Report

0 Readme — How to execute

Run all cells in Jupiter notebook :) (parameter settings are discussed in the following report)

Since I validated several models, it may need around 10 min to run the cross-validation part.

1 Data Preprocessing

1.1 Imputation

1.1.1 Info (tabular data)

Only bacteria includes NaN, so impute it by 0 which indicates no bacteria for the subsequent text processing.

```
info_df.loc[:,'Bacteria'] = info_df.loc[:,'Bacteria'].fillna(0)
```

1.1.2 TPR (time-series data)

Adopting last observation carried forward (LOCF) first and if it still has NaN, then use backward imputation for each patient.

```
dfs = []
for patient_id, patient_df in tpr_df.groupby('No'):
     # last observation carried forward (LOCF)
    dfs.append(patient_df.fillna(method='ffill').fillna(method='bfill'))
tpr_df = pd.concat(dfs).sort_index()
```

1.2 Handling text and use one-hot coding to create dummies

1.2.1 Handling text

For these two features: ‘antibiotics’, ‘bacteria’, I split text, removed numbers, symbols, and the quantities of doses. To correct text automatically, I used “pyspellchecker”, a python package which adopts Levenshtein Distance algorithm, to detect misspellings and correct them based on the word thesaurus I had saved.

Theasarus

```
from spellchecker import SpellChecker

spell = SpellChecker(language=None, case_sensitive=True)

# Antibiotics
spell.word_frequency.load_words(['acyclovir', 'amikacin', 'amoxicillin',
'amphotericin_b_deoxycholate', 'ampicillin', 'azithromycin', 'cefaclor',
'cefazolin', 'cefepime', 'cefixime', 'cefmetazole', 'cefoperazone',
'ceftazidime', 'ceftibuten', 'ceftriaxone', 'cephradine', 'cilastatin',
'ciprofloxacin', 'clavulanic_acid', 'clindamycin', 'cobicistat',
'colistimethate', 'darunavir', 'dicloxacillin_sod', 'doxycycline',
'emtricitabine', 'entecavir', 'ertapenem_sod', 'erythromycin_estolate',
'famciclovir', 'flomoxef', 'fluconazole', 'fusidate_sod', 'gemifloxacin',
'hydroxychloroquine', 'imipenem', 'levofloxacin', 'linezolid', 'meropenem',
'metronidazole', 'minocycline', 'moxifloxacin', 'moxifloxacin_hydrochloride',
'norfloxacin', 'nystatin', 'oseltamivir', 'oxacillin_sodium', 'piperacillin',
'po', 'rifampin', 'rilpivirine', 'ritonavir', 'smz_tmp', 'sertaconazole',
'sulbactam', 'sultamicillin', 'tazobactam', 'teicoplanin', 'tenofovir',
'tenofovir_alafenamide', 'tigecycline', 'triamcinolone_acetonide',
'trimethoprim', 'unifradine', 'valganciclovir', 'vancomycin'])

# Bacteria
spell.word_frequency.load_words(['achromobacter_denitrificans',
'acinetobacter_baumannii', 'candida_albicans', 'candida_glabrata',
'candida_tropicalis', 'citrobacter_koseri', 'corynebacterium_striatum',
'cyberlindnera_fabianii', 'enterobacter_cloacae_ssp_cloacae',
'enterococcus_faecalis', 'enterococcus_faecium', 'escherichia_coli',
'klebsiella_pneumoniae_ssp_ozaenae', 'klebsiella pneumoniae_ssp_pneumoniae',
'morganella_morganii_ssp_morganii', 'proteus_mirabilis', 'providencia_stuartii',
'pseudomonas_aeruginosa', 'pseudomonas_putida', 'salmonella_group_c1',
'salmonella_group_d1(non typhi)', 'staphylococcus_aureus',
'staphylococcus_capitis', 'staphylococcus_caprae', 'staphylococcus_epidermidis',
'staphylococcus_haemolyticus', 'staphylococcus_pettenkoferi', 'yeast_like',
'klebsiella_pneumoniae_ssp_pneumoniae', 'staphylococcus_aureus_mrsa',
'salmonella_roup_dnon_typhi', 'enterococcus_faecium_vre'])
```

