Rapport de la Compétition 1
Classification of extreme weather events

présenté à

Guillaume Rabusseau

IFT3395

Équipe Kaggle : SW
Membres :
Steve Lévesque - 20149530
Weiyue Cai - 20006782

2 novembre 2021

Table des matières

1 Introduction

Le but de ce travail de compétition Kaggle est de trouver le meilleur modèle possible de régression linéraire (et le meilleur en général pour la question suivant cela) capable de classifier des événements météorologiques en 3 catégories : standard (0), Cyclones tropicaux (1) et Rivières athmosphériques (2).

Voici un résumé des approches qu'on a utilisées avec meilleur score privé et public :
(1) Régression logistique à partir de zéro avec sigmoid (0.76311, 0.75109)
(2) Régression logistique à partir de zéro avec softmax (0.77021, 0.75710)
(3) Réseau de neurones (0.78333, 0.77131)
(4) Voting classifier (0.77568, 0.76721
(5) Stacking classifier (0.79043, 0.78661)
Le stacking classifier nous a donné le meilleur score sur le public leaderboard et le private leaderboard.

2 Feature Design

Nous avons pré-traité les données de plusieurs manières pour entreprendre nos prédictions :

- Standardisation des données.
- Séparation one-hot des données par saison ('spring', 'summer', 'automn', 'winter') après avoir effectué la visualisation des données.
- Choix des attributs inutiles ou répétitifs ('S.No', 'time', 'PS') à exclure du jeu de données.
- Suppression des attributs ('PRECT', 'T200') qui ne sont pas significatifs grâce à la méthode de "feature importance" des modèles random forest, gradient boosting et XGBoost.

Veuillez voir les notebooks en annexe pour plus de détails.

3 Algorithmes

3.1 Régression logistique

Pour les aspects d'algorithmes d'apprentissage, nous avons utilisé la descente du gradient régularisé pour ce qui est de l'optimisation du coût.

3.2 Comparaison des classifiers

Voici la liste des algorithmes qu'on a essayées avec très peu de hyperparameters tuning (avec la moyenne et l'écart-type du taux de précision).

```
Logistic Regression: 0.821001, 0.0034
```

```
SVM: 0.844116, 0.0033
Decision Tree Classifier: 0.855653, 0.0025
KNN: 0.889175, 0.0027
Random Forest Classifier: 0.859883, 0.0029
Gradient Boosting Classifier: 0.863714, 0.0027
Extra Trees Classifier: 0.861642, 0.0017
Light Gradient Boosting Machine: 0.889363, 0.0031
eXtreme Gradient Boosting: 0.852931, 0.0034
```

3.3 Réseau des neurones

Nous avons implémenté à l'aide de Keras le réseau des neurones avec 1 hidden layer de 14 neurones. Nous avons appliqué la régularisation L2 sur le output layer.

3.4 Voting Classifier

Nous avons gardé tous les attributs sauf 'S.No', 'time', 'PS' et choisi tous les modèles mentionnés ci-haut sauf KNN pour le soft voting classifier.

3.5 Stacking Classifier

Nous avons choisi deux algorithmes moins performants (XGB et GB) de la famille de boosting, SVM et Random Forest comme la combinaison des estimateurs et Logistic Restression comme l'estimateur final.

4 Méthodologie

4.1 Stratégies de régularisation

En ce qui concerne la régression logistique à partir de zéro, nous avons utilisé des algorithmes gloutons pour arriver à complétion de nos tâches. Par exemple, la fouille pour trouver les meilleurs hyperparamètres est un algorithme "brute-force" glouton.

4.2 Choix des modèles et hyperparamètres

Au niveau du choix des modèles pour le soft voting classifier et le stacking classifier, nous n'avons pas choisi les modèles les plus performants (KNN et Light Gradient Boosting Machine). Certaines raisons peuvent justifier notre choix :
- Très bonne performance sur le modèle d'entraînement pourrait causer le overfitting sur le jeu de données de test.
- Puisque les labels ne sont pas distribués de façon équilibré (le nombre de la classe 0 est

beaucoup plus élevé que les deux autres classes), un meilleur taux de précision pourrait s'expliquer par le fait que le modèle prédit plus de 0.

Similairement, nous avons utilisé le GridSearchCV pour trouver les meilleurs hyperparamètres sur le jeu de données d'entraînement. Mais cela ne signifie pas nécessairement que ces hyperparamètres vont donner une meilleur prédiction.

5 Résultats

5.1 Régression logistique

5.1.1 Sigmoid

Le meilleur private score : 0.76311 (battu logreg baseline)
public score : 0.75109
Pré-traiement des données :
- Standardiser les données
- Enlever l'attribut PS
Paramètres :
alpha = 1.5, max iter : 25000, lambda : 3.5.

5.1.2 Softmax

Le meilleur private score : 0.77021 (battu TA baseline)
public score : 0.75710
Pré-traiement des données :
- Standardiser les donnees - Enlever les attributs PS et PRECT
Paramètres :
alpha = 0.85, max iter = 4000, lambda = 3.

5.2 Autres modèles (avec modules)

5.2.1 Réseau des neurones

Voici les paramètres qu'on a utilisés pour obtenir le meilleur score avec le réseau des neurones.
Private score : 0.78005, Public score 0.77459

```
# create model
model = Sequential()
model.add(Dense(14, input_dim=19, activation='relu'))
model.add(Dense(3, activation='softmax', kernel_regularizer='l2'))
# Compile model
```

```
model.compile(loss='categorical_crossentropy',
optimizer='adam',
metrics=['accuracy'])

training_result = model.fit(X_train_nn,
                            y_train_nn,
                            batch_size=10,
                            epochs=50,
                            validation_data=(X_val_nn, y_val_nn))
```

5.2.2 Soft Voting Classifier

Nous avons utilisé le model suivant et toutes les données en enlevant les attributs "S.NO", "PS" et "time".
Private score : 0.77568, Public score 0.76721

```
final_models = [
      ('lr', LogisticRegression(solver='newton-cg')),
      ('dt', DecisionTreeClassifier()),
      ('rf', RandomForestClassifier()),
      ('gb', GradientBoostingClassifier()),
      ('etc', ExtraTreesClassifier()),
      ('svc', SVC(probability=True)),
      ('lgbm', LGBMClassifier()),
      ('xgb', XGBClassifier())
      ]
```

5.2.3 Stacking Classifier

Voici le modèle qui nous a donné le meilleur socre :
Private score : 0.79043, Public score 0.78661

```
models = [
      ('xgb',  XGBClassifier(
                objective='multi:softprob',
                eval_metric='merror',
                colsample_bytree=1,
                learning_rate=0.02,
                max_depth=4,
                n_estimators=10,
                )
      ),
      ('gb',  GradientBoostingClassifier()),
      ('svc', SVC(C=0.1, random_state=42)),
      ('rf', RandomForestClassifier(max_depth=7))
      ]
```

```
    clf_lr = StackingClassifier(
    estimators=models,
    final_estimator=LogisticRegression(C=0.1,
    multi_class='multinomial',
    solver='lbfgs')
)
```

Nous avons fait plusieurs tentatives avec différentes combinaisons des estimateurs et 4 choix de l'estimateur final (Logistic Regression, Decision Tree, Random Forest et LGBM). Le private score est généralement plus grand que 0.77. Avec la combinaison de svm + rf + xgb et gb comme estimateurs et la regression logistique comme l'estimateur final, le private score est toujours plus grand que 0.78.

6 Discussion

6.1 Régression Logistique à partir de zéro

Notre approche tiens compte de méthodes traditionnelles pour faire sa tâche de manière efficace et concise. Voici les avantages et inconvénients :

6.1.1 Avantages

Convivial à implémenter, comprendre, modifier et à accomplir sa tâche de classification.

6.1.2 Inconvénients

Aucune méthodologie poussée utilisée (i.e. Stochastic GD/BFGS, optimal step search) puisque le dernier "baseline" a été battu sans réflexion plus avancées et avec des opérations gloutones. Donc, le modèle souffre très possiblement d'un temps de calcul plus haut que ce qu'il en vaudrait ainsi que d'un temps d'opération (pour compléter la tâche) plus haut.

6.2 Réseau des neurones

6.2.1 Avantages

En général, le réseau de neurones est très performant si on a bien choisi le nombre des layers, le nombre des neurones et les hyperparamètres pour la régularisation et dropout.

6.2.2 Inconvénients

Le réseau des neurones n'est pas nécessairement une approche pertinente pour notre tâche de classification. En plus, les résultats se fluctuent beaucoup même avec les mêmes données, hyperparamètres et machines.

6.3 Voting Classifier

Cette méthode considère tout les algorithmes pour réduire l'"overfitting" et avoir un plus petit écart-type. Il y a deux manière de l'utiliser : soft et hard.

Hard : prend en considération un vote majoritaire sans poids.

Soft : ce cas permet de se baser sur une probabilité pour sélectionner la meilleur classe.

Après l'essai des deux types, le "soft voting" performe mieux.

