = [
        ('scale', mlp_scale, continuous),
        ('cat', mlp_ohe, categorical),
    ],
    remainder='passthrough'
)

mlp_trans = TransformedTargetRegressor(regressor=mlp,
                                       transformer=MinMaxScaler())
```

```
mlp = Pipeline(steps = [('preprocessor', mlp_pre),
                        ('transformer', mlp_trans)])

# mlp.fit(X_train, y_train)
```

1.6.3 Gradient Boosting Regressor

For the gradient booster we only apply `OneHotEncoder` to the categorical columns.

[13]:
```
gbr = GradientBoostingRegressor(random_state=123)

gbr_ohe = Pipeline(
    steps=[('onehot', OneHotEncoder(handle_unknown='ignore'))]
)

gbr_pre = ColumnTransformer(
    transformers = [
        ('cat', gbr_ohe, categorical)
    ],
    remainder='passthrough'
)

gbr = Pipeline(steps = [('preprocessor', gbr_pre),
                        ('regressor', gbr)])

# gbr.fit(X_train, y_train)
```

1.7 Parameter tuning

We set up the training process to conduct basic parameter tuning and cross validation.

[14]:
```
kf = KFold(n_splits=5, random_state=123, shuffle=True)
```

[15]:
```
gbr_grid = {'regressor__max_depth': [5, 10, 15],
            'regressor__n_estimators': [50, 100, 200, 300],
            'regressor__min_samples_split': [2, 4],
            'regressor__min_samples_leaf': [2, 4],
            'regressor__max_features': ['auto']}

gbr_gs = GridSearchCV(estimator=gbr,
                      param_grid=gbr_grid,
                      cv=kf,
                      n_jobs=8,
                      scoring='neg_root_mean_squared_error',
                      verbose=2)
```

[16]:
```
layers = [(100, 100, 100),
          (150, 200, 150),
          (200, 400, 200)]

mlp_grid = {'transformer__regressor__activation': ['relu'],
            'transformer__regressor__solver': ['adam'],
            'transformer__regressor__learning_rate': ['adaptive'],
            'transformer__regressor__learning_rate_init': [0.01, 0.001, 0.0001],
            'transformer__regressor__hidden_layer_sizes': layers}

mlp_gs = GridSearchCV(estimator=mlp,
                      param_grid=mlp_grid,
                      cv=kf,
                      n_jobs=8,
                      scoring='neg_root_mean_squared_error',
                      verbose=2)
```

1.8 Training

[17]:
```
dmr.fit(X_train, y_train)
```

[17]:
```
Pipeline(steps=[('preprocessor',
                 Pipeline(steps=[('onehot',
                                  OneHotEncoder(handle_unknown='ignore'))])),
                ('regressor', DummyRegressor())])
```

[18]:
```
mlp = mlp_gs.fit(X_train, y_train).best_estimator_
mlp
```

```
Fitting 5 folds for each of 9 candidates, totalling 45 fits
```

```
[Parallel(n_jobs=8)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=8)]: Done  25 tasks      | elapsed:   25.5s
[Parallel(n_jobs=8)]: Done  45 out of  45 | elapsed:  1.2min finished
```

[18]:
```
Pipeline(steps=[('preprocessor',
                 ColumnTransformer(remainder='passthrough',
                                   transformers=[('scale',
                                                  Pipeline(steps=[('scale',
MinMaxScaler())]),
                                                  ['Area', 'Rooms', 'Bathrooms',
                                                   'Bool Sum', 'Total Rooms',
                                                   'Log Area',
                                                   'Area to Bool Sum',
                                                   'Rooms to Bool Sum',
                                                   'Rooms to Bathrooms',
                                                   'Area to Rooms',
```

```
                                                    'Area to Bathrooms',
                                                    'Area to Total Rooms']),
                                                  ('cat',
                                                   Pipeline(steps=[('onehot',
OneHotEncoder(handle_unknown='ignore'))]),
                                                   ['District', 'Seller',
                                                    'Parking'])])),
                ('transformer',
TransformedTargetRegressor(regressor=MLPRegressor(hidden_layer_sizes=(200,
       400,
       200),
learning_rate='adaptive',
learning_rate_init=0.0001,
max_iter=20000,
random_state=123),
                                            transformer=MinMaxScaler()))])
```

CV RMSE score for MLPRegressor:

[19]:
```
print(round(abs(mlp_gs.best_score_)))
```

