template

December 10, 2023

1 Your Title Here

Name(s): (your name(s) here)

Website Link: (your website link)

1.1 Code

[ ]:
```
import pandas as pd
import numpy as np
import os

import plotly.express as px
pd.options.plotting.backend = 'plotly'
```

1.1.1 Framing the Problem

We use the data of Power Outage. Much like the steps in Project3, we samely read the data and clean it using the same way. But some slightly changes will be added.

Different from the Project 3, we add predealing process, removing all the rows if the outage is missing. Below shows the steps.

We note that the datetime data type isn't suitable for modeling and later works, so we'd convert it into numbers. We'll perform it later in the pipeline using `FunctionTransformer`.

[ ]:
```
# skip first 5 rows; or use an online URL as param instead
data = pd.read_excel("outage.xlsx", header=5)
# drop the 7th row
data = data.drop(0)
# drop the 1st column
data = data.drop(data.columns[0], axis=1)
# show the data
print(data)

# clone the raw data
df = data.copy()
# data type convertion
df['DEMAND.LOSS.MW'] = df['DEMAND.LOSS.MW'].fillna(0)
```

	OBS	YEAR	MONTH	U.S._STATE	POSTAL.CODE	NERC.REGION	\
1	1.0	2011.0	7.0	Minnesota	MN	MRO	
2	2.0	2014.0	5.0	Minnesota	MN	MRO	
3	3.0	2010.0	10.0	Minnesota	MN	MRO	
4	4.0	2012.0	6.0	Minnesota	MN	MRO	
5	5.0	2015.0	7.0	Minnesota	MN	MRO	
…	…	…	…	…	…	…	
1530	1530.0	2011.0	12.0	North Dakota	ND	MRO	
1531	1531.0	2006.0	NaN	North Dakota	ND	MRO	
1532	1532.0	2009.0	8.0	South Dakota	SD	RFC	
1533	1533.0	2009.0	8.0	South Dakota	SD	MRO	
1534	1534.0	2000.0	NaN	Alaska	AK	ASCC	

	CLIMATE.REGION	ANOMALY.LEVEL	CLIMATE.CATEGORY	OUTAGE.START.DATE	\
1	East North Central	-0.3	normal	2011-07-01 00:00:00	
2	East North Central	-0.1	normal	2014-05-11 00:00:00	
3	East North Central	-1.5	cold	2010-10-26 00:00:00	
4	East North Central	-0.1	normal	2012-06-19 00:00:00	
5	East North Central	1.2	warm	2015-07-18 00:00:00	
…	…	…	…	…	
1530	West North Central	-0.9	cold	2011-12-06 00:00:00	
1531	West North Central	NaN	NaN	NaN	
1532	West North Central	0.5	warm	2009-08-29 00:00:00	
1533	West North Central	0.5	warm	2009-08-29 00:00:00	
1534	NaN	NaN	NaN	NaN	

	…	POPPCT_URBAN	POPPCT_UC	POPDEN_URBAN	POPDEN_UC	POPDEN_RURAL	\
1	…	73.27	15.28	2279	1700.5	18.2	
2	…	73.27	15.28	2279	1700.5	18.2	
3	…	73.27	15.28	2279	1700.5	18.2	
4	…	73.27	15.28	2279	1700.5	18.2	
5	…	73.27	15.28	2279	1700.5	18.2	
…	…	…	…	…	…	…	
1530	…	59.9	19.9	2192.2	1868.2	3.9	
1531	…	59.9	19.9	2192.2	1868.2	3.9	
1532	…	56.65	26.73	2038.3	1905.4	4.7	
1533	…	56.65	26.73	2038.3	1905.4	4.7	
1534	…	66.02	21.56	1802.6	1276	0.4	

	AREAPCT_URBAN	AREAPCT_UC	PCT_LAND	PCT_WATER_TOT	PCT_WATER_INLAND
1	2.14	0.6	91.592666	8.407334	5.478743
2	2.14	0.6	91.592666	8.407334	5.478743
3	2.14	0.6	91.592666	8.407334	5.478743
4	2.14	0.6	91.592666	8.407334	5.478743
5	2.14	0.6	91.592666	8.407334	5.478743
…	…	…	…	…	…
1530	0.27	0.1	97.599649	2.401765	2.401765
1531	0.27	0.1	97.599649	2.401765	2.401765

```
1532              0.3         0.15   98.307744        1.692256           1.692256
1533              0.3         0.15   98.307744        1.692256           1.692256
1534             0.05         0.02   85.761154       14.238846           2.901182

[1534 rows x 56 columns]
```

The prediction problem we'd focus on is predicting the severity of a major power outage.