Tokenizing

```
def tokenizer(s):
    def customized_split(s, token):
        result = []
        for v in s.split(token):
            result.append(v.split(' ')[0])
        return result

    if s == 0: return ''
    if isinstance(s, int):
        return str(s)

    legal_string = ''
    # Trim illegal marks, numbers, and 'mg'
    for i, ch in enumerate(s):
        # isNumber
        if 48 <= ord(ch) <= 57 or ch in '/()': continue

        # mg
        if ch == 'm' and i < len(s)-1:
            if s[i+1] == 'g': continue
        if ch == 'g' and i > 0:
            if s[i-1] == 'm' or 48 <= ord(s[i-1]) <= 57 or s[i-1] == ' ': continue

        # split small + large
        if 65 <= ord(ch) <= 90 and i > 0:
            if 97 <= ord(s[i-1]) <= 122:
                legal_string += ','+ ch.lower()
                continue

        # isAlphebet
        if 97 <= ord(ch.lower()) <= 122:
            legal_string += ch.lower()
        elif ch in (' ', '-'):
            legal_string += ' '
        else:
            legal_string += ','

    # Check spelling
    legal_concepts = []
    for c in legal_string.split(','):
        if c in ('', ' '): continue

        concept = '_'.join([x for x in c.split(' ') if x != ''])

        if concept in spell:
            legal_concepts.append(concept)
            continue

        misspelled = spell.unknown([concept])
        for word in misspelled:
            corrected_word = spell.correction(word)
            if corrected_word in spell: legal_concepts.append(corrected_word)

    return ','.join(legal_concepts)
```

```
info_df.loc[:,'Comorbidities'] = info_df.loc[:,'Comorbidities'].apply(lambda s: str(s)  if s != 0 else '')
info_df.loc[:,'Antibiotics']   = info_df.loc[:,'Antibiotics'].apply(tokenizer)
info_df.loc[:,'Bacteria']      = info_df.loc[:,'Bacteria'].apply(tokenizer)
```

1.2.2 Create dummies by one-hot coding

Created dummies for these three features: ‘comorbidities’, ’antibiotics’, ‘bacteria’ by one-hot coding. After trimming the less frequent features/categories (threshold frequency >= 5), there are 3 categories for comorbidities, 32 categories for antibiotics, and 8 categories for bacteria.

```
# Create dummies
dummy_dfs = []
if mode == 'train':
    for col in ('Comorbidities', 'Antibiotics', 'Bacteria'):
        dummy_df = info_df[col].str.get_dummies(sep=',')
        # trim those have low frequencies
        dummy_cols = []
        for dummy in dummy_df.columns:
            if dummy_df[dummy].sum() >= 5:
                dummy_cols.append(dummy)

        dummies.append(dummy_cols)
        dummy_df = dummy_df[dummy_cols]
        dummy_df.rename(columns=rename_dummies(dummy_df, col), inplace=True)
        dummy_dfs.append(dummy_df)
```

```
elif mode == 'test':
    for i, col in enumerate(('Comorbidities', 'Antibiotics', 'Bacteria')):
        feature_columns = dummies[i]
        n_col = len(feature_columns)
        dummy_df = pd.DataFrame(np.zeros((info_df.shape[0], n_col)), columns=feature_columns)
        for dummy_name in dummy_df.columns:
            dummy_df.loc[:,dummy_name] = info_df.loc[:,col].apply(lambda s: int(dummy_name in s.split(',')))
        dummy_df.rename(columns=rename_dummies(dummy_df, col), inplace=True)
        dummy_dfs.append(dummy_df)
```

1.3 Generate features for TPR

Extract new features for each patient by aggregating each feature in TPR: mean/min/max/median/std

```
# Window-based summary
# generate features: mean/min/max/median/std
tpr_df = tpr_df.groupby('No').agg(['mean', 'min', 'max', 'median', 'std'])
tpr_df.columns = list(map(lambda tup: tup[0]+'_'+tup[1], tpr_df.columns.to_flat_index()))
```

1.4 Create interaction terms

All interaction terms of comorbidities: 1:2, 1:4, 2:4, 1:2:4

```
# Create interaction terms among comorbidities
info_df['Comorbidities_1:2']   = info_df['Comorbidities_1']*info_df['Comorbidities_2']
info_df['Comorbidities_1:4']   = info_df['Comorbidities_1']*info_df['Comorbidities_4']
info_df['Comorbidities_2:4']   = info_df['Comorbidities_2']*info_df['Comorbidities_4']
info_df['Comorbidities_1:2:4'] = info_df['Comorbidities_1']*info_df['Comorbidities_2']*info_df['Comorbidities_4']
```