6.3.1 Avantages

- Peut prendre en entrée une grande sorte d'algorithmes pour les 2 types de voting et permet de bien généraliser la plupart du temps.

6.3.2 Inconvénients

- Un ou plusieurs algorithmes peuvent donner une influence qui falsifie les résultats si on ne fait pas attention aux choix de ceux-ci (i.e. on ne prends pas assez d'algorithmes, on en prends des trop similaires, etc.).

6.4 Stacking Classifier

Il est possible d'empiler des algorithmes, ici appelés des estimateurs pour ensuite passer le résultat à l'estimateur final qui procède à l'entraînement final.

6.4.1 Avantages

Il est possible d'avoir de meilleur résultat et un temps d'entraînenment moindre que Voting Classifier dans le cas où nous avons bien choisi et "tuned" nos algorithmes (estimateurs).

6.4.2 Inconvénients

- Avec une connaissance moins développée des algorithmes (estimateurs) utilisés pour la Stacking, il peut y avoir des difficultés à choisir le groupe d'estimateur et l'estimateur final pour maximiser le résultat.

7 Division des contributions

On a séparer les tâches de manière équitable à peu près.

Références

[1] https://www.coursera.org/learn/machine-learning

[2] ufldl tutorial Softmax Regression, http://ufldl.stanford.edu/tutorial/supervised/SoftmaxRegression/

[3] Logistic and Softmax Regression, https://houxianxu.github.io/2015/04/23/logistic-softmax-regression/

[4] Softmax Regression, https://zhuanlan.zhihu.com/p/98061179?utm_source=wechat_session&utm_medium=social&utm_oi=777418892074061824

[5] https://github.com/hankcs/CS224n/tree/master/assignment1

[6] https://github.com/hartikainen/stanford-cs224n/tree/master/assignment1

[7] Scikit-Learn Stacking Classifier, https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.StackingClassifier.html

[8] Scikit-Learn Voting Classifier, https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.VotingClassifier.html

[9] Keras API reference, https://keras.io/api/

[10] XGBoost Parameters, https://xgboost.readthedocs.io/en/latest/parameter.html

8 Annexe

Logistic Regression with Sigmoid

Reference

[1] Machine Learning by Andrew Ng (https://www.coursera.org/learn/machine-learning (https://www.coursera.org/learn/machine-learning))

In [63]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
```

In [64]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

In [65]:
```
data = pd.read_csv('/content/drive/My Drive/Competition/train.csv')
```

In [66]:
```
len_data = data.shape[0]
len_data
```

Out[66]:
```
47760
```

In [67]:
```
y_all_train = data.iloc[:, -1]
y_all_train
```

Out[67]:
```
0        0
1        1
2        0
3        0
4        0
       ..
47755    2
47756    2
47757    0
47758    0
47759    0
Name: LABELS, Length: 47760, dtype: int64
```

In [68]:
```
y_all_train.shape
```

Out[68]:
```
(47760,)
```

In [69]:
```
def label_percentages(labels):
  n0 = 0
  n1 = 0
  n2 = 0
  total = labels.shape[0]
  for label in labels:
    if label == 0:
      n0 += 1
    elif label == 1:
      n1 += 1
    elif label == 2:
      n2 += 1

  return (n0, n1, n2), (n0/total, n1/total, n2/total)
```

In [70]:
```
label_percentages(y_all_train)
```

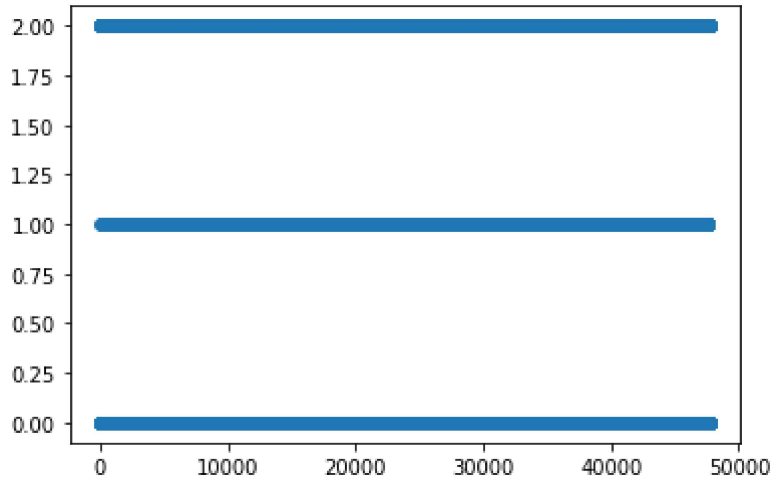
Out[70]:
```
((37535, 2002, 8223),
 (0.7859087102177554, 0.0419179229480737, 0.17217336683417087))
```

In [71]:
```
# distribution of labels
labels = y_all_train.copy().to_numpy()
labels.shape
```

Out[71]:
```
(47760,)
```

In [72]:
```
plt.scatter([i for i in range(labels.shape[0])], labels)
```

Out[72]:
```
<matplotlib.collections.PathCollection at 0x7fb4404cb890>
```



In [73]:
```
test = pd.read_csv('/content/drive/My Drive/Competition/test.csv')
```

In [74]:
```
all_features = pd.concat([data.iloc[:, :-1], test]).reset_index(drop=True)
```

In [75]:
```
all_features
```

Out[75]:

	S.No	lat	lon	TMQ	U850	V850	UBOT	VBOT	QRE
0	0	-24.758801	242.1875	16.019615	-4.391696	4.777769	-6.388222	7.725320	0.01(
1	1	23.820078	277.8125	47.802036	8.623652	9.308566	4.596105	9.938286	0.01{
2	2	23.820078	276.8750	11.556691	-2.483993	-6.009627	-3.503036	-5.921963	0.00{
3	3	13.494133	253.1250	53.186630	0.150933	-1.319407	3.757741	-2.172120	0.01{
4	4	-24.524120	241.2500	23.353998	-7.467506	-5.113565	-9.545109	-4.691221	0.01'
...	...	...	...	...	...	...	...	...	
55075	7315	24.054759	276.5625	51.415295	-1.095974	-8.194263	2.484773	-10.520496	0.02(
55076	7316	24.054759	276.8750	52.377407	-0.265653	-8.730537	3.783044	-10.748092	0.02'
55077	7317	24.054759	277.1875	54.639217	0.775797	-9.646189	5.087689	-10.786784	0.02'
55078	7318	24.054759	277.5000	56.121231	1.813888	-10.849813	6.442380	-10.859090	0.02'
55079	7319	24.054759	277.8125	56.888420	2.819084	-12.172881	7.671317	-10.641853	0.02{

55080 rows × 20 columns

Preprocessing data

(1) Remove non-significant features

(2) Standarlise data

(3) Add bias term

(4) Split train and test

In [ ]:
```
def preprocessing(features):
  X = features.copy()
  X = X.iloc[:, 1:19]
  X = X.drop(columns="PS")
  X.insert(0, 'bias', 1)
  X_means = np.mean(X)
  X_std = np.std(X)
  X_scale = (X - X_means) / X_std
  X_scale.iloc[:, 0] = np.ones((X_scale.shape[0], 1))
  return X_scale
```

In [ ]:
```
features_scale = preprocessing(all_features)
features_scale
```

Out[ ]:

	bias	lat	lon	TMQ	U850	V850	UBOT	VBOT	QREFH
0	1.0	-0.952799	-0.659826	-1.456660	-0.544970	1.082394	-0.689690	1.750971	-1.01902
1	1.0	1.167947	0.208099	0.980482	1.330311	2.126323	1.095283	2.212530	1.02801
2	1.0	1.167947	0.185259	-1.798887	-0.270104	-1.403102	-0.220842	-1.095445	-1.71571
3	1.0	0.717160	-0.393358	1.393384	0.109542	-0.322441	0.959048	-0.313340	1.09709
4	1.0	-0.942553	-0.682666	-0.894244	-0.988140	-1.196643	-1.202690	-0.838749	-0.67312
...	...	...	...	...	...	...	...	...	...
55075	1.0	1.178192	0.177646	1.257555	-0.070115	-1.906459	0.752188	-2.054562	1.62259
55076	1.0	1.178192	0.185259	1.331331	0.049520	-2.030020	0.963159	-2.102031	1.73687
55077	1.0	1.178192	0.192872	1.504772	0.199574	-2.240993	1.175167	-2.110102	1.82509
55078	1.0	1.178192	0.200486	1.618416	0.349145	-2.518317	1.395306	-2.125182	1.84868
55079	1.0	1.178192	0.208099	1.677245	0.493976	-2.823162	1.595011	-2.079873	1.94768

55080 rows × 18 columns

In [ ]:
```
# separate data and test data
train_data = features_scale.iloc[0:len_data, :]
test_data = features_scale.iloc[len_data:features_scale.shape[0], :]
train_data.shape, test_data.shape
```

Out[ ]:
```
((47760, 18), (7320, 18))
```

In [ ]:
```
train_data
```

Out[ ]:

	bias	lat	lon	TMQ	U850	V850	UBOT	VBOT	QREFH
0	1.0	-0.952799	-0.659826	-1.456660	-0.544970	1.082394	-0.689690	1.750971	-1.01902
1	1.0	1.167947	0.208099	0.980482	1.330311	2.126323	1.095283	2.212530	1.02801
2	1.0	1.167947	0.185259	-1.798887	-0.270104	-1.403102	-0.220842	-1.095445	-1.71571
3	1.0	0.717160	-0.393358	1.393384	0.109542	-0.322441	0.959048	-0.313340	1.09709
4	1.0	-0.942553	-0.682666	-0.894244	-0.988140	-1.196643	-1.202690	-0.838749	-0.67312
...	...	...	...	...	...	...	...	...	...
47755	1.0	-1.208927	2.050537	-1.559338	0.542162	0.960851	0.282271	0.865639	-1.50132
47756	1.0	-0.963044	-0.659826	-0.419844	1.410663	-0.103645	1.506244	-1.000442	-0.18382
47757	1.0	-1.208927	2.050537	-1.075436	1.198620	-1.466387	0.950071	-1.395281	-0.95668
47758	1.0	1.157701	0.177646	0.818513	-0.347110	0.046155	-0.779050	0.281748	1.55431
47759	1.0	1.178192	0.185259	1.114539	-0.284359	-0.212958	-0.678358	-0.274555	0.78549

47760 rows × 18 columns

In [ ]:
```
test_data
```

Out[ ]:

	bias	lat	lon	TMQ	U850	V850	UBOT	VBOT	QREFHT
47760	1.0	-1.229417	2.042923	-0.969527	-0.199640	-0.019019	-0.908190	-0.249263	-1.53332
47761	1.0	-1.229417	2.050537	-1.033650	-0.208858	0.073115	-1.022537	-0.056414	-1.61189
47762	1.0	-1.229417	2.058150	-1.115755	-0.218882	0.146943	-1.088501	0.171496	-1.65935
47763	1.0	-1.229417	2.065764	-1.153955	-0.218557	0.240390	-1.077335	0.400787	-1.70300
47764	1.0	-1.229417	2.073377	-1.182211	-0.208022	0.401727	-1.028380	0.603821	-1.75904
...	...	...	...	...	...	...	...	...	...
55075	1.0	1.178192	0.177646	1.257555	-0.070115	-1.906459	0.752188	-2.054562	1.62259
55076	1.0	1.178192	0.185259	1.331331	0.049520	-2.030020	0.963159	-2.102031	1.73687
55077	1.0	1.178192	0.192872	1.504772	0.199574	-2.240993	1.175167	-2.110102	1.82509
55078	1.0	1.178192	0.200486	1.618416	0.349145	-2.518317	1.395306	-2.125182	1.84868
55079	1.0	1.178192	0.208099	1.677245	0.493976	-2.823162	1.595011	-2.079873	1.94768

7320 rows × 18 columns

Logistic Regression Model

$$g(z) = \frac{1}{1 + e^{-z}}$$

In [ ]:
```
def sigmoid(z):
  return 1.0/(1 + np.exp(-z))
```

$$h_\theta(x) = g(\theta^T x),$$

$$X\theta = \begin{bmatrix} -(x^{(1)})^T\theta - \\ -(x^{(2)})^T\theta - \\ \vdots \\ -(x^{(m)})^T\theta - \end{bmatrix} = \begin{bmatrix} -\theta^T(x^{(1)}) - \\ -\theta^T(x^{(2)}) - \\ \vdots \\ -\theta^T(x^{(m)}) - \end{bmatrix}$$

Cost function

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left[-y^{(i)}\log(h_\theta(x^{(i)})) - (1 - y^{(i)})\log(1 - h_\theta(x^{(i)}))\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2,$$

Note that you should not regularize the parameter θ_0.

In [ ]:
```
def cost(X, y, theta, lambda_):
  # m: number of examples
  # n: number of features with bias term