```
119689
```

[20]:
```
gbr = gbr_gs.fit(X_train, y_train).best_estimator_
gbr
```

```
Fitting 5 folds for each of 48 candidates, totalling 240 fits
```

```
[Parallel(n_jobs=8)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=8)]: Done  25 tasks      | elapsed:    9.7s
[Parallel(n_jobs=8)]: Done 146 tasks      | elapsed:  1.4min
[Parallel(n_jobs=8)]: Done 240 out of 240 | elapsed:  3.5min finished
```

[20]:
```
Pipeline(steps=[('preprocessor',
                 ColumnTransformer(remainder='passthrough',
                                   transformers=[('cat',
                                                  Pipeline(steps=[('onehot',
OneHotEncoder(handle_unknown='ignore'))]),
                                                   ['District', 'Seller',
                                                    'Parking'])])),
                ('regressor',
                 GradientBoostingRegressor(max_depth=5, max_features='auto',
                                           min_samples_leaf=4,
                                           random_state=123))])
```

CV RMSE score for GradientBoostingRegressor:

[21]:
```
print(round(abs(gbr_gs.best_score_)))
```

117912

1.8.1 Voting Regressor

We create a `VotingRegressor` with uniform weights to be able to combine predictions of our models.

[22]:
```
vote = VotingRegressor(estimators=[['mlp', mlp], ['gbr', gbr]], n_jobs=8)
vote = vote.fit(X_train, y_train)
```

1.9 Model performance

We obtain predictions for the testing set and compare `RMSE`, `MAE` and `MSLE` scores of our models.

[23]:
```
def get_scores(regressor, X_test, y_true, verb=True):
    """
    Obtain RMSE, MAE and MSLE for test set.
    """

    y_pred = regressor.predict(X_test)

    rmse = mean_squared_error(y_pred=y_pred,
                              y_true=y_test,
                              squared=False)

    mae = mean_absolute_error(y_pred=y_pred,
                              y_true=y_test)

    msle = mean_squared_log_error(y_pred=y_pred,
                                  y_true=y_test)

    if verb:

        print(f'RMSE: {rmse:10.2f}')
        print(f'MAE:  {mae:10.2f}')
        print(f'MSLE: {msle:10.2f}')

    return (rmse, mae, msle)
```

1.9.1 Dummy

[24]:
```
dmr_score = get_scores(regressor=dmr,
                       X_test=X_test,
                       y_true=y_test)
```

```
RMSE:  222475.70
MAE:   161219.86
MSLE:       0.13
```

1.9.2 Multilayer Perceptrion

[25]:
```
mlp_score = get_scores(regressor=mlp,
                       X_test=X_test,
                       y_true=y_test)
```

```
RMSE:  120493.86
MAE:    79646.51
MSLE:       0.03
```

1.9.3 Gradient Boosting Regressor

[26]:
```
gbr_score = get_scores(regressor=gbr,
                       X_test=X_test,
                       y_true=y_test)
```

```
RMSE:  119237.05
MAE:    76182.62
MSLE:       0.03
```

1.9.4 Voting Regressor

[27]:
```
vote_score = get_scores(regressor=vote,
                        X_test=X_test,
                        y_true=y_test)
```

```
RMSE:  116478.02
MAE:    74993.84
MSLE:       0.03
```

1.9.5 Comparison

We are happy to see that the `VotingRegressor` outperforms the `DummyRegressor` model as well the `GradientBoostingRegressor` and the `MLPRegressor`.

[28]:
```
scores = [dmr_score,
          mlp_score,
          gbr_score,
          vote_score]

scores = pd.DataFrame(scores,
                      index=['DMR', 'MLP', 'GBR', 'VOTE'],
                      columns=['RMSE', 'MAE', 'MSLE'])