Pairwise interaction terms of TPR features: T:P, T:R, T:NBPS, T:NBPD, P:R, P:NBPS, P:NBPD, R:NBPS, R:NBPD, NBPS:NBPD

```
# Create interaction terms among tpr features (T, P, R, NBPS, NBPD)
for f1, f2 in combinations(['T', 'P', 'R', 'NBPS', 'NBPD'], 2):
    tpr_df[f1+':'+f2] = tpr_df[f1+'_mean']*tpr_df[f2+'_mean']
```

1.5 Combine tabular and time-series data into tabular data (Info+TPR)

Merging TPR to Info.

```
merged_df = pd.merge(info_df, tpr_df, left_on='No', right_index=True)
```

1.6 Normalization

To make all values non-negative and ranging in [0, 1], I used minmax method to normalize the data values.

```
def normalization(df):
    # Normalizaion
    for col in df.columns:
        if col in ('Target', 'No'): continue
        if len(df.loc[:,col].unique()) == 1: continue
        # minmax
        min_value = df.loc[:,col].min()
        max_value = df.loc[:,col].max()
        delta = max_value - min_value
        df.loc[:,col] = df.loc[:,col].apply(lambda x: (x-min_value)/delta)
    return df
```

During the testing phase, since the distribution of testing data may differ from that of training data, I use minmax to rescale both training data and testing data such that they are both in the same scale and ranging in [0, 1].

```
def rescaling(train_df, test_df):
    # let the scale of test_df the same as scale of train_df
    df = pd.concat([train_df, test_df])
    rescaled_df = normalization(df)
    n = len(train_df)
    return rescaled_df[:n], rescaled_df[n:].drop({'Target'}, axis=1)
```

2 Formula

2.1 Feature Selection (Filter method)

2.1.1 Drop ‘No’ and those features which feature values are unique.

2.1.2 Find the redundant features which are highly correlated to each other (>=0.95), and only preserve the one that is more correlated to the target.

2.1.3 Find statistically significant features on the rest features. Test p-value of each feature by scikit-learn: f_classif and chi2. F-test (f_classif) for numerical features, and Chi-squares (chi2) for boolean features. In addition, if an interaction term is statistically significant, we should include all individual features in that interaction term.

```python
def filter_features(self, X, Y):
    # reduce redundancy
    drop_features = {col for col in X if len(X[col].unique()) == 1 or col=='No'}

    # drop redundant features
    drop_features |= self.redundant(X.drop(drop_features, axis=1), Y)

    # feature selected by p-value
    significant_features = self.feature_seletion(X.drop(drop_features, axis=1), Y)

    drop_features.clear()
    drop_features = {col for col in X if col not in significant_features}

    return drop_features
```

```python
def redundant(self, X, Y):
    corr_matrix = X.corr()
    n = corr_matrix.shape[0]
    redundant_feature_groups = []
    for (i, j) in combinations(range(n),2):
        if abs(corr_matrix.iloc[i,j]) < 0.95: continue
        if not redundant_feature_groups:
            redundant_feature_groups.append([i, j])
            continue

        add = False
        for group in redundant_feature_groups:
            if i in group and j in group:
                add = True
                break
            elif i in group:
                group.append(j)
                add = True
                break
            elif j in group:
                group.append(i)
                add = True
                break

        if not add:
            redundant_feature_groups.append([i, j])

    redundant_features = set()
    for group in redundant_feature_groups:
        correlations = np.zeros((len(group),))
        for i, _id in enumerate(group):
            correlations[i] = X.iloc[:,_id].corr(Y)
        for i in np.argsort(correlations)[:-1]:
            redundant_features.add(X.columns[group[i]])

    return redundant_features
```

```python
def feature_seletion(self, X, Y):

    boolean_features   = {col for col in X if len(X[col].unique()) == 2}
    numerical_features = {col for col in X if len(X[col].unique()) > 2}

    X_numerical = X.drop(boolean_features, axis=1)
    X_boolean   = X.drop(numerical_features, axis=1)

    pvalue_numerical = f_classif(X_numerical, Y)[1]
    pvalue_boolean   = chi2(X_boolean, Y)[1]

    selected_features = set()
    for i, col in enumerate(X_numerical.columns):
        if pvalue_numerical[i] < 0.05:
            selected_features |= {col}
    for i, col in enumerate(X_boolean.columns):
        if pvalue_boolean[i] < 0.05:
            selected_features |= {col}
    individual_features = set()
    for f in selected_features:
        # check interactions
        if ':' not in f: continue
        if 'Comorbidities_' in f:
            terms = f.split('Comorbidities_')[1].split(':')
            individual_features |= {'Comorbidities_'+i for i in terms}
        else:
            terms = f.split(':')
            individual_features |= {i+'_mean' for i in terms}
    selected_features |= individual_features

    return selected_features
```