  # X: (m, n), y (one col of y_onehot): (m, 1),
  # theta: (n, 1)
  m = X.shape[0]
  n = X.shape[1]

  h = sigmoid(np.dot(X, theta))
  # h: (m, nb_classes)

  log_h = np.log(h)  # (m, 1)
  log_one_minusH = np.log(1 - h) # (m, 1)

  yT = np.transpose(y)  # (1, m)
  one_minus_yT = np.transpose(1 - y)  # (1, m)

  sum_ = np.dot(-yT, log_h) - np.dot(one_minus_yT, log_one_minusH)
  # (1, m) * (m, 1) - (1, m) * (m, 1)

  theta_without_bias = theta[1:n]
  reg = (lambda_ / (2*m)) * np.sum(theta_without_bias ** 2)

  return sum_/ m + reg
```

Gradient

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j^{(i)} \qquad \text{for } j = 0,$$

$$\frac{\partial J(\theta)}{\partial \theta_j} = \left(\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j^{(i)}\right) + \frac{\lambda}{m}\theta_j \qquad \text{for } j \geq 1,$$

To vectorize this operation over the dataset, we start by writing out all the partial derivatives explicitly for all θ_j,

$$\begin{bmatrix} \frac{\partial J(\theta)}{\partial \theta_0} \\ \frac{\partial J(\theta)}{\partial \theta_1} \\ \frac{\partial J(\theta)}{\partial \theta_2} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_n} \end{bmatrix} = \frac{1}{m}\begin{bmatrix} \sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_0^{(i)} \\ \sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_1^{(i)} \\ \sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_2^{(i)} \\ \vdots \\ \sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_n^{(i)} \end{bmatrix} = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x^{(i)} = \frac{1}{m}X^T(h_\theta(x) - y) \quad (1)$$

In [ ]:
```
def gradient(X, y, theta, lambda_):
  m = X.shape[0]
  n = X.shape[1]
  h = sigmoid(np.dot(X, theta))

  sum_ = np.transpose(X).dot(h - y)  # (n, m) * (m, 1)
  gradient = sum_ / m

  theta_without_bias = theta[1:n]
  reg = lambda_ / m * theta_without_bias

  gradient[1:n] = gradient[1:n] + reg
  return gradient
```

In [ ]:
```
def gradient_descent(X, y, theta, lambda_, eps, alpha, max_iter): # alpha is l
earning rate
  losses = []
  i = 0
  print("Iteration: Cost")

  while(i < max_iter):
    i += 1
    grad = gradient(X, y, theta, lambda_)
    theta -= alpha * grad
    loss = cost(X, y, theta, lambda_)
    if (i % 1000 == 0):
      print("{}: {:.8f}".format(i, loss))

    len_losses = len(losses)
    if (len_losses == 0):
      diff = np.abs(loss)
    else :
      diff = np.abs(losses[len_losses-1] - loss)

    losses.append(loss)
    if(diff < eps):
      return theta, losses

  return theta, losses
```

Training Model

In [ ]:
```
y_all_train.shape
```

Out[ ]:
```
(47760,)
```

In [ ]:
```
def split_train_test(X, y, training_size, val_size):
    m = X.shape[0]
    nb_train = int(m * training_size)
    X_train = X.iloc[0:nb_train, :]
    y_train = y[0:nb_train]

    nb_val = int(m * val_size)

    val_index = nb_train + nb_val
    X_val = X.iloc[nb_train: val_index, :]
    y_val = y[nb_train: val_index]

    X_test = X.iloc[val_index: m, :]
    y_test = y[val_index: m]
    return X_train, y_train, X_val, y_val, X_test, y_test
```

In [ ]:
```
X_train, y_train, X_val, y_val, X_test, y_test = split_train_test(train_data,
y_all_train, 0.7, 0.15)
```

In [ ]:
```
X_train.shape, y_train.shape, X_val.shape, y_val.shape, X_test.shape, y_test.s
hape
```

Out[ ]:
```
((33432, 18), (33432,), (7164, 18), (7164,), (7164, 18), (7164,))
```

In [ ]:
```
def onehot_y(labels, classes):
    size = labels.shape[0]
    result = np.zeros((size, classes))
    for i in range(size):
        cl = int(labels[i])
        result[i][cl] = 1
    return result
```

In [ ]:
```
y_label = pd.Series.to_numpy(y_train.copy())
```

In [ ]:
```
y_onehot = onehot_y(y_label, 3)
y_onehot
```

Out[ ]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [1., 0., 0.],
       ...,
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.]])
```

In [ ]:
```
y_onehot.shape
```

Out[ ]:
```
(33432, 3)
```

In [ ]:
```
X_train_array = X_train.copy().to_numpy()
X_train_array
```

Out[ ]:
```
array([[ 1.        , -0.95279851, -0.65982622, ...,  1.42921582,
        -0.4285615 , -0.9506135 ],
       [ 1.        ,  1.16794656,  0.20809904, ..., -1.47779616,
         1.27702377,  1.39059963],
       [ 1.        ,  1.16794656,  0.1852589 , ...,  0.56987529,
        -1.43640373, -1.10560504],
       ...,
       [ 1.        ,  1.16794656,  0.20048566, ..., -0.34950255,
         0.18667299,  0.62839043],
       [ 1.        , -0.93230821, -0.67505298, ...,  1.42888277,
        -2.08529432, -1.45474775],
       [ 1.        ,  1.08598539, -0.60653257, ..., -0.99833767,
         0.85495341,  0.07425633]])
```

