Esther Edith Spurlock (12196692)
CAPP 30254
Assignment 5: Writeup

Note: I could not run this code from beginning to end because of memory issues.

Code Files:
My code contains 3 .py files:
- Full_pipeline: where I call to functions from the beginning through to the end of the pipeline
- Prep_data: where I clean and prepare the data for analysis
- Modeling: where I split the data into training and testing sets by date, create the models, and evaluate those models

Important prep_data.py Functions:
For each testing and training set, I needed to create the different variables and features.

clean_data:
For all numeric values, I imputed the missing values based on the average and then I added another column specifying that the value had been imputed.

```
for col in all_cols:
    ser = df_all_data[col]
    if ser.dtype in ['float64', 'int64']:
        new_col = col + '_imputed_mean'
        col_mean = ser.mean()
        df_all_data[col] = ser.fillna(col_mean)
        #If an entry is equal to the column mean we say it is imputed
        df_all_data[new_col] = df_all_data[col].apply(lambda x:\
            1 if x == col_mean else 0)
```

generate_var_feat:
The variable for this data set depended on whether Date Fully Funded - Date Posted <= 60. I applied this logic in the following code.

```
df_all_data[VAR] = df_all_data[FUNDED] - df_all_data[POSTED]
df_all_data[VAR] = df_all_data[VAR]\
    .apply(lambda x: 0 if x.days <= i else 1)
```

After generating the variable columns, I moved on to features. For every non-numeric column, I created a dummy variable for each value.

```
for col in all_cols:
    #I do not want to include values that are different for every entry
    #I also do not want to include the variable or the dates in my features
    if col not in [PROJ_ID, POSTED, FUNDED, VAR]:
        ser = df_all_data[col]
        if ser.dtype not in ['float64', 'int64']:
            #Find all unique values in the column
            val_unique = ser.unique()
            for val in val_unique:
                new_col = col + "_" + str(val)
                #Create a dummy variable on the column value
                df_all_data[new_col] = df_all_data[col]\
                    .apply(lambda x: 1 if x == val else 0)
                features.append(new_col)
        else:
            features.append(col)
```

After going through this, I ended up with so many features that my code would not run. The features include the numeric columns, whether a numeric value has been imputed or not, and a dummy variable for each of the non-numeric values available to me.

Important modeling.py Functions:
Now that I had my data prepared, I needed to focus on splitting the data and putting it into models.

split_by_date:
To avoid hard-coding, I calculated how many days would be in each split given variable i (the days we want a project funded in) and a variable num_splits (the number of splits we want)

```
#Now we need to find the number of days in each split
#Fist, we find the number of days between the max and min dates
between = (final_date - first_date).days
#Then, we need to find how many days are in the split
days_in_split = ((between / i) / num_splits) * i
```

Then, I split up my days

```python
while end_test < final_date:
    #the training data begins the day after the ending of last train data
    begin_train = end_train + timedelta(days=1)
    end_train = begin_train + timedelta(days=days_in_split)
    #Testing data begins i days after training data ends
    begin_test = end_train + timedelta(days=i)
    end_test = begin_test + timedelta(days=days_in_split-i)
    #Prevents there being a set that is just a few days
    if (final_date - end_test).days <= i:
        end_test = final_date
    dates = str(begin_test) + " - " + str(end_test)
```

After I got my dates, I could get my data for my testing and training variable and features. Then, I created models off those variables and features.

```python
train_variable, train_features, test_variable, test_features =\
    create_train_test_df(df_all_data, begin_train, end_train,\
        begin_test, end_test, split, i)


#Now we create the models dictionary
models_dict[dates] = training_models(train_variable, train_features,\
    test_variable, test_features)
```

create_train_test_df:
I called this function from split_by_date so I could pull out only the information I needed and so I did not impute on future data.