2.1.4 Drop features that are not selected

```python
def fit_transform(self, X_train, y_train):
    drop_features = self.filter_features(X_train.copy(), y_train.copy())
    self.drop_features = drop_features

    return X_train.drop(drop_features, axis=1)
# ------------------------------------------------------------------------
def transform(self, X_test):
    # Feature Selection
    return X_test.drop(self.drop_features, axis=1)
```

2.2 Logistic Regression Model

```
class LogisticRegression():
    # -----------------------------------------------------------------------
    def __init__(self, lambda_=1, max_iter=100, eta=1e-2, solver='simple_GD'):
        self.lambda_ = lambda_ # regularization term
        self.max_iter = max_iter
        self.eta = eta
        self.solver = solver
        self.weight = []
        self.X = []
        self.y = []
        self.features = []
```

2.2.1 Fit (Input: Tr, model/Output: weights)

2.2.1.1 Procedure of Logistic Regression

Add an additional dimension with constant 1 to the original training dataset X as the constant term.

$$y_{LinearRegression} = \sum_{i=1}^{n} \left(w_0 \cdot 1 + w_i x_i\right) = \sum_{i=0}^{n} \left(w_i x_i\right) = \mathbf{w}^\top \mathbf{X}, \text{ where } x_0 = 1$$

Use gradient descent to update weights in logistic regression. Note that since GD method is vulnerable to initialization, I tried 20 start values and pick the best one with highest score.

```
def fit(self, X, y):
    self.features = X.columns
    self.X = np.hstack((np.ones((X.shape[0],1)),X)).copy() # X_[:,0] = [1, ..., 1]
    self.y = y.copy()

    score = np.zeros((20,))
    ws = np.zeros((20, self.X.shape[1]))

    for i in range(20):
        w = np.random.rand(self.X.shape[1], )
        if self.solver == 'simple_GD':
            w = self.simple_GD(w)
        elif self.solver == 'momentum_GD':
            w = self.momentum_GD(w)
          elif self.solver == 'BFGS':
              w = self.BFGS(w)
        score[i] = self.f(w)
        ws[i] = w
    self.weight = ws[np.argmax(score)]

    return self.weight
```

2.2.1.2 Logistic function (sigmoid)

$$\sigma(a) = \frac{1}{1 + e^{-a}}$$

However, since original logistic function is easy to overflow in practice, I change it in a more stable one.

$$\sigma(a) = \frac{0.5}{1 + \tanh(0.5 \cdot a)}$$

```
def logit(self, x):
    return .5 * (1 + np.tanh(.5 * x))
```

2.2.1.3 Derivation of update formula

Maximum Likelihood

$$P(D \mid \mathbf{w}) = \prod_{i=1}^{N} \left(\frac{1}{1 + e^{-\mathbf{w}^\top x_i}}\right)^{y_i} \left(\frac{e^{-\mathbf{w}^\top x_i}}{1 + e^{-\mathbf{w}^\top x_i}}\right)^{1-y_i}$$

Take logarithm as score function

$$\log\left[P(D\,|\,\mathbf{w})\right] = \sum_{i=1}^{N}\left\{y_i \cdot \log(\sigma(\mathbf{w}^\top x_i)) + (1-y_i)\cdot\log(1-\sigma(\mathbf{w}^\top x_i))\right\}$$

Take partial derivative w.r.t w

$$\begin{aligned}
\nabla E(\mathbf{w}) &= \frac{\partial \log\left[P(D\,|\,\mathbf{w})\right]}{\partial \mathbf{w}} \\
&= \sum_{i=1}^{N}\left[-y_i\cdot\frac{1}{1+e^{-\mathbf{w}^\top x_i}}\cdot e^{-\mathbf{w}^\top x_i}\cdot(-x_i) + (1-y_i)\cdot\left(-x_i - \frac{1}{1+e^{-\mathbf{w}^\top x_i}}\cdot e^{-\mathbf{w}^\top x_i}\cdot(-x_i)\right)\right] \\
&= \sum_{i=1}^{N}\left[y_i\,x_i\cdot\frac{\alpha}{1+\alpha} - x_i + y_i\,x_i + \frac{\alpha}{1+\alpha}x_i - y_i\,x_i\frac{\alpha}{1+\alpha}\right] \\
&= \sum_{i=1}^{N} x_i\left[y_i - \frac{1}{1+\alpha}\right] \\
&= \sum_{i=1}^{N} x_i\left[y_i - \frac{1}{1+e^{-\mathbf{w}^\top x_i}}\right] = X^\top\left[Y - \sigma(\mathbf{w}^\top X)\right]
\end{aligned}$$