In [ ]:
```
# y_train: onehot of y
# lambda_: hyperparameter for regularization (or penalty)
# alpha: learning rate
# theta0: dim is (n, nb_classes) n is number of features including bias term
# return theta
# X, y, theta, lambda_, eps, alpha, max_iter, batch_size for sgd

def train(X_train, y_train, theta0, lambda_, eps, alpha, max_iter, nb_classes
):
  n = X_train.shape[1]  # number of features including bias term
  theta = np.zeros((n, 3))
  loss_dict = {}
  for i in range(nb_classes):
    print("Cost for {}th column of theta".format(i))
    losses = []
    theta[:, i], losses = gradient_descent(X_train,
                                           y_train[:, i],
                                           theta0[:, i],
                                           lambda_,
                                           eps,
                                           alpha,
                                           max_iter)
    loss_dict[i] = losses
  return theta, loss_dict
```

Hyperparameter Tuning

In [ ]:
```
def calculate_accuracy(X_test, y_test, theta):
  X_test_array = X_test.to_numpy()
  mat = X_test_array.dot(theta)
  y_pred = np.argmax(mat, axis=1)
  y_test_array = y_test.to_numpy()
  accuracy_rate = np.sum(y_test_array == y_pred) / y_test_array.shape[0]
  return accuracy_rate
```

In [ ]:
```
def hyperparameter_tuning(lambda_list, X_train, y_onehot, X_test, y_test, eps,
alpha, max_iter, nb_classes):
  n = X_train.shape[1]
  all_theta = {}
  all_losses = {}
  print("Hyperparameter tuning: Lambda")
  for each_lambda in lambda_list:
    theta0 = np.zeros((n, nb_classes))
    print(each_lambda)
    theta, loss_dict = train(X_train, y_onehot, theta0, each_lambda, eps, alph
a, max_iter, nb_classes)
    all_theta[each_lambda] = theta
    all_losses[each_lambda] = loss_dict
    accuracy = calculate_accuracy(X_test, y_test, theta)
    print("accuracy for lambda = {}: {:.8f}".format(each_lambda, accuracy))
    print("-------------------------------------------------")

  return all_theta, all_losses
```

In [ ]:
```
X_test.shape, y_test.shape
```

Out[ ]:
```
((7164, 18), (7164,))
```

Prediction

In [ ]:
```
test_data
```

Out[ ]:

	bias	lat	lon	TMQ	U850	V850	UBOT	VBOT	QREFHT
47760	1.0	-1.229417	2.042923	-0.969527	-0.199640	-0.019019	-0.908190	-0.249263	-1.53332
47761	1.0	-1.229417	2.050537	-1.033650	-0.208858	0.073115	-1.022537	-0.056414	-1.61189
47762	1.0	-1.229417	2.058150	-1.115755	-0.218882	0.146943	-1.088501	0.171496	-1.65935
47763	1.0	-1.229417	2.065764	-1.153955	-0.218557	0.240390	-1.077335	0.400787	-1.70300
47764	1.0	-1.229417	2.073377	-1.182211	-0.208022	0.401727	-1.028380	0.603821	-1.75904
...	...	...	...	...	...	...	...	...	...
55075	1.0	1.178192	0.177646	1.257555	-0.070115	-1.906459	0.752188	-2.054562	1.62259
55076	1.0	1.178192	0.185259	1.331331	0.049520	-2.030020	0.963159	-2.102031	1.73687
55077	1.0	1.178192	0.192872	1.504772	0.199574	-2.240993	1.175167	-2.110102	1.82509
55078	1.0	1.178192	0.200486	1.618416	0.349145	-2.518317	1.395306	-2.125182	1.84868
55079	1.0	1.178192	0.208099	1.677245	0.493976	-2.823162	1.595011	-2.079873	1.94768

7320 rows × 18 columns

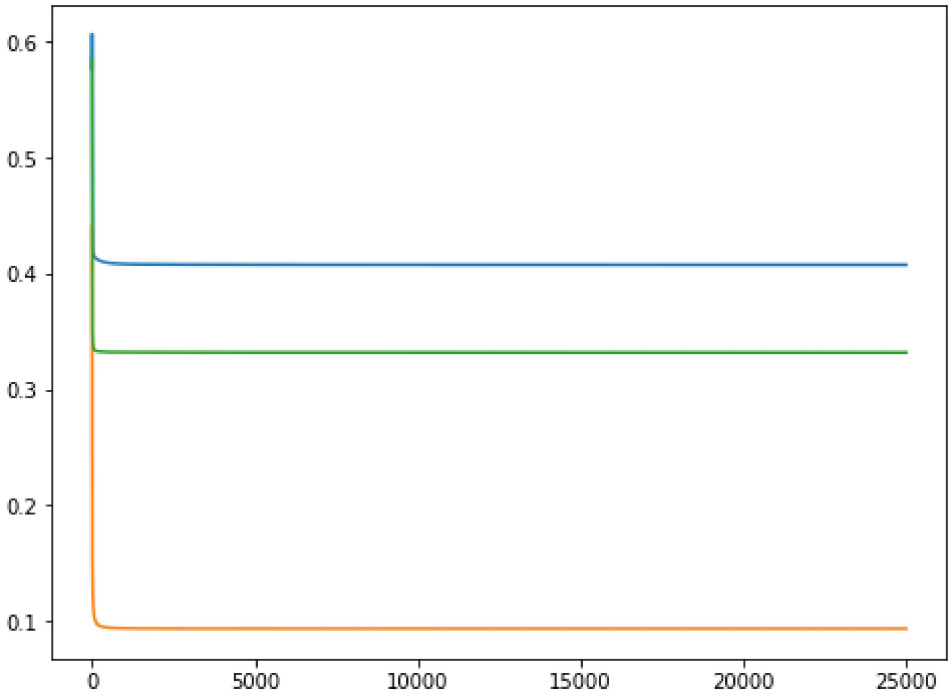
In [ ]:
```
def plot_loss(losses):
  plt.figure(figsize=(8, 6))
  for i in range(3):
    plt.plot([i for i in range(len(losses[i]))], losses[i])
  plt.show()
```

In [ ]:
```
n = test_data.shape[1]
nb_classes = 3
theta0 = np.zeros((n, nb_classes))

lambda_ = 3.5
eps = 10^-6
alpha = 1.5
max_iter = 25000
final_theta, loss_dict_final = train(X_train, y_onehot, theta0, lambda_, eps,
alpha, max_iter, nb_classes)
```

```
Cost for 0th column of theta
Iteration: Cost
1000: 0.40827548
2000: 0.40789532
3000: 0.40777280
4000: 0.40769939
5000: 0.40765096
6000: 0.40761863
7000: 0.40759702
8000: 0.40758257
9000: 0.40757292
10000: 0.40756646
11000: 0.40756215
12000: 0.40755927
13000: 0.40755734
14000: 0.40755605
15000: 0.40755519
16000: 0.40755461
17000: 0.40755423
18000: 0.40755397
19000: 0.40755380
20000: 0.40755369
21000: 0.40755361
22000: 0.40755356
23000: 0.40755352
24000: 0.40755350
25000: 0.40755349
Cost for 1th column of theta
Iteration: Cost
1000: 0.09352998
2000: 0.09325629
3000: 0.09320555
4000: 0.09319153
5000: 0.09318557
6000: 0.09318218
7000: 0.09317995
8000: 0.09317842
9000: 0.09317734
10000: 0.09317657
11000: 0.09317602
12000: 0.09317562
13000: 0.09317534
14000: 0.09317513
15000: 0.09317499
16000: 0.09317488
17000: 0.09317481
18000: 0.09317475
19000: 0.09317471
20000: 0.09317469
21000: 0.09317467
22000: 0.09317465
23000: 0.09317464
24000: 0.09317463
25000: 0.09317463
Cost for 2th column of theta
Iteration: Cost
1000: 0.33206051
```

```
2000: 0.33195964
3000: 0.33189458
4000: 0.33185035
5000: 0.33182025
6000: 0.33179977
7000: 0.33178583
8000: 0.33177634
9000: 0.33176989
10000: 0.33176549
11000: 0.33176250
12000: 0.33176047
13000: 0.33175909
14000: 0.33175814
15000: 0.33175750
16000: 0.33175707
17000: 0.33175677
18000: 0.33175657
19000: 0.33175643
20000: 0.33175634
21000: 0.33175627
22000: 0.33175623
23000: 0.33175620
24000: 0.33175618
25000: 0.33175617
```

In [ ]:
```
plot_loss(loss_dict_final)
```



In [ ]:
```
# accuracy on X_train
calculate_accuracy(X_train, y_train, final_theta)
```