scores
```

[28]:

	RMSE	MAE	MSLE
DMR	222,475.70	161,219.86	0.13

```
MLP  120,493.86  79,646.51  0.03
GBR  119,237.05  76,182.62  0.03
VOTE 116,478.02  74,993.84  0.03
```

1.10 Visualizations

We produce a couple of plots the visually inspect the performance of our model. We use the test data set with the predicted `Amount` to produce the plots.

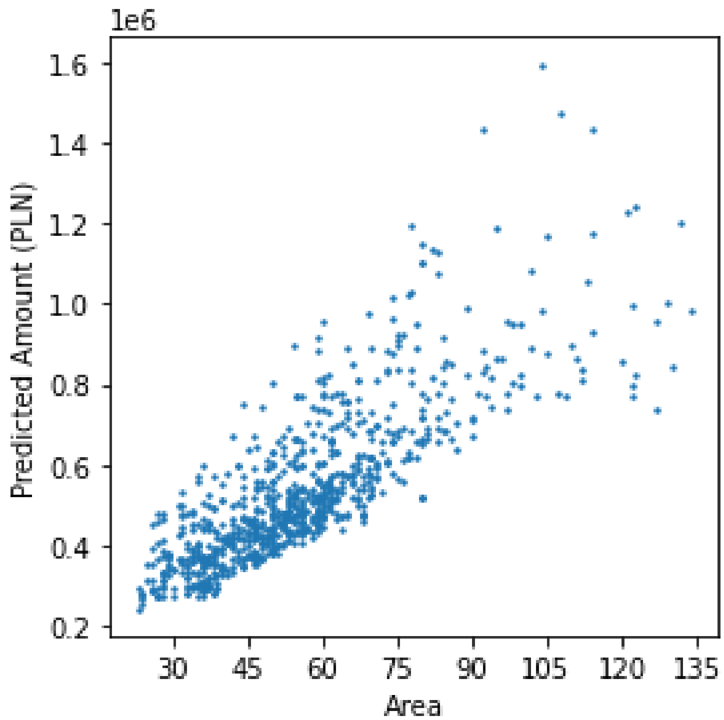
[29]:
```
cols = ['Amount', 'Predicted Amount',
        'District', 'Area', 'Total Rooms']

X_pred = X_test.copy()
X_pred.loc[:, 'Amount'] = y
X_pred.loc[:, 'Predicted Amount'] = vote.predict(X_test)
X_pred = X_pred.loc[:, cols]

X_pred.head()
```

[29]:

	Amount	Predicted Amount	District	Area	Total Rooms
1241	660000	657,844.33	podgorze	79	5
3535	600000	607,834.91	lagiewniki	67	4
2449	339000	342,399.17	bronowice	30	2
2992	528000	531,523.50	czyzyny	55	4
2756	850000	776,874.86	stare miasto	63	3

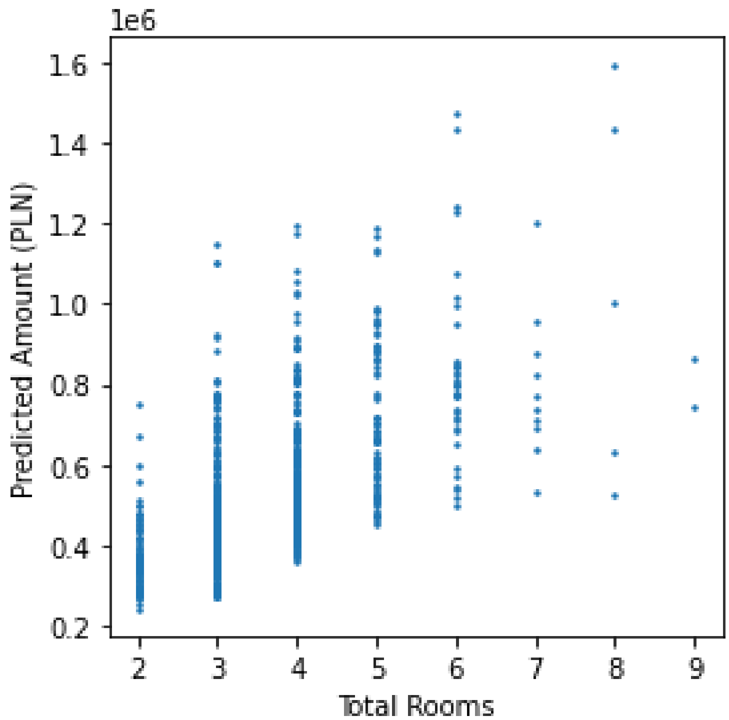
On our first visual it can be seen that there exists a fairly linear relationship between the `Predicted Amount` and the `Area` of the property.

[30]:
```
plt.scatter(X_pred['Area'], X_pred['Predicted Amount'], s=2)
plt.xlabel('Area')
plt.ylabel('Predicted Amount (PLN)')