Update formula (maximization):

$$\mathbf{w}_{n+1} = \mathbf{w}_n + \eta\cdot\nabla E(\mathbf{w}) = \mathbf{w}_n + \eta\cdot X^\top\left[Y - \sigma(\mathbf{w}^\top X)\right]$$

Ridge regularization (L_2 norm)

$$\text{Score function} = \log\left[P(D\,|\,\mathbf{w})\right] + \lambda\mathbf{w}^\top\mathbf{w}$$

$$\Rightarrow \mathbf{w}_{n+1} = \mathbf{w}_n + \eta\cdot\left\{X^\top\left[Y - \sigma(\mathbf{w}^\top X)\right] + 2\lambda\mathbf{w}\right\}$$

```python
def f(self, w):
    X = self.X
    y = self.y # (N,)
    sigmoid = self.logit(X.dot(w))
    return np.sum(y*np.log(sigmoid) + (1-y)*np.log(1-sigmoid)) - self.lambda_*w.dot(w)
# ---------------------------------------------------------------------------
def f_diff(self, w):
    '''
    input w
    Output (P,)
    '''
    X = self.X # (320,80)
    y = self.y # (320,)
    sigmoid = self.logit(X.dot(w))
    gradient = X.T.dot(y-sigmoid) - 2*self.lambda_*w
    return gradient
```

Add momentum

$$\mathbf{w}_{n+1} = \mathbf{w}_n + \Delta_n$$

$$\Delta_n = 0.1\cdot\eta\nabla E(\mathbf{w}) + 0.9\cdot\Delta_{n-1} = 0.1\cdot\eta\left\{X^\top\left[Y - \sigma(\mathbf{w}^\top X)\right] + \lambda\mathbf{w}\right\} + 0.9\cdot\Delta_{n-1}$$

(Typically the default weight for momentum is 0.9)

```
def momentum_GD(self, w0):
    w = w0.copy()
    momentum = np.zeros(w.shape)
    for iter in range(self.max_iter):
        gradient = self.f_diff(w)
        momentum_new = 0.1*self.eta*gradient + 0.9 * momentum
        w_new = w + momentum_new

        # Check converge
        if LA.norm(w_new - w) < 1e-6:
            return w_new

        momentum = momentum_new
        w = w_new
    return w
```

Update until convergence or meeting the maximum iteration.

2.2.2 Predict (Input: Ts, model/Output: labels)

$$y = \begin{cases} 1 & \text{if } \sigma(\mathbf{w}^\top X) > 0.5 \\ 0 & \text{if } \sigma(\mathbf{w}^\top X) \leq 0.5 \end{cases}$$

```
def predict(self, X):
    X_ = np.hstack((np.ones((X.shape[0],1)),X))
    y = self.logit(X_.dot(self.weight))
    y[y > 0.5] = 1
    y[y <= 0.5] = 0
    return y
```

2.2.3 Predict Probability (Input: Ts, model/Output: probabilities)

$$\text{probability} = \sigma(\mathbf{w}^\top X)$$

```
def predict_probability(self, X):
    X_ = np.hstack((np.ones((X.shape[0],1)),X))
    prob  = self.logit(X_.dot(self.weight))
    return prob
```