There may be many columns which can measure the severity, such as number of affected customers, duration, or demand loss. Here, we use the number of affected customers as the only measurement. That is to say, the number of affected customers in an outage is our prediction target.

The reason we use outage rather than other columns(like the number of customers, demand loss, etc.) is that:

1. Choosing "number of customers affected" as the primary factor for predicting power outage severity is effective because it directly reflects the impact's extent and is a clear indicator of socio-economic effects. This measure is typically more reliable and accessible than others.
2. Choosing other factors like "duration" or "demand loss" might not always proportionately reflect the outage's severity and could complicate the model.
3. Also, there's too many missing values of the `DEMAND.LOSS.MW`, which makes it difficult to use.
4. Additionally, integrating multiple factors could increase complexity and risk of collinearity, detracting from the model's manageability and predictive accuracy.

```
[ ]:  # check there's too many missing values of DEMAND.LOSS.MW
print(data['DEMAND.LOSS.MW'].isna().sum())
df['DEMAND.LOSS.MW'] = df['DEMAND.LOSS.MW'].fillna(0)
```

```
705
```

```
[ ]: df = df.dropna(subset=['OUTAGE.DURATION', 'CUSTOMERS.AFFECTED', 'TOTAL.PRICE'])
```

Below we check the result after cleaning the missing values.

```
[ ]: print(df.shape[0]) # origin is 1534
print(df.iloc[:5][['OUTAGE.DURATION', 'CUSTOMERS.AFFECTED']])
```

```
1047
  OUTAGE.DURATION  CUSTOMERS.AFFECTED
1            3060             70000.0
3            3000             70000.0
4            2550             68200.0
5            1740            250000.0
6            1860             60000.0
```

We use the formula $severity = \log_2(number\_of\_customers + 1)$ to measure the severity by experience. The reason for the transformation is that, by observing the data, we found that there're a large difference of the order of magnitude, if we directly use the `CUSTOMERS.AFFECTED` feature, it's both hard to measure and train the model, since in large numbers, any "slight" difference will be great.

```
[ ]: MEASUREMENT_COLUMN = 'SEVERITY'
df[MEASUREMENT_COLUMN] = np.log(df['CUSTOMERS.AFFECTED'] + 1)
```

Clearly, it's a regression model. The response variable which the model is going to predict is the logarithmic value of `CUSTOMERS.AFFECTED`, the number of people affected by the outage, which can roughly measure the severity of an outage.

We use R^2 as metric to measure our model. The reasons are that:

1. Since it's not a classification model, so we won't use classification metrics like precision or recall.
2. The two metrics RMSE and R^2 are classic for regression model. But we only need one of them to determine which model better. So we compare them as below:
3. RMSE is preferred when the absolute size of errors is crucial, as it directly reflects the average difference between the predicted and actual values and is more sensitive to larger errors.
4. R^2 is better suited for assessing a model's explanatory power, as it measures how well the model explains the variability of the target variable, and is useful in standardized performance evaluation across different datasets.
5. We consider the explanatory power and standardized performance more important in our problem, so we use the R^2.