ax = plt.gca()
ax.xaxis.set_major_locator(MaxNLocator(integer=True))

plt.tight_layout()
plt.savefig('img/area_vs_amount.png')
plt.show()
```



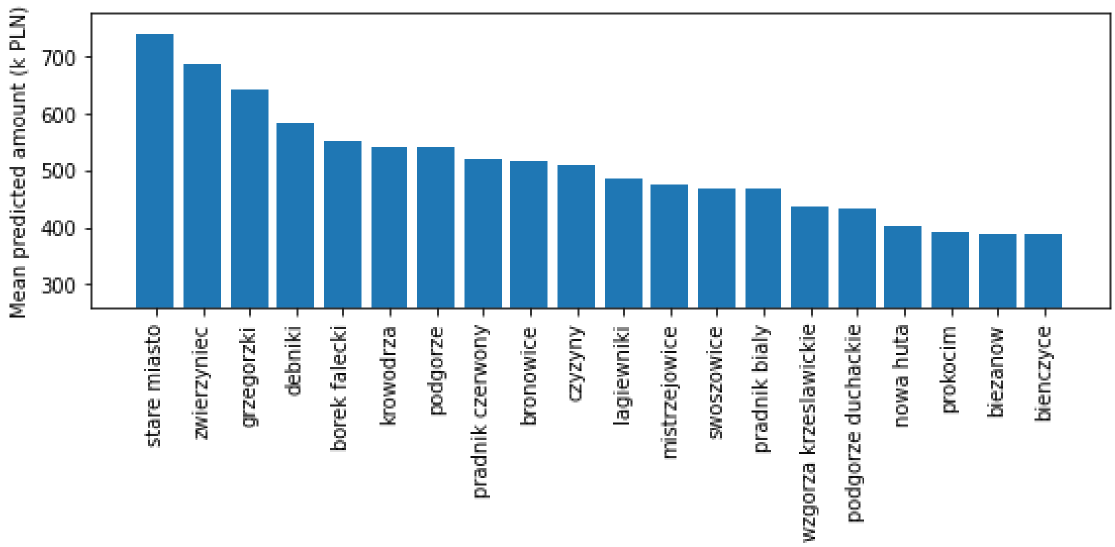
On the second visual it can bee seen, as expected the more `Total Rooms` in a `Property` the more it should cost.

[31]:
```
plt.scatter(X_pred['Total Rooms'], X_pred['Predicted Amount'], s=2)
plt.xlabel('Total Rooms')
plt.ylabel('Predicted Amount (PLN)')

ax = plt.gca()
ax.xaxis.set_major_locator(MaxNLocator(integer=True))

plt.tight_layout()
plt.savefig('img/rooms_vs_amount.png')
plt.show()
```



Next we want to check if the model distinguishes between districts. We group the data by `District` and calculate the mean of the predictions with the group. We produce a bar chart sorted from highest average to lowest. Clearly the model distinguishes between district that are near the city center (`stare miasto`, `zwierzyniec`) and those further away (`łagiewniki`, `bieżanów`).

[32]:
```
width = 1600
height = width/2
dpi = 200