Out[ ]:
```
0.8192151232352237
```

In [ ]:
```
# accuracy on split X_val
calculate_accuracy(X_val, y_val, final_theta)
```

Out[ ]: 0.8164433277498604

In [ ]:
```
# accuracy on split X_test
calculate_accuracy(X_test, y_test, final_theta)
```

Out[ ]: 0.8104410943606923

In [ ]:
```
mat_prob_test = test_data.dot(final_theta)
mat_prob_test
```

Out[ ]:

	0	1	2
47760	1.302737	-13.424771	-1.367839
47761	1.323159	-13.540540	-1.370454
47762	1.400580	-13.719122	-1.399481
47763	1.459107	-13.803240	-1.431322
47764	1.449182	-13.707072	-1.451464
...	...	...	...
55075	2.808538	-1.239371	-4.427883
55076	2.641542	-1.147226	-4.441608
55077	2.409571	-1.011491	-4.336809
55078	2.199825	-0.873350	-4.249391
55079	2.106112	-0.861428	-4.240929

7320 rows × 3 columns

In [ ]:
```
mat_prob_test_array = mat_prob_test.to_numpy()
mat_prob_test_array
```

Out[ ]:
```
array([[  1.30273674, -13.42477076,  -1.36783944],
       [  1.32315888, -13.54053973,  -1.37045396],
       [  1.40057968, -13.71912249,  -1.39948076],
       ...,
       [  2.40957123,  -1.0114905 ,  -4.33680941],
       [  2.19982524,  -0.87335026,  -4.24939078],
       [  2.10611199,  -0.86142788,  -4.24092889]])
```

In [ ]:
```
pred_test = np.argmax(mat_prob_test_array, axis=1)
pred_test
```

Out[ ]: array([0, 0, 0, ..., 0, 0, 0])

In [ ]:
```
label_percentages(pred_test)
```

Out[ ]:
```
((6534, 126, 660),
 (0.8926229508196721, 0.01721311475409836, 0.09016393442622951))
```

In [ ]:
```
submission = pd.read_csv('/content/drive/My Drive/Competition/sample_submission.csv')
submission
```

Out[ ]:

	S.No	LABELS
0	0	1
1	1	1
2	2	1
3	3	1
4	4	1
...	...	...
7315	7315	1
7316	7316	1
7317	7317	1
7318	7318	1
7319	7319	1

7320 rows × 2 columns

In [ ]:
```
submission.iloc[:,1] = pred_test
submission
```

Out[ ]:

	S.No	LABELS
0	0	0
1	1	0
2	2	0
3	3	0
4	4	0
...	...	...
7315	7315	0
7316	7316	0
7317	7317	0
7318	7318	0
7319	7319	0

7320 rows × 2 columns

In [ ]:
```
from google.colab import files
submission.to_csv('submission_sigmoid.csv', index=False)
files.download('submission_sigmoid.csv')
```

In [ ]:

Logistic Regression with Softmax

Reference:
[1] http://ufldl.stanford.edu/tutorial/supervised/SoftmaxRegression/ (http://ufldl.stanford.edu/tutorial/supervised/SoftmaxRegression/)

[2] https://houxianxu.github.io/2015/04/23/logistic-softmax-regression/ (https://houxianxu.github.io/2015/04/23/logistic-softmax-regression/)

[3] https://zhuanlan.zhihu.com/p/98061179?utm_source=wechat_session&utm_medium=social&utm_oi=777418892074061824 (https://zhuanlan.zhihu.com/p/98061179?utm_source=wechat_session&utm_medium=social&utm_oi=777418892074061824)

[4] https://github.com/hankcs/CS224n/tree/master/assignment1 (https://github.com/hankcs/CS224n/tree/master/assignment1)

[5] https://github.com/hartikainen/stanford-cs224n/tree/master/assignment1 (https://github.com/hartikainen/stanford-cs224n/tree/master/assignment1)

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

In [ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

In [ ]:
```
data = pd.read_csv('/content/drive/My Drive/Competition/train.csv')
```

In [ ]:
```
y_all_train = data.iloc[:, -1]
```

In [ ]:
```
y_all_train.shape
```

Out[ ]:
```
(47760,)
```

In [ ]:
```
def label_percentages(labels):
  n0 = 0
  n1 = 0
  n2 = 0
  total = labels.shape[0]
  for label in labels:
    if label == 0:
      n0 += 1
    elif label == 1:
      n1 += 1
    elif label == 2:
      n2 += 1

  return (n0, n1, n2), (n0/total, n1/total, n2/total), total
```

In [ ]:
```
label_percentages(y_all_train)
```

Out[ ]:
```
((37535, 2002, 8223),
 (0.7859087102177554, 0.0419179229480737, 0.17217336683417087),
 47760)
```

In [ ]:
```
test = pd.read_csv('/content/drive/My Drive/Competition/test.csv')
```

In [ ]:
```
all_features = pd.concat([data.iloc[:, :-1], test]).reset_index(drop=True)
```

In [ ]:
```
def preprocessing(features):
  X = features.copy()
  X = X.iloc[:, 1:19]
  X = X.drop(columns="PS")
  X = X.drop(columns="PRECT")
  X.insert(0, 'bias', 1)
  X_means = np.mean(X)
  X_std = np.std(X)
  X_scale = (X - X_means) / X_std
  X_scale.iloc[:, 0] = np.ones((X_scale.shape[0], 1))
  return X_scale
```

In [ ]:
```
features_scale = preprocessing(all_features)
features_scale
```

Out[ ]:

	bias	lat	lon	TMQ	U850	V850	UBOT	VBOT	QREFH
0	1.0	-0.952799	-0.659826	-1.456660	-0.544970	1.082394	-0.689690	1.750971	-1.01902
1	1.0	1.167947	0.208099	0.980482	1.330311	2.126323	1.095283	2.212530	1.02801
2	1.0	1.167947	0.185259	-1.798887	-0.270104	-1.403102	-0.220842	-1.095445	-1.71571
3	1.0	0.717160	-0.393358	1.393384	0.109542	-0.322441	0.959048	-0.313340	1.09709
4	1.0	-0.942553	-0.682666	-0.894244	-0.988140	-1.196643	-1.202690	-0.838749	-0.67312
...	...	...	...	...	...	...	...	...	...
55075	1.0	1.178192	0.177646	1.257555	-0.070115	-1.906459	0.752188	-2.054562	1.62259
55076	1.0	1.178192	0.185259	1.331331	0.049520	-2.030020	0.963159	-2.102031	1.73687
55077	1.0	1.178192	0.192872	1.504772	0.199574	-2.240993	1.175167	-2.110102	1.82509
55078	1.0	1.178192	0.200486	1.618416	0.349145	-2.518317	1.395306	-2.125182	1.84868
55079	1.0	1.178192	0.208099	1.677245	0.493976	-2.823162	1.595011	-2.079873	1.94768

55080 rows × 17 columns

In [ ]:
```
len_data = data.shape[0]
```

In [ ]:
```
# separate data and test data
train_data = features_scale.iloc[0:len_data, :]
test_data = features_scale.iloc[len_data:features_scale.shape[0], :]
train_data.shape, test_data.shape
```

Out[ ]:
```
((47760, 17), (7320, 17))
```

In [ ]:
```
train_data
```

Out[ ]:

	bias	lat	lon	TMQ	U850	V850	UBOT	VBOT	QREFH
0	1.0	-0.952799	-0.659826	-1.456660	-0.544970	1.082394	-0.689690	1.750971	-1.01902
1	1.0	1.167947	0.208099	0.980482	1.330311	2.126323	1.095283	2.212530	1.02801
2	1.0	1.167947	0.185259	-1.798887	-0.270104	-1.403102	-0.220842	-1.095445	-1.71571
3	1.0	0.717160	-0.393358	1.393384	0.109542	-0.322441	0.959048	-0.313340	1.09709
4	1.0	-0.942553	-0.682666	-0.894244	-0.988140	-1.196643	-1.202690	-0.838749	-0.67312
...	...	...	...	...	...	...	...	...	...
47755	1.0	-1.208927	2.050537	-1.559338	0.542162	0.960851	0.282271	0.865639	-1.50132
47756	1.0	-0.963044	-0.659826	-0.419844	1.410663	-0.103645	1.506244	-1.000442	-0.18382
47757	1.0	-1.208927	2.050537	-1.075436	1.198620	-1.466387	0.950071	-1.395281	-0.95668
47758	1.0	1.157701	0.177646	0.818513	-0.347110	0.046155	-0.779050	0.281748	1.55431
47759	1.0	1.178192	0.185259	1.114539	-0.284359	-0.212958	-0.678358	-0.274555	0.78549