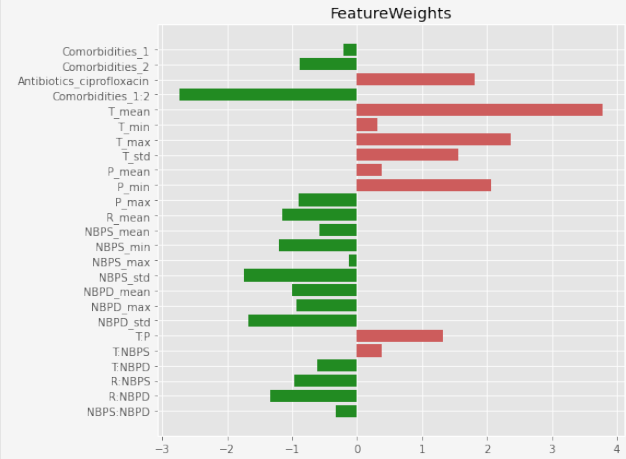
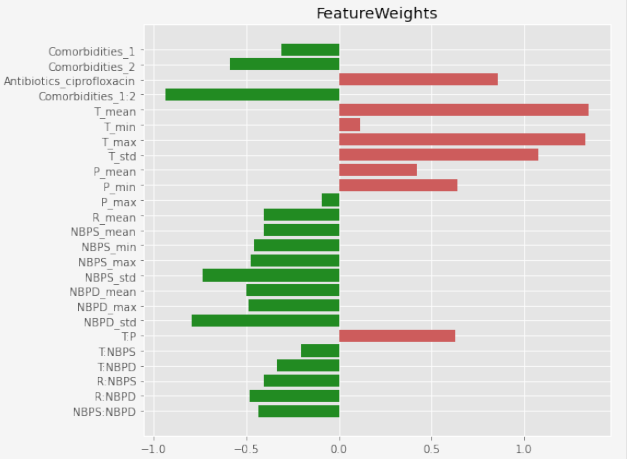
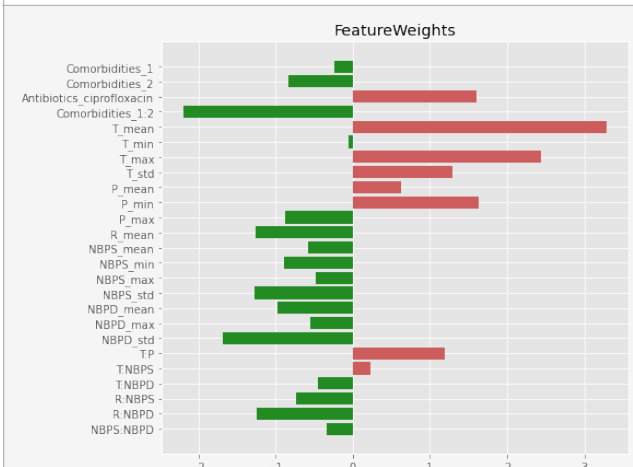
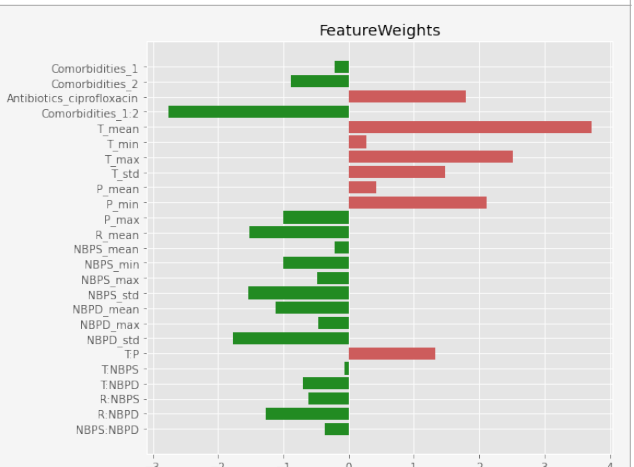
2.2.4 Visualizations

```
def visualize(self):
    weights = self.weight[1:]
    colors = ['indianred' if w > 0  else 'forestgreen' for w in weights]
    features = self.features
    y_pos = np.arange(len(features))

    fig, ax = plt.subplots(figsize=(8,6))
    ax.barh(y_pos, weights, color=colors)
    ax.set_yticks(y_pos)
    ax.set_yticklabels(features)
    ax.invert_yaxis()  # labels read top-to-bottom
    ax.set_title('FeatureWeights')

    fig.patch.set_visible(False)
    fig.tight_layout()
    plt.show()
```

Discussion: Since I filtered the non-significant features at first, the effect of regularization is not that apparent. Yet, it still penalized several features or even change the sign of a feature's weight.