Below we define a function to calculate the metric value and compare them.

```
[ ]: from sklearn.metrics import r2_score
def get_R2(y_real, y_pred):
    return r2_score(y_real, y_pred)
def measure(y_real, y_pred):
    return get_R2(y_real, y_pred)
```

For convenience, we write a helper function to export the `plotly` figure into HTML file.

```
[ ]: def export_plotly_fig(fig, filename):
    fig.write_html(filename, include_plotlyjs='cdn')
```

1.1.2 Baseline Model

We try analyzed many features manually(due to space constraints, the process is omitted here), i.e. `U.S._STATE`, `POSTAL.CODE`, `CLIMATE.REGION`, `ANOMALY.LEVEL`, `OUTAGE.START.DATE`, `OUTAGE.START.TIME`, `OUTAGE.RESTORATION.DATE`, `OUTAGE.RESTORATION.TIME`, `TOTAL.PRICE`, `TOTAL.SALES`, `TOTAL.CUSTOMERS`, `POPULATION`, `POPDEN_URBAN`, we try using them singularly and together, but little effect is found. So we think them as irrelevant features. But fortunately we try out a crucial feature `CAUSE.CATEGORY`.

Consequently, we adopt the single feature `CAUSE.CATEGORY` for baseline model and state the possible reasons why this feature is useful.

More features and fine adjustments will be added later in the final model.

We adopt the classic scheme that using 75% of the data as training set, 25% of the data as validation set.

We've try several different classic model(due to space constraints, the process is omitted here), and we figure out that the `LinearRegression`, `KNeighborsRegressor` and `SVR` models cannot work well. While the `DecisionTreeRegressor` and `RandomForestRegressor` works well.

We adopt the classic `DecisionTreeRegressor` as baseline model, and we will try compare it later with `RandomForestRegressor` and choose the best one as the final model.

Below, we show the detailed steps of implementations.

To begin with, we divide the data using the codes below. It will randomly pick 75% of the data as training set, and the remaining 25% as validation set. To make our reported result stable and reproducible, we set the random seed manually below.

[ ]:
```
SEED=580
np.random.seed(SEED)
```

We split the data into training set and validation set.

[ ]:
```
from sklearn.model_selection import train_test_split
X = df.drop(MEASUREMENT_COLUMN, axis=1)
y = df[MEASUREMENT_COLUMN]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,␣
↪random_state=SEED)
```

The data is shown below, which is expected to be the same in any times of running.

[ ]:
```
print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
print(y_train)
print(X_train.head())
```

```
(785, 56) (262, 56) (785,) (262,)
272      0.000000
19      11.066654
522     11.002117
188     12.380030
128     11.894788
          …
894      0.000000
1238     6.846943
1054    11.532248
296     11.751950
158     11.314487
Name: SEVERITY, Length: 785, dtype: float64
       OBS    YEAR  MONTH U.S._STATE POSTAL.CODE NERC.REGION  \
272  272.0  2011.0    6.0      Texas          TX         SPP
19    19.0  2011.0    4.0  Tennessee          TN        SERC
522  522.0  2006.0    2.0   Maryland          MD         RFC
188  188.0  2000.0    5.0      Texas          TX         TRE
128  128.0  2013.0    1.0   Michigan          MI         RFC

  CLIMATE.REGION ANOMALY.LEVEL CLIMATE.CATEGORY    OUTAGE.START.DATE  \
```

```
272                South          -0.3          normal  2011-06-29 00:00:00
19               Central          -0.5            cold  2011-04-19 00:00:00
522            Northeast          -0.6            cold  2006-02-12 00:00:00
188                South          -0.7            cold  2000-05-02 00:00:00
128   East North Central          -0.4          normal  2013-01-20 00:00:00

     … POPPCT_URBAN POPPCT_UC POPDEN_URBAN POPDEN_UC POPDEN_RURAL  \
272  …         84.7      9.35       2435.3    1539.9         15.2
19   …        66.39     12.02       1450.3    1076.2         55.6
522  …         87.2      3.66       2511.4    1291.8           96
188  …         84.7      9.35       2435.3    1539.9         15.2
128  …        74.57      8.19       2034.1    1390.4         47.5

    AREAPCT_URBAN AREAPCT_UC   PCT_LAND  PCT_WATER_TOT PCT_WATER_INLAND
272          3.35       0.58  97.258336       2.742036         2.090873
19           7.05       1.72  97.843109       2.156891         2.156891
522         20.65       1.69  78.244398      21.755602         6.190553
188          3.35       0.58  97.258336       2.742036         2.090873
128          6.41       1.03  58.459995      41.540005         2.068987

[5 rows x 56 columns]
```

First, we define a helper class to convert the cause category strings into ordinal values, which will be used in the `ColumnTransformer` later.