47760 rows × 17 columns

In [ ]:
```
test_data
```

Out[ ]:

	bias	lat	lon	TMQ	U850	V850	UBOT	VBOT	QREFHT
47760	1.0	-1.229417	2.042923	-0.969527	-0.199640	-0.019019	-0.908190	-0.249263	-1.53332
47761	1.0	-1.229417	2.050537	-1.033650	-0.208858	0.073115	-1.022537	-0.056414	-1.61189
47762	1.0	-1.229417	2.058150	-1.115755	-0.218882	0.146943	-1.088501	0.171496	-1.65935
47763	1.0	-1.229417	2.065764	-1.153955	-0.218557	0.240390	-1.077335	0.400787	-1.70300
47764	1.0	-1.229417	2.073377	-1.182211	-0.208022	0.401727	-1.028380	0.603821	-1.75904
...	...	...	...	...	...	...	...	...	...
55075	1.0	1.178192	0.177646	1.257555	-0.070115	-1.906459	0.752188	-2.054562	1.62259
55076	1.0	1.178192	0.185259	1.331331	0.049520	-2.030020	0.963159	-2.102031	1.73687
55077	1.0	1.178192	0.192872	1.504772	0.199574	-2.240993	1.175167	-2.110102	1.82509
55078	1.0	1.178192	0.200486	1.618416	0.349145	-2.518317	1.395306	-2.125182	1.84868
55079	1.0	1.178192	0.208099	1.677245	0.493976	-2.823162	1.595011	-2.079873	1.94768

7320 rows × 17 columns

Logistic Regression with Softmax

Given a test input x, we want our hypothesis to estimate the probability that $P(y = k|x)$ for each value of $k = 1, \ldots, K$. I.e., we want to estimate the probability of the class label taking on each of the K different possible values. Thus, our hypothesis will output a K-dimensional vector (whose elements sum to 1) giving us our K estimated probabilities. Concretely, our hypothesis $h_\theta(x)$ takes the form:

$$h_\theta(x) = \begin{bmatrix} P(y = 1|x; \theta) \\ P(y = 2|x; \theta) \\ \vdots \\ P(y = K|x; \theta) \end{bmatrix} = \frac{1}{\sum_{j=1}^{K} \exp(\theta^{(j)\top} x)} \begin{bmatrix} \exp(\theta^{(1)\top} x) \\ \exp(\theta^{(2)\top} x) \\ \vdots \\ \exp(\theta^{(K)\top} x) \end{bmatrix}$$

Here $\theta^{(1)}, \theta^{(2)}, \ldots, \theta^{(K)} \in \Re^n$ are the parameters of our model. Notice that the term $\frac{1}{\sum_{j=1}^{K} \exp(\theta^{(j)\top} x)}$ normalizes the distribution, so that it sums to one.

(http://ufldl.stanford.edu/tutorial/supervised/SoftmaxRegression/ (http://ufldl.stanford.edu/tutorial/supervised/SoftmaxRegression/))

(5 points) Prove that softmax is invariant to constant offsets in the input, that is, for any input vector $\boldsymbol{x}$ and any constant c,

$$\text{softmax}(\boldsymbol{x}) = \text{softmax}(\boldsymbol{x} + c)$$

where $\boldsymbol{x} + c$ means adding the constant c to every dimension of $\boldsymbol{x}$. Remember that

$$softmax(\boldsymbol{x})_i = \frac{e^{\boldsymbol{x}_i}}{\sum_j e^{\boldsymbol{x}_j}} \tag{1}$$

Note: In practice, we make use of this property and choose $c = -\max_i x_i$ *when computing softmax probabilities for numerical stability (i.e., subtracting its maximum element from all elements of* $\boldsymbol{x}$*).*

(Standford CS 224n 2017W, assignement 1)

In [ ]:
```
# input product = X * theta
def softmax(product):
    if len(product.shape) > 1:
      max_each_row = np.max(product, axis=1, keepdims=True)
      exps = np.exp(product - max_each_row)
      sum_exps = np.sum(exps, axis=1, keepdims=True)
      res = exps / sum_exps

    else:
        product_max = np.max(product)
        product = product - product_max
        numerator = np.exp(product)
        denominator = 1.0 / np.sum(numerator)
        res = numerator.dot(denominator)
    return res
```

Cost Function

We now describe the cost function that we'll use for softmax regression. In the equation below, $1\{\cdot\}$ is the "indicator function," so that $1\{\text{a true statement}\} = 1$, and $1\{\text{a false statement}\} = 0$. For example, $1\{2+2=4\}$ evaluates to 1; whereas $1\{1+1=5\}$ evaluates to 0. Our cost function will be:

$$J(\theta) = -\left[\sum_{i=1}^{m}\sum_{k=1}^{K} 1\left\{y^{(i)}=k\right\}\log\frac{\exp(\theta^{(k)\top}x^{(i)})}{\sum_{j=1}^{K}\exp(\theta^{(j)\top}x^{(i)})}\right]$$

Notice that this generalizes the logistic regression cost function, which could also have been written:

$$\begin{aligned} J(\theta) &= -\left[\sum_{i=1}^{m}(1-y^{(i)})\log(1-h_\theta(x^{(i)})) + y^{(i)}\log h_\theta(x^{(i)})\right] \\ &= -\left[\sum_{i=1}^{m}\sum_{k=0}^{1} 1\left\{y^{(i)}=k\right\}\log P(y^{(i)}=k|x^{(i)};\theta)\right] \end{aligned}$$

The softmax cost function is similar, except that we now sum over the K different possible values of the class label. Note also that in softmax regression, we have that

$$P(y^{(i)}=k|x^{(i)};\theta) = \frac{\exp(\theta^{(k)\top}x^{(i)})}{\sum_{j=1}^{K}\exp(\theta^{(j)\top}x^{(i)})}$$

(http://ufldl.stanford.edu/tutorial/supervised/SoftmaxRegression/ (http://ufldl.stanford.edu/tutorial/supervised/SoftmaxRegression/))

Regularized Cost Function

$$\text{Cost} = -\frac{1}{m}\left\{\left[\sum_{i=1}^{m}\sum_{j=1}^{K} 1\{y_i=j\}\log\frac{e^{\theta_j^T x_i}}{\sum_{l=1}^{K} e^{\theta_l^T x_i}}\right] + \lambda\cdot\sum_{i=1}^{k}\sum_{j=1}^{n}\theta_{ij}^2\right\}$$

↳ softmax

m: number of samples n: number of features

k: number of classes

We should not regularize the θ_0

In [ ]:
```
def reg_cost_softmax(X, y_onehot, theta, lambda_):
  n_samples = X.shape[0]
  softmax_res = softmax(np.dot(X, theta.T))  # (n_samples, n_classes)
  cost = - (1.0 / n_samples) * np.sum(y_onehot * np.log(softmax_res))

  theta_without_bias = theta[:, 1:theta.shape[1]]
  reg = lambda_ / n_samples  * np.sum(theta_without_bias ** 2)
  return cost + reg
```

Gradient with L2 Regularization

$$\frac{\partial\, cost}{\partial \theta_j} = -\frac{1}{m}\left\{\left[\sum_{i=1}^{m} x_i \left(1\{y_i = j\} - P(y_i = j \mid x_i; \theta)\right)\right] - \lambda \cdot \theta_j\right\}$$

for $j \geq 1$.

$$\frac{\partial\, Cost}{\partial \theta} = -\frac{1}{m}\left[(y - p)^T X + \lambda\theta\right]$$

↳ softmax

We should not regularize the θ_0

In [ ]:
```
def reg_gradient_softmax(X, y_onehot, theta, lambda_):
  n_samples = X.shape[0]
  softmax_res = softmax(np.dot(X, theta.T))

  gradient = (-1.0 / n_samples) * np.dot((y_onehot - softmax_res).T, X)
  # (n_classes, n_features)

  theta_without_bias = theta[:, 1:theta.shape[1]]
  # theta: (n_classes, n_features)
  # n_feautres = X features + 1(bias term)
  # theta_without_bias: (n_classes, n_features - 1)
  reg = -lambda_ / n_samples * theta_without_bias

  gradient[:, 1:gradient.shape[1]] = gradient[:, 1:gradient.shape[1]] + reg

  return gradient
```

Gradient Descent

In [ ]:
```
# alpha is learning rate
def gradient_descent(X, y_onehot, theta, lambda_, eps, alpha, max_iter):
  losses = []
  i = 0
  print("Iteration: Cost")

  while(i < max_iter):
    i += 1
    grad = reg_gradient_softmax(X, y_onehot, theta, lambda_)
    theta -= alpha * grad

    loss = reg_cost_softmax(X, y_onehot, theta, lambda_)
    if (i % 1000 == 0):
      print("{}: {:.8f}".format(i, loss))

    len_losses = len(losses)
    if (len_losses == 0):
      print("{}: {:.8f}".format(i, loss))
      diff = np.abs(loss)
    else :
      diff = np.abs(losses[len_losses-1] - loss)

    losses.append(loss)
    if(diff < eps):
      return theta, losses

  return theta, losses
```