X_grp = X_pred[['District', 'Predicted Amount']]
X_grp = X_grp.groupby('District', as_index=False).mean()
X_grp = X_grp.sort_values('Predicted Amount', ascending=False)

plt.figure(figsize=(width/dpi, height/dpi))

plt.bar(X_grp['District'], X_grp['Predicted Amount'] / 1000)

plt.ylabel('Mean predicted amount (k PLN)')
plt.ylim(X_grp['Predicted Amount'].min() * 0.67 / 1000, None)
plt.xticks(rotation=90)

plt.tight_layout()
plt.savefig('img/district_vs_avg_amount.png')
plt.show()
```



1.11 Getting predictions

Next we would like see how the model handles sets of arbitrary parameters. We write a function to transform inputs to desired format and obtain prediction from the model.

[33]:
```
def get_pred(district,
             seller,
             area,
             rooms,
             bathrooms,
             parking,
             garden,
             balcony,
             terrace,
             floor,
             new,
             estate,
             townhouse,
             apartment,
             land,
             studio):

    columns = ['District',
               'Seller',
               'Area',
               'Rooms',
               'Bathrooms',
               'Parking',
               'Garden',
```

```python
            'Balcony',
            'Terrace',
            'Floor',
            'New',
            'Estate',
            'Townhouse',
            'Apartment',
            'Land',
            'Studio',
            'Log Area',
            'Bool Sum',
            'Area to Bool Sum',
            'Rooms to Bool Sum',
            'Rooms to Bathrooms',
            'Total Rooms',
            'Area to Rooms',
            'Area to Bathrooms',
            'Area to Total Rooms']

log_area = np.log(area)

all_bools = [garden,
             balcony,
             terrace,
             floor,
             new,
             estate,
             townhouse,
             apartment,
             land,
             studio]

bool_sum = sum(all_bools)
area_to_bool_sum = area / (bool_sum + 1)
rooms_to_bool_sum = rooms / (bool_sum + 1)
rooms_to_bathrooms = rooms / bathrooms
total_rooms = rooms + bathrooms
area_to_rooms = area / total_rooms
area_to_bathrooms = area / bathrooms
area_to_total_rooms = area / total_rooms

x = [district,
     seller,
     area,
     rooms,
     bathrooms,
     parking,
```

```
        garden,
        balcony,
        terrace,
        floor,
        new,
        estate,
        townhouse,
        apartment,
        land,
        studio,
        log_area,
        bool_sum,
        area_to_bool_sum,
        rooms_to_bool_sum,
        rooms_to_bathrooms,
        total_rooms,
        area_to_rooms,
        area_to_bathrooms,
        area_to_total_rooms]

    x = pd.DataFrame([x], columns=columns)
    x = float(vote.predict(x))

    return int(round(x, -3))
```

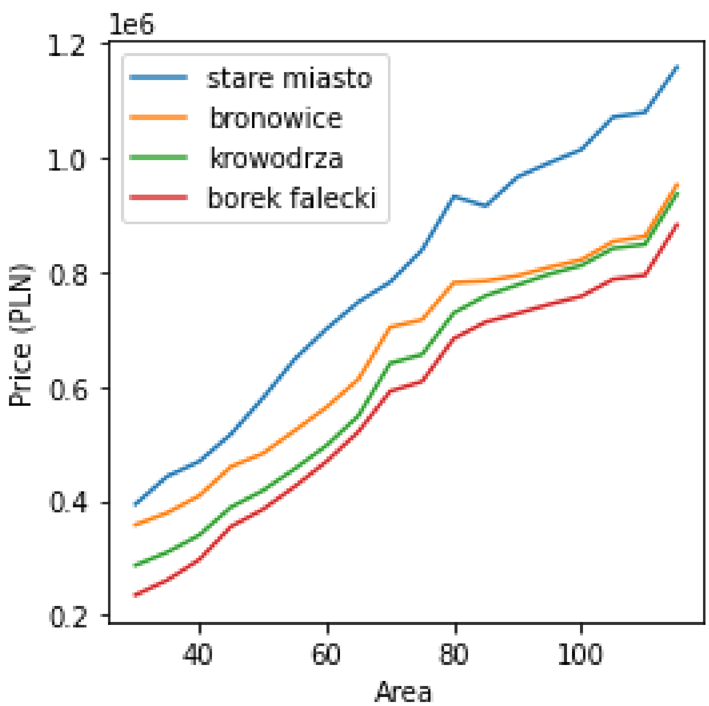
We create lists of inputs for the model to predict.

[34]:
```
areas = range(30, 120, 5)
rooms = range(1, 5)
districts = ['stare miasto',
             'bronowice',
             'krowodrza',
             'borek falecki']
```

Next we loop over lists of possible `Area`'s and `Room`'s and plot the outputs. First we check how the model reacts to different districts.

[35]:
```
plt.figure()