We find observe all the different values of cause category.

[ ]:
```python
cause_category = list(df['CAUSE.CATEGORY'].unique())
cause_mapping = {v:i for i,v in enumerate(cause_category)}
print(cause_mapping)
```

```
{'severe weather': 0, 'intentional attack': 1, 'public appeal': 2, 'system
operability disruption': 3, 'islanding': 4, 'equipment failure': 5, 'fuel supply
emergency': 6}
```

We then use it to construct helper class `CauseCategoryTransformer`.

[ ]:
```python
from sklearn.base import BaseEstimator, TransformerMixin
class CauseCategoryTransformer(BaseEstimator, TransformerMixin):
    def fit(self, X, y=None):
        return self

    def transform(self, X):
        return np.array([cause_mapping[item] for item in X.iloc[:, 0]]).
 ↪reshape(-1, 1)
```

Later, we define a `ColumnTransformer` to make transform described above.

[ ]:
```python
from sklearn.compose import ColumnTransformer
baseline_col_transformer = ColumnTransformer(
```

```
    transformers=[
        ('cause category', CauseCategoryTransformer(), ['CAUSE.CATEGORY']),
    ]
)
```

Before using it, we'd check it by outputting the transformed results.

[ ]:
```
from sklearn.pipeline import Pipeline
temp_pipeline = Pipeline(steps=[('transform', baseline_col_transformer)])
temp_values = temp_pipeline.fit_transform(X_train)
print(temp_values[:5])
print(set(temp_values.flatten()))
```

```
[[2]
 [0]
 [0]
 [0]
 [0]]
{0, 1, 2, 3, 4, 5, 6}
```

Then, we adds up the decision tree regression model into our baseline pipeline model.

[ ]:
```
from sklearn.tree import DecisionTreeRegressor
baseline_pipeline = Pipeline(steps=[
    ('transform', baseline_col_transformer),
    ('model', DecisionTreeRegressor(random_state=SEED))
])
```

We use the baseline model to predict values in both train set and validation set, and calculate the metric selected above.

We define a helper funcion to reuse better and using it twice and later.

[ ]:
```
def perform_pipeline(pipeline, verbose=1, return_result=False, fit=True):
    if fit:
        pipeline.fit(X_train, y_train)
    y_train_pred = pipeline.predict(X_train)
    y_test_pred = pipeline.predict(X_test)
    print('train evaluate:', measure(y_train, y_train_pred))
    print('test evaluate:',  measure(y_test, y_test_pred))
    if verbose >= 1: #print some samples
        print('samples of train prediction:')
        for i in range(5):
            print(y_train_pred[i], y_train.to_numpy()[i])
        print('samples of test prediction:')
        for i in range(5):
            print(y_test_pred[i], y_test.to_numpy()[i])
    if verbose == 2: # plot some samples
        from plotly.subplots import make_subplots
        import plotly.graph_objects as go
```

```
        titles = (('Prediction on Train Data', 'Prediction on Test Data'))
        fig = make_subplots(rows=1, cols=2, subplot_titles=titles)
        fig.add_trace(
            go.Scatter(x=list(range(len(y_train))),
                       y=y_train.to_numpy(),␣
↪name='train_real',mode='markers'),row=1,col=1
        )
        fig.add_trace(
            go.Scatter(x=list(range(len(y_train_pred))),
                       y=y_train_pred,␣
↪name='train_pred',mode='markers'),row=1,col=1
        )
        fig.add_trace(
            go.Scatter(x=list(range(len(y_test))),
                       y=y_test.to_numpy(),␣
↪name='test_real',mode='markers'),row=1,col=2
        )
        fig.add_trace(
            go.Scatter(x=list(range(len(y_test_pred))),
                       y=y_test_pred,␣
↪name='test_pred',mode='markers'),row=1,col=2
        )
        # fig.show()
        export_plotly_fig(fig, 'perform_pipeline.html')
        return fig
    if return_result:
        return y_train_pred, y_test_pred
```