Trainining model

In [ ]:
```
y_all_train.shape
```

Out[ ]:
```
(47760,)
```

In [ ]:
```
def split_train_test(X, y, training_size, val_size):
  m = X.shape[0]
  nb_train = (int) (m * training_size)
  X_train = X.iloc[0:nb_train, :]
  y_train = y[0:nb_train]

  nb_val = (int) (m * val_size)

  val_index = nb_train + nb_val
  X_val = X.iloc[nb_train : val_index, :]
  y_val = y[nb_train : val_index]

  X_test = X.iloc[val_index : m, :]
  y_test = y[val_index : m]
  return X_train, y_train, X_val, y_val, X_test, y_test
```

In [ ]:
```
def onehot_y(labels, classes):
    size = labels.shape[0]
    result = np.zeros((size, classes))
    for i in range(size):
        cl = int(labels[i])
        result[i][cl] = 1
    return result
```

In [ ]:
```
X_train, y_train, X_val, y_val, X_test, y_test = split_train_test(train_data,
y_all_train, 0.8, 0.1)
```

In [ ]:
```
X_train.shape, y_train.shape, X_val.shape, y_val.shape, X_test.shape, y_test.s
hape
```

Out[ ]:
```
((38208, 17), (38208,), (4776, 17), (4776,), (4776, 17), (4776,))
```

In [ ]:
```
y_label = pd.Series.to_numpy(y_train.copy())
```

In [ ]:
```
y_onehot = onehot_y(y_label, 3)
y_onehot
```

Out[ ]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [1., 0., 0.],
       ...,
       [1., 0., 0.],
       [0., 1., 0.],
       [1., 0., 0.]])
```

In [ ]:
```
y_onehot.shape
```

Out[ ]:
```
(38208, 3)
```

In [ ]:
```
X_train_array = X_train.copy().to_numpy()
X_train_array
```

Out[ ]:
```
array([[ 1.        , -0.95279851, -0.65982622, ...,  1.42921582,
        -0.4285615 , -0.9506135 ],
       [ 1.        ,  1.16794656,  0.20809904, ..., -1.47779616,
         1.27702377,  1.39059963],
       [ 1.        ,  1.16794656,  0.1852589 , ...,  0.56987529,
        -1.43640373, -1.10560504],
       ...,
       [ 1.        ,  1.09623054, -0.61414595, ..., -0.33071481,
        -0.02934039, -0.88646481],
       [ 1.        ,  0.69666987, -0.38574456, ..., -1.14146989,
         0.86990277,  1.06717879],
       [ 1.        ,  0.70691502, -0.40097132, ..., -1.08116037,
         0.6735045 ,  0.990746  ]])
```

In [ ]:
```
X_train_array.shape
```

Out[ ]:
```
(38208, 17)
```

In [ ]:
```
# y_train: onehot of y
# lambda_: hyperparameter for regularization (or penalty)
# alpha: learning rate
# theta0: dim is (n, nb_classes) n is number of features including bias term
# return theta
# X, y, theta, lambda_, eps, alpha, max_iter, batch_size for sgd

def train(X_train, y_train, theta0, lambda_, eps, alpha, max_iter, nb_classes
):
  n_features = X_train.shape[1]  # number of features including bias term
  theta, losses = gradient_descent(X_train, y_train, theta0, lambda_, eps, alp
ha, max_iter)
  return theta, losses
```

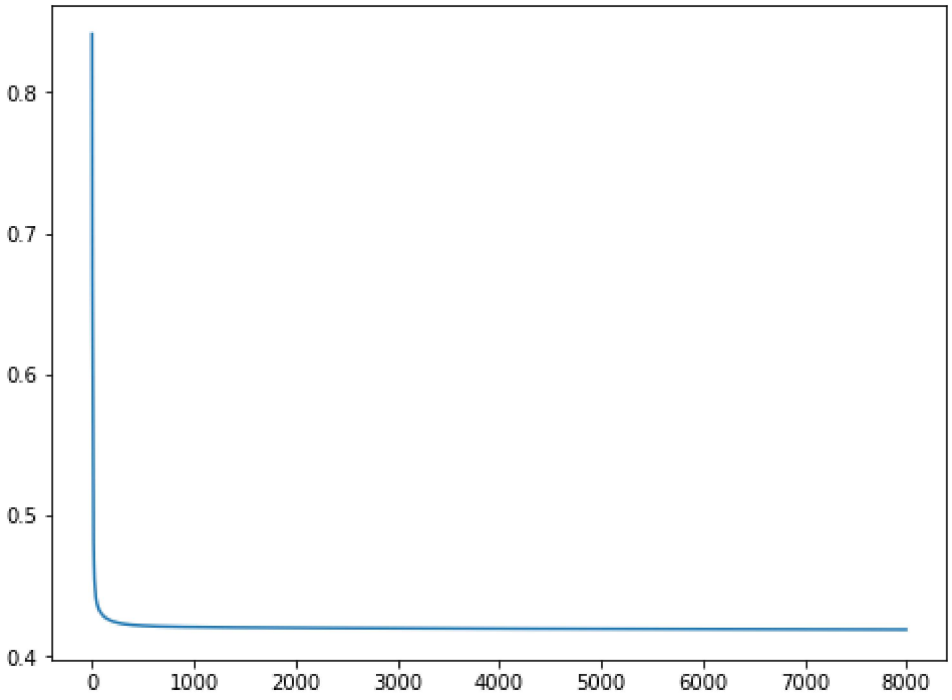
In [ ]:
```
def plot_loss(losses):
  plt.figure(figsize=(8, 6))
  plt.plot([i for i in range(len(losses))], losses)
  plt.show()
```

In [ ]:
```
theta0 = np.zeros((3, 17))
```

In [ ]:
```
# learning rate = 0.85 and lambda = 0
final_theta_0_85, losses_0_85 = train(X_train_array, y_onehot, theta0, 0, 10^-
6, 0.85, 8000, 3)
```

```
Iteration: Cost
1: 0.84122577
1000: 0.42055552
2000: 0.41996294
3000: 0.41963985
4000: 0.41939887
5000: 0.41919385
6000: 0.41901017
7000: 0.41884251
8000: 0.41868849
```

In [ ]:
```
plot_loss(losses_0_85)
```



In [ ]:
```
def calculate_accuracy(X_test, y_test, theta):
  X_test_array = X_test.to_numpy()
  mat = X_test_array.dot(theta.T)
  y_pred = np.argmax(mat, axis=1)
  y_test_array = y_test.to_numpy()
  accuracy_rate = np.sum(y_test_array == y_pred) / y_test_array.shape[0]
  return accuracy_rate
```

In [ ]:
```
accuracy_on_train = calculate_accuracy(X_train, y_train, final_theta_0_85)
accuracy_on_train
```

Out[ ]:
```
0.8234401172529313
```

In [ ]:
```
accuracy_on_val = calculate_accuracy(X_val, y_val, final_theta_0_85)
accuracy_on_val
```

Out[ ]: 0.8140703517587939

In [ ]:
```
accuracy_on_test = calculate_accuracy(X_test, y_test, final_theta_0_85)
accuracy_on_test
```

Out[ ]: 0.8134422110552764

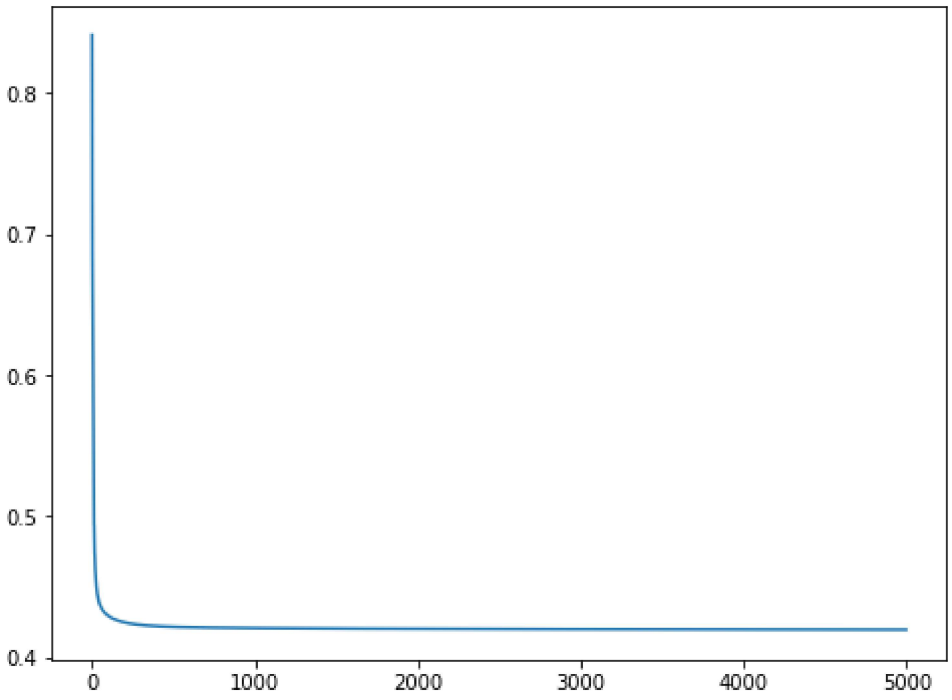
Hyperparameter Tuning

In [ ]:
```
def hyperparameter_tuning(lambda_list, X_train, y_onehot, X_test, y_test, eps,
alpha, max_iter, nb_classes):
  n = X_train.shape[1]
  all_theta = {}
  all_losses = {}
  print("Hyperparameter tuning: Lambda")
  for each_lambda in lambda_list:
    theta0 = np.zeros((3, 17))
    print(each_lambda)
    theta, loss_dict = train(X_train, y_onehot, theta0, each_lambda, eps, alph
a, max_iter, nb_classes)
    all_theta[each_lambda] = theta
    all_losses[each_lambda] = loss_dict
    accuracy = calculate_accuracy(X_test, y_test, theta)
    print("accuracy for lambda = {}: {:.8f}".format(each_lambda, accuracy))
    print("-----------------------------------------------")