First, I isolated only the dates I needed.

```python
filt =\
    (df_all_data[split] <= end_test) &\
    (df_all_data[split] >= begin_train)
our_data = df_all_data[filt]
```

Next, I sent the data to be cleaned and then to create variables and features.

```
our_data = prep_data.clean_data(df_all_data, all_cols)
all_cols = our_data.columns
our_data, variable, features =\
    prep_data.generate_var_feat(our_data, all_cols, i, split)
```

Finally, I split up the training and testing data by variables and features.

```
train_variable = train_data[variable]
train_features = train_data[features]
test_variable = test_data[variable]
test_features = test_data[features]
```

training_models:
In this function, I expand on the dictionary I created in split_by_date. For this, I created another dictionary. This time, I used the model name as the key and the information about the different models as the value.

```
models_dict[REGRESSION], models_dict[SVM] =\
    regression_svm_modeling(train_variable, train_features, test_variable,\
    test_features)
models_dict[KNN] = knn_modeling(train_variable, train_features,\
    test_variable, test_features)
models_dict[FOREST], models_dict[EXTRA], models_dict[TREE] =\
    forest_modeling(train_variable, train_features, test_variable,\
    test_features)
models_dict[ADA_BOOSTING] = ada_boost_modeling(train_variable,\
    train_features, test_variable, test_features)
models_dict[BAGGING] = bagging_modeling(train_variable, train_features,\
    test_variable, test_features)
```

As you can see from this code, I wrote different functions to create the different models. This is because most models required different parameters. For those that needed the same parameters, I used the same function to call them. After creating an empty model, each of the model functions then called to the training function.

test_models:
This function begins by fitting the current model to the data.

```
model = model_unfit.fit(train_features, train_variable)
```

I could have put the fit into the different model functions, but I wanted to abstract the code a little and not have to write this same line of code for each model I created in each model function.

After fitting the model, I then predicted the model probability that the testing data would result in funding within 60 days.

```
if is_svm:
    probabilities = model.decision_function(test_features)
else:
    probabilities = model.predict_proba(test_features)[:,1]
```

I then added to the dictionary with another dictionary. For this one, the threshold was the key and the value was the different evaluations.

I first called to ROC AUC evaluation since that evaluation only needs the probabilities. I then created a best guess as to whether the output of a row would be 1 or 0 depending on the threshold and used that column to evaluate the model using different evaluations.

```
key = "No Threshold"
roc_auc = roc_auc_score(y_true=test_var, y_score=probabilities)
eval_dict[key] = {ROC_AUC: roc_auc}

#All other evaluations need to loop through the thresholds
for thresh in THRESHOLDS:
    calc_threshold = lambda x,y: 0 if x < y else 1
    predicted = np.array([calc_threshold(score, thresh) for score in
        probabilities])
    key = "Threshold: " + str(thresh)
    eval_dict[key] = evaluate_models(test_var, predicted)
```

evaluate_models:
This function creates the final dictionary. The key is the name of the evaluation metric and the value is how poorly or well the model performs on the metric.

```
eval_dict[ACCURACY] = accuracy(y_true=true, y_pred=predicted)
eval_dict[PRECISION] = precision_score(y_true=true, y_pred=predicted)
eval_dict[RECALL] = recall_score(y_true=true, y_pred=predicted)
eval_dict[F1] = f1_score(y_true=true, y_pred=predicted)
```

plot_pre_rec:
I do not have a call to this function in my code. This function creates a precision-recall curve and saves it to my folder. However, since I did not want to have several thousand graphs saved on my VM, I decided I would only call this function for the models I wanted to see the full curve for. Since I couldn't even create the data to analyze, I don't have an example here.

Important full_pipeline.py Functions:
I created my final return object back in this .py file.

table_models_eval:
Once I have my full dictionary of the dates, models, and evaluations, I needed to put the data into the proper return format. I did this by looping through the dictionary to put the data into a pandas dataframe to return.

```
for dates, model_dict in models_eval.items():
    for model, param_dict in model_dict.items():
        for param, eval_dict in param_dict.items():
            for threshold, eval_outcome_dict in eval_dict.items():
                for eval_name, outcome in eval_outcome_dict.items():
                    this_lst = [dates, model, param, eval_name, threshold,\
                        outcome]
                    df_lst.append(this_lst)

df_evaluated_models = pd.DataFrame(np.array(df_lst), columns=col_lst)
```

Places For Improvements:

- I did not have time to change thresholds to percent of the population.
- I did not have time to figure out how to not make functions for all of my classifiers.
- I do not know what to do for data exploration.
- I did not run this from beginning to end because of memory issues.
- Mostly, I wish I had more time to figure everything out. I'm sorry.