[ ]:
```
perform_pipeline(baseline_pipeline, return_result=False, verbose=2)
```

```
train evaluate: 0.797048102321281
test evaluate: 0.7960633608313297
samples of train prediction:
4.473081447563073 0.0
11.497164180226365 11.06665398721974
11.497164180226365 11.002116507732017
11.497164180226365 12.380030154325457
11.497164180226365 11.89478753335453
samples of test prediction:
11.497164180226365 11.33201491224167
0.7464137459255406 0.0
11.497164180226365 10.829055121190493
11.497164180226365 14.056017686076457
10.888206112773632 12.577639650232573
```

We find that our model work well on both train and test data. The R^2 are both approximately 0.79, and by looking at some real examples of the prediction, we find it gets a near value. This means that the 7 different types of cause category can roughly related to 7 different order of magnitude in the number of affected customers.

1.1.3 Final Model

The first improvement may lies in hypermarameter selection.

We first present a hyperparameter seaching helper function using `GridSearchCV`.

[ ]:
```
from sklearn.model_selection import PredefinedSplit, GridSearchCV
def grid_search(pipeline, param_grid, verbose=0):
    test_fold = np.concatenate((
        -np.ones(X_train.shape[0]),
        np.zeros(X_test.shape[0])
    ))
    X_ = pd.concat([X_train, X_test])
    y_ = pd.concat([y_train, y_test])
    ps = PredefinedSplit(test_fold)
    search = GridSearchCV(pipeline, param_grid, cv=ps, refit=True,␣
 ↪verbose=verbose)
    search.fit(X_,y_)
    print('best param:', search.best_params_)
    return search
def search_and_compare(pipeline, param_grid, verbose=0, returned=False):
    print('Before: ')
    perform_pipeline(pipeline, verbose=0)
    pipeline = grid_search(pipeline, param_grid, verbose=verbose)
    print('After: ')
    perform_pipeline(pipeline, verbose=0, fit=False)
    if returned:
        return pipeline
```

we find that the `max_depth` parameter is important for the `DecisionTreeRegression`, so we use it to search the tree depth.

[ ]:
```
param_grid = {'model__max_depth':
 ↪[1,2,3,4,5,6,7,8,9,10,12,14,16,18,20,22,24,26,28,30,50]}
from sklearn.base import clone
baseline_pipeline_hyp = clone(baseline_pipeline)
search_and_compare(baseline_pipeline_hyp, param_grid)
```

```
Before:
train evaluate: 0.797048102321281
test evaluate: 0.7960633608313297
best param: {'model__max_depth': 5}
After:
train evaluate: 0.7942281907245848
test evaluate: 0.8153732877355853
```

As we expected, no improvement found, because the `CAUSE.CATEGORY` is too simple(only 7 different values), there may have little improvement by changing tree depth.

Since it's a categorical feature, it's classic that we try using either converting it into nominal encoding or ordinal encoding.

Therefore, we then try using `OneHotEncoder` converting it into nominal encoding to replace the ordinal encoding.

```
[ ]: from sklearn.tree import DecisionTreeRegressor
from sklearn.preprocessing import OneHotEncoder
baseline_pipeline_onehot = Pipeline(steps=[
    ('transform', ColumnTransformer(
        transformers=[
            ('cause category', OneHotEncoder(categories=[cause_category]),␣
 ↪['CAUSE.CATEGORY']),
        ])),
    ('model', DecisionTreeRegressor(random_state=SEED))
])
search_and_compare(baseline_pipeline_onehot, param_grid)
```

```
Before:
train evaluate: 0.797048102321281
test evaluate: 0.7960633608313297
best param: {'model__max_depth': 6}
After:
train evaluate: 0.7942281907245848
test evaluate: 0.8153732877355852
```

Again, no improvement found.

We'd like to find new features now.