$\lambda = 0$	$\lambda = 1$
	
$\lambda = 1e-1$	$\lambda = 1e-2$
	

2.3 Ensemble

I used a modified bagging method as my ensemble method. Using n_estimators to specify number of estimated models for each input base_models; thus, the total number of models is len(base_models)*(n_estimators+1). (+1: original model)

```
class BaggingClassifier():
    # ------------------------------------------------------------------------
    def __init__(self, base_models, n_estimators):
        self.base_models = base_models
        self.n_estimators = n_estimators
        self.n_models = int(len(base_models)*(n_estimators+1))
        self.models = []
```

Estimated models are built by each base_model and resampled input dataset.
(Applying bootstrap sampling for each estimated model)
Original models are built by each base_model and original input dataset.
Here, I used 10 estimated models + 1 original model.

```
def fit(self, X, y):
    # Model Generation
    models = []
    for base_model in self.base_models:
        for i in range(self.n_estimators):
            sampled_indices = np.random.choice(X.shape[0], X.shape[0])

            X_ = X.iloc[sampled_indices, :]
            y_ = y.iloc[sampled_indices]

            # Train model on Tr_ and store it
            model = deepcopy(base_model)
            model.fit(X_, y_)
            models.append(model)
        model = deepcopy(base_model)
        model.fit(X, y)
        models.append(model)

    self.models = models
```

At prediction phase, obtaining the predicted labels of each models and take majority vote. If using probability instead of labels (0,1), change the input parameter as 'probability'; this takes average of corresponding probabilities.

```
def predict(self, X, method):
    if method == 'majority_vote':
        # Predict by Voting
        Y = np.zeros((self.n_models, X.shape[0]))
        for i, model in enumerate(self.models):
            Y[i] = model.predict(X)

        # Majority Vote
        y = Y.mean(axis=0)
        y[y > 0.5] = 1
        y[y <= 0.5] = 0

    elif method == 'probability':
        # Predict by accumulation
        prob = np.zeros((self.n_models, X.shape[0]))
        for i, model in enumerate(self.models):
            prob[i] = model.predict_probability(X)

        # Take average on probability
        y = prob.mean(axis=0)
        y[y > 0.5] = 1
        y[y <= 0.5] = 0

    return y
```

3 Validation Method: 5-fold Cross Validation

3.1 StratifedKfold

Use scikit-learn StratifiedKfold with shuffling to split training data into 5 folds under the nature of original distribution of target class.

```
def cross_validation(df, kfold:int):

    # split X, Y
    X = df.drop('Target', axis=1)
    y = df.loc[:,'Target']

    # Stratified K fold
    kf = StratifiedKFold(n_splits=kfold, shuffle=True)
    kf.get_n_splits(X)
```

3.2 Iteration

Choose each fold in turn as test data, and the rest are training data.

For each iteration,

3.2.1 Do feature selection first,

3.2.2 Build models

Parameter settings

Logistic Regression	Ensemble
inital η = 1e-2, max iteration = 200, solver = momentum gradient descent	n_estimators = 10

I tried seven different penalties λ = 0, 1, 1e-1, 5e-2, 1e-2, 5e-3, 1e-3, and build an ensemble model for each penalty setting. In addition, I built an additional advanced ensemble model. Such advanced model ensembles models in which penalty=1e-2, 5e-3, 1e-3, and this is my final selection of model.

3.2.3 Record f1 score for each model

3.3 Finally, take average and standard deviation of f1 scores for each model. Then return.

```
param = {'lambda_': [0, 1, 1e-1, 5e-2, 1e-2, 5e-3, 1e-3]}
validated_table = pd.DataFrame(np.zeros((8,8)), columns=['lambda_', 'f1_score_0', 'f1_score_1',
                                                         'f1_score_2', 'f1_score_3', 'f1_score_4',
                                                         'mean', 'std'])
j = 0
for train_index, test_index in kf.split(X, y):
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # Feature Selection
    selector = FeatureSelection()
    X_train = selector.fit_transform(X_train.copy(), y_train.copy())
    X_test = selector.transform(X_test.copy())

    # Training Models and Record F1 scores
    ## Loop over different models with different lambda (penalty)
    models = []
    for i, lambda_ in enumerate(param['lambda_']):
        model    = LogisticRegression(lambda_=lambda_, max_iter=200, eta=1e-2, solver='momentum_GD')
        ensemble = BaggingClassifier([model], n_estimators=10)
        ensemble.fit(X_train.copy(), y_train.copy())

        prediction     = ensemble.predict(X_test.copy(), method='majority_vote')
        tn, fp, fn, tp = confusion_matrix(y_test.to_numpy().flatten(), prediction).ravel()

        validated_table.loc[i, 'lambda_'] = lambda_
        validated_table.loc[i, 'f1_score_'+str(j)] = 2*tp/(2*tp+fp+fn)

        if 1e-3 <= lambda_ <= 1e-2:
            models.append(model)

    ensemble = BaggingClassifier(models, n_estimators=10)
    ensemble.fit(X_train.copy(), y_train.copy())

    prediction = ensemble.predict(X_test.copy(), method='majority_vote')
    tn, fp, fn, tp = confusion_matrix(y_test.to_numpy().flatten(), prediction).ravel()
    validated_table.loc[i+1, 'lambda_'] = 'ensemble'
    validated_table.loc[i+1, 'f1_score_'+str(j)] = 2*tp/(2*tp+fp+fn)

    j += 1

validated_table.loc[:, 'mean'] = validated_table.iloc[:, 1:6].mean(axis=1)
validated_table.loc[:, 'std']  = validated_table.iloc[:, 1:6].std(axis=1)

return validated_table
```