for d in districts:
    value = list()
    for a in areas:
        pred = get_pred(district=d,
         seller='realtor',
         area=a,
         rooms=2,
         bathrooms=1,
         parking='street',
```

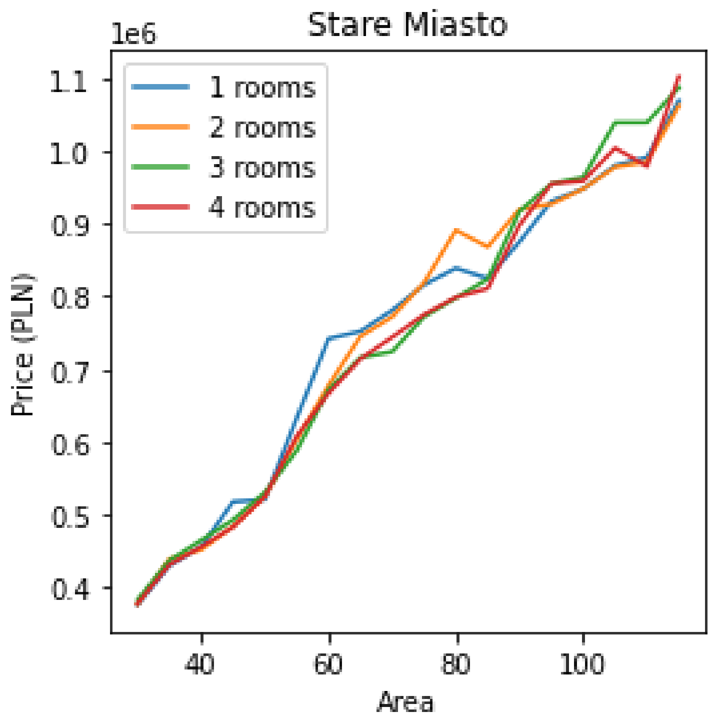
```
            garden=False,
            balcony=False,
            terrace=False,
            floor=False,
            new=True,
            estate=False,
            townhouse=True,
            apartment=False,
            land=False,
            studio=True)
        value.append(pred)
    plt.plot(areas, value, label=d)

plt.ylabel('Price (PLN)')
plt.xlabel('Area')
plt.legend(loc='best')
plt.savefig('img/area_vs_amount_by_district')
plt.show()
```



We do the same for different amounts of `Room`'s.

[36]:
```
plt.figure()

for r in rooms:
```

```python
    value = list()
    for a in areas:
        pred = get_pred(district='stare miasto',
         seller='owner',
         area=a,
         rooms=r,
         bathrooms=1,
         parking='street',
         garden=False,
         balcony=True,
         terrace=False,
         floor=False,
         new=True,
         estate=False,
         townhouse=True,
         apartment=False,
         land=False,
         studio=True)
        value.append(pred)
    plt.plot(areas, value, label=f'{r} rooms')

plt.title('Stare Miasto')
plt.ylabel('Price (PLN)')
plt.xlabel('Area')
plt.legend(loc='best')
plt.savefig('img/area_vs_amount_by_rooms')
plt.show()
```



1.12 Final training

The last step is to fit the model to the entire dataset and save it for later use.

[37]:
```
start = datetime.now()

gbr.fit(X, y)
joblib.dump(gbr, f'../flats-model/gbr.joblib')

mlp.fit(X, y)
joblib.dump(mlp, f'../flats-model/mlp.joblib')

vote.fit(X, y)
joblib.dump(vote, f'../flats-model/vote.joblib')

end = datetime.now()

duration = (end - start).seconds

print(f'Full training took {int(duration)} seconds.')
```

```
Full training took 648 seconds.
```

[38]:

```
# Copy files to portfolio
# fromDirectory = '.'
# toDirectory = '/home/dev/Github/data-science-portfolio/flats-in-cracow'
# copy_tree(fromDirectory, toDirectory)
```