We adopt `NERC.REGION` now, since we consider different regions of NERC have different ability to deal with outage, thus making severity different.

```
[ ]: nerc_category = df['NERC.REGION'].unique()
pipeline_nerc_added = Pipeline(steps=[
    ('transform', ColumnTransformer(
        transformers=[
            ('cause category', CauseCategoryTransformer(), ['CAUSE.CATEGORY']),
            ('nerc', OneHotEncoder(categories=[nerc_category]), ['NERC.
 ↪REGION']),
        ])),
    ('model', DecisionTreeRegressor(random_state=SEED))
])
search_and_compare(pipeline_nerc_added, param_grid)
```

```
Before:
train evaluate: 0.8381944183671518
test evaluate: 0.8005079197906323
best param: {'model__max_depth': 7}
After:
```

```
train evaluate: 0.8332869648011452
test evaluate: 0.8299545165859643
```

We find that adding `NERC.REGION` can improve a little. So we adopt it.

We then try many other features to add into the model, but almost no more valid improvement can be seen(due to space constraints, the process is omitted here).

We find that another useful feature is `DEMAND.LOSS.MW`. However, the `DEMAND.LOSS.MW` fature may belong to the feature we would not know at the "time of prediction", the improvement is shown below, but we won't add it into our final model.

[ ]:
```
nerc_category = df['NERC.REGION'].unique()
pipeline_loss_added = Pipeline(steps=[
    ('transform', ColumnTransformer(
        transformers=[
            ('cause category', CauseCategoryTransformer(), ['CAUSE.CATEGORY']),
            ('nerc', OneHotEncoder(categories=[nerc_category]), ['NERC.
 ↪REGION']),
            ('loss', 'passthrough', ['DEMAND.LOSS.MW']),
        ])),
    ('model', DecisionTreeRegressor(random_state=SEED))
])
search_and_compare(pipeline_loss_added, param_grid)
```

```
Before:
train evaluate: 0.9259276580511695
test evaluate: 0.8507129310451179
best param: {'model__max_depth': 7}
After:
train evaluate: 0.9028152408042884
test evaluate: 0.8892435495735955
```

So in conclusion, we try as many as near 20 features and their combinations, but only find three features useful, which are `CAUSE.CATEGORY`, `NERC.REGION`, `DEMAND.LOSS.MW`, while the first two are the information we would know at the "time of prediction", so we only use two features `CAUSE.CATEGORY` as well as `NERC.REGION`.

Finally, we try changing it into `RandomForestRegressor` and perform hyperparameter seaching again.

[ ]:
```
from sklearn.ensemble import RandomForestRegressor
pipeline_randomforest = Pipeline(steps=[
    ('transform', ColumnTransformer(
        transformers=[
            ('cause category', CauseCategoryTransformer(), ['CAUSE.CATEGORY']),
            ('nerc', OneHotEncoder(categories=[nerc_category]), ['NERC.
 ↪REGION']),
        ])),
    ('model', RandomForestRegressor(random_state=SEED))
```

```
])
param_grid_randomforest = {'model__max_depth':[1,2,3,4,5,6,7,8,20,50,100],
              'model__n_estimators':[1,10,25,50,100]}
search_and_compare(pipeline_randomforest, param_grid_randomforest)
```

```
Before:
train evaluate: 0.8367708545475604
test evaluate: 0.7952380087604589
best param: {'model__max_depth': 8, 'model__n_estimators': 50}
After:
train evaluate: 0.8335971481348943
test evaluate: 0.8283529043288325
```

We find that the two models, `DecisionTreeRegressor` and `RandomForestRegressor`, are almost the same. Also, we've performed the `LinearRegression`, `KNeighborsRegressor` and `SVR`, the three models all work terribly(due to space constraints, the process is omitted here). So we simply adopt the `DecisionTreeRegressor`.

Therefore, our final model is shown below. The visualization that describes our mode's performance is shown below.

[ ]:
```
final_model = clone(pipeline_nerc_added)
final_model = grid_search(final_model, param_grid)
perform_pipeline(final_model, fit=False, verbose=2)
```

```
best param: {'model__max_depth': 7}
train evaluate: 0.8332869648011452
test evaluate: 0.8299545165859643
samples of train prediction:
3.1575938521873894 0.0
11.436979145325328 11.06665398721974
11.668984523874991 11.002116507732017
11.990076190339858 12.380030154325457
11.668984523874991 11.89478753335453
samples of test prediction:
11.436979145325328 11.33201491224167
1.1816056682928768 0.0
11.436979145325328 10.829055121190493
11.436979145325328 14.056017686076457
11.467240818602734 12.577639650232573
```

We'd perform it in the whole data, compared with baseline model.