Validation of different models

Main - Training & Cross Validation

```
Info_train_df = pd.read_excel('Training data.xlsx', sheet_name=0)
TPR_train_df  = pd.read_excel('Training data.xlsx', sheet_name=1)
dummies = []
train_df, dummies = preprocessing(deepcopy(Info_train_df), deepcopy(TPR_train_df), 'train', dummies)
```

```
Tr = normalization(train_df.copy())
```

```
kfold = 5
table = cross_validation(Tr, kfold)
```

```
table # max_iter: 200, eta: 1e-2, n_estimator: 10
```

	lambda_	f1_score_0	f1_score_1	f1_score_2	f1_score_3	f1_score_4	mean	std
0	0	0.538462	0.500000	0.600000	0.615385	0.615385	0.573846	0.052058
1	1	0.454545	0.200000	0.363636	0.400000	0.545455	0.392727	0.127662
2	0.1	0.518519	0.500000	0.620690	0.615385	0.615385	0.573995	0.059497
3	0.05	0.592593	0.466667	0.600000	0.560000	0.560000	0.555852	0.053121
4	0.01	0.538462	0.518519	0.625000	0.714286	0.583333	0.595920	0.077953
5	0.005	0.538462	0.518519	0.580645	0.666667	0.615385	0.583935	0.059564
6	0.001	0.538462	0.500000	0.562500	0.666667	0.615385	0.576603	0.065454
7	ensemble	0.538462	0.482759	0.600000	0.714286	0.640000	0.595101	0.089546

The model in the last row is the advanced ensemble model that ensembles the models w/ λ=1e-2, 5e-3, 1e-3.

We can see that the poorest model is the one with λ=1. Ideally, the model with regularization (after pruning) should perform quite better than that without regularization (before pruning). Yet, this effect was not apparent in my results. I speculated that it was because I did feature selection first (filtered non-significant features).

Furthermore, with ensemble, the variance became lower and the performance were preserved.

My selection of model: the last row in the validation table above. (advanced ensemble model)

Main - Testing (Prediction)

```
Info_test_df = pd.read_excel('Testing data.xlsx', sheet_name=0)
TPR_test_df  = pd.read_excel('Testing data.xlsx', sheet_name=1)
test_df, dummies = preprocessing(deepcopy(Info_test_df), deepcopy(TPR_test_df), 'test', dummies)
Tr, Ts = rescaling(train_df.copy(), test_df.copy())
```

```
# split X, Y
X_train = Tr.drop('Target', axis=1)
y_train = Tr.loc[:,'Target']

param = {'lambda_': [0, 1, 1e-1, 5e-2, 1e-2, 5e-3, 1e-3]}
models = []
# Feature Selection
selector = FeatureSelection()
X_train = selector.fit_transform(X_train.copy(), y_train.copy())
X_test = selector.transform(Ts.copy())

for i, lambda_ in enumerate(param['lambda_']):
    model = LogisticRegression(lambda_=lambda_, max_iter=200, eta=1e-2, solver='momentum_GD')
    model.fit(X_train.copy(), y_train.copy())
    models.append(model)

ensemble = BaggingClassifier(models[4:], n_estimators=10)
ensemble.fit(X_train.copy(), y_train.copy())
```

```
prediction = ensemble.predict(X_test.copy(), method='majority_vote')
```

```
# Write file
submission = pd.read_csv('0856617.csv')
for i in range(len(submission)):
    submission.loc[i, 'Target'] = prediction[i]
submission = submission.astype('int32')
submission.to_csv('0856617.csv', index=False)
```