[ ]:
```
def perform(pipeline, returned=False):
    y_pred = pipeline.predict(X)
    print("R2:", get_R2(y, y_pred))
    if returned:
        return y_pred
```

[ ]:
```
perform(baseline_pipeline)
perform(final_model)
```

```
R2: 0.7969215399890912
R2: 0.832557579647293
```

There's improvement on R^2 in the final model, which means that our improvement methods are useful.

1.1.4 Fairness Analysis

To answer the question that whether our model is fair, that is, if it work worse for individuals in some groups than it does in others, we'd perform a fairness analysis below.

The quantitative attribute(evaluation metric) we adopt is R^2, so we use R^2 across two groups to perform the analysis, that is, absolute difference between the R^2 values: $|R^2_{groupX} - R^2_{groupY}|$.

We simply define:

1. group X as the outage where `CLIMATE.CATEGORY` is `cold`
2. group Y as the outage where `CLIMATE.CATEGORY` is not `cold`.

Obviously, it's a binary groups.

we use permutation test to perform it.

Null hypothesis: Our model is fair. Its precision for the outage where the climate is cold and not cold are roughly the same, and any differences are due to random chance.

Alternative hypothesis: Our model is unfair. Its precision for the outage where the climate is cold is lower than that of the outage where the climate is not cold, or otherwise.

Significance level: 0.05.

Since p-value measures the probability of a extreme case happens if null hypothesis is true, and if it's not the same(which means extreme), the evaluation metric will be greater, so we adds up p-value when simated value is greater than observed value. Codes are shown below.

[ ]:
```
def calc_R2(model, X, y):
    y_pred = model.predict(X)
    return get_R2(y, y_pred)
def diff_of_R2(model, df, col, val):
    df1 = df[df[col]==val]
    df2 = df[df[col]!=val]
    r21 = calc_R2(model, df1.drop(MEASUREMENT_COLUMN, axis=1),␣
 ↪df1[MEASUREMENT_COLUMN])
    r22 = calc_R2(model, df2.drop(MEASUREMENT_COLUMN, axis=1),␣
 ↪df2[MEASUREMENT_COLUMN])
    return abs(r21-r22)
def report_perm_test(observed, p_value, simulated, col, val):
    fig = px.histogram(pd.DataFrame(simulated), x=0, nbins=20,␣
 ↪histnorm='probability')
    fig.add_vline(x=observed, line_color='red')
```

```
    fig.add_annotation(text=f'<span style="color:red">Observed =␣
↪{round(observed, 2)}, p_value = {round(p_value ,2)}</span>',
                   x= 0.4, showarrow=False, y=0.1)
    fig.update_layout(title = f"Empirical Distribution to check whether <br>␣
↪{col} Have Different Precision Performance when <br> {col} is {val} and is␣
↪not {val}", xaxis_title="Diff of R2")
    fig.show()
    export_plotly_fig(fig, filename="permutation_test.html")
def permutation_test(model, df, col, val, rounds=500):
    observed = diff_of_R2(model, df, col, val)
    simulated = np.zeros(rounds)
    df2 = df.copy()
    for _ in range(rounds):
        df2[col] = df[col].sample(frac=1,random_state=SEED+_).
↪reset_index(drop=True)
        simulated[_] = diff_of_R2(model, df2, col, val)
    p_value = np.mean(simulated >= observed)
    report_perm_test(observed, p_value, simulated, col, val)
    return observed, p_value
print(permutation_test(final_model, df, 'CLIMATE.CATEGORY', 'cold'))
```

```
(0.09275024280299682, 0.378)
```

To make our result reproducible, we use a static seed list(`SEED+_`) to random shuffle permutation above.

The result shows that p-value is 0.378.

We use a significance level of 0.05. Since p-value is greater than 0.05, we fail to reject the null hypothesis, which means that it's more possible that our model is fair, its precision for different groups are roughly the same.

It also imply that `CLIMATE.CATEGORY` have no effect on predicting the severity, which proves our conclusion that `CLIMATE.CATEGORY` is useless feature to predicting the severity is correct.