  return all_theta, all_losses
```

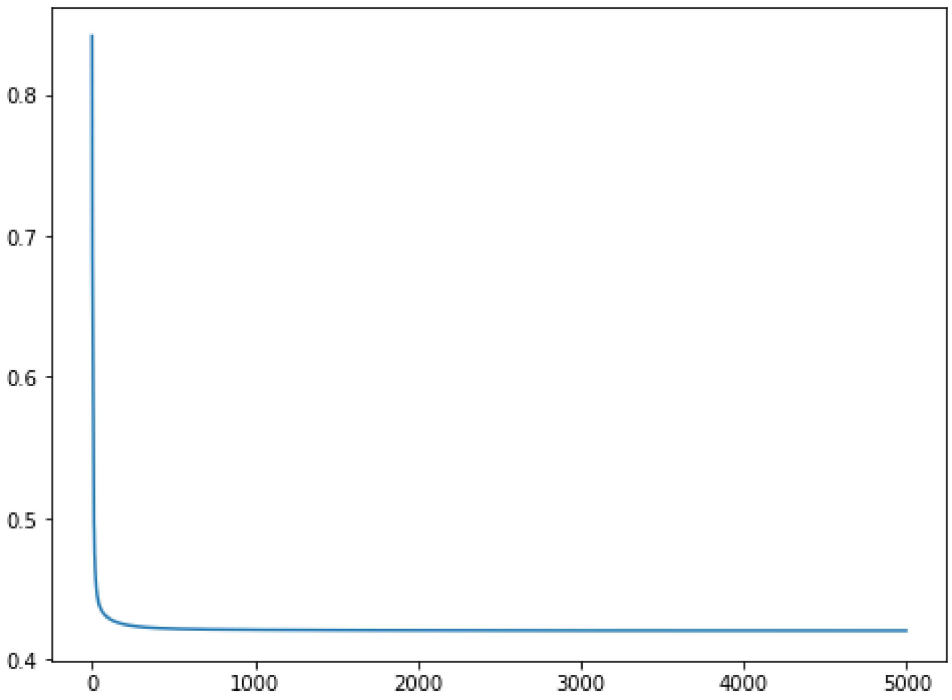
In [ ]:
```
all_theta, all_losses = hyperparameter_tuning([1, 3], X_train, y_onehot, X_val
, y_val, 10^-6, 0.85, 5000, 3)
```

```
Hyperparameter tuning: Lambda
1
Iteration: Cost
1: 0.84122942
1000: 0.42076321
2000: 0.42022391
3000: 0.41994593
4000: 0.41974874
5000: 0.41958812
accuracy for lambda = 1: 0.81344221
-------------------------------------------------
3
Iteration: Cost
1: 0.84123672
1000: 0.42119166
2000: 0.42077951
3000: 0.42062294
4000: 0.42055778
5000: 0.42054738
accuracy for lambda = 3: 0.81386097
-------------------------------------------------
```

In [ ]:
```
plot_loss(all_losses[1])
```



In [ ]:
```
plot_loss(all_losses[3])
```



In [ ]:
```
def check_accuracy(X_train, y_train, X_val, y_val, X_test, y_test, theta):
  train_acc = calculate_accuracy(X_train, y_train, theta)
  print("accuracy on X_train =  {:.8f}".format(train_acc))
  val_acc = calculate_accuracy(X_val, y_val, theta)
  print("accuracy on X_val = {:.8f}".format(val_acc))
  test_acc = calculate_accuracy(X_test, y_test, theta)
  print("accuracy on X_test = {:.8f}".format(test_acc))
```

In [ ]:
```
check_accuracy(X_train, y_train, X_val, y_val, X_test, y_test, all_theta[1])
```

```
accuracy on X_train =  0.82312605
accuracy on X_val = 0.81344221
accuracy on X_test = 0.81344221
```

In [ ]:
```
check_accuracy(X_train, y_train, X_val, y_val, X_test, y_test, all_theta[3])
```

```
accuracy on X_train =  0.82328308
accuracy on X_val = 0.81386097
accuracy on X_test = 0.81323283
```

Prediction

In [ ]:
```
test_data
```

Out[ ]:

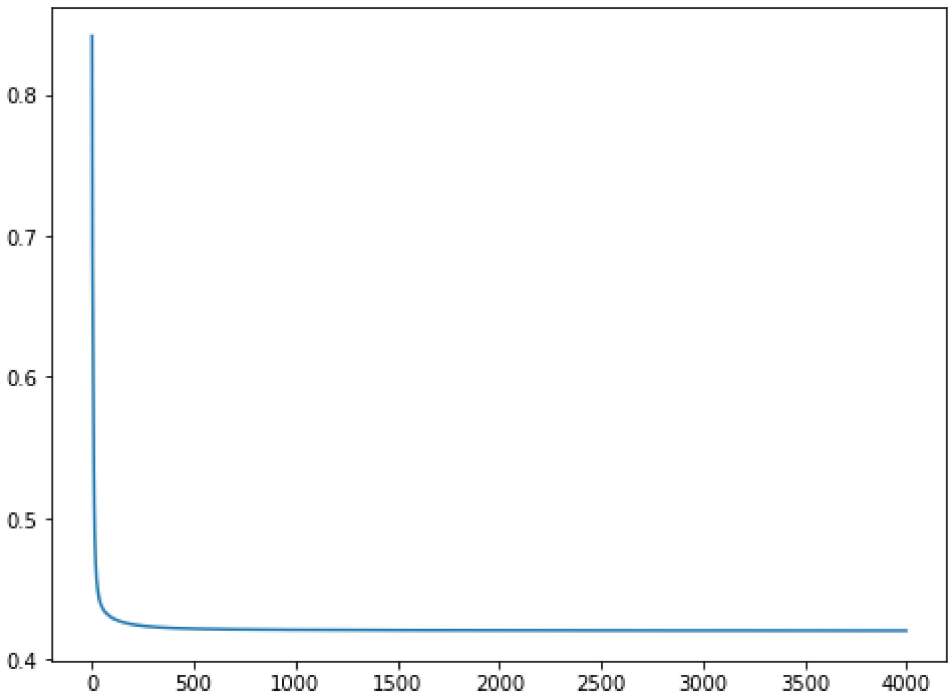
	bias	lat	lon	TMQ	U850	V850	UBOT	VBOT	QREFHT
47760	1.0	-1.229417	2.042923	-0.969527	-0.199640	-0.019019	-0.908190	-0.249263	-1.53332
47761	1.0	-1.229417	2.050537	-1.033650	-0.208858	0.073115	-1.022537	-0.056414	-1.61189
47762	1.0	-1.229417	2.058150	-1.115755	-0.218882	0.146943	-1.088501	0.171496	-1.65935
47763	1.0	-1.229417	2.065764	-1.153955	-0.218557	0.240390	-1.077335	0.400787	-1.70300
47764	1.0	-1.229417	2.073377	-1.182211	-0.208022	0.401727	-1.028380	0.603821	-1.75904
...	...	...	...	...	...	...	...	...	...
55075	1.0	1.178192	0.177646	1.257555	-0.070115	-1.906459	0.752188	-2.054562	1.62259
55076	1.0	1.178192	0.185259	1.331331	0.049520	-2.030020	0.963159	-2.102031	1.73687
55077	1.0	1.178192	0.192872	1.504772	0.199574	-2.240993	1.175167	-2.110102	1.82509
55078	1.0	1.178192	0.200486	1.618416	0.349145	-2.518317	1.395306	-2.125182	1.84868
55079	1.0	1.178192	0.208099	1.677245	0.493976	-2.823162	1.595011	-2.079873	1.94768

7320 rows × 17 columns

In [ ]:
```
n = test_data.shape[1]
nb_classes = 3
theta0 = np.zeros((3, 17))
# X_train, y_onehot, theta0, each_lambda, eps, alpha, max_iter, nb_classes
lambda_ = 3
eps = 10^-6
alpha = 0.85
max_iter = 4000
final_theta, loss_final = train(X_train, y_onehot, theta0, lambda_, eps, alpha
, max_iter, nb_classes)
```

```
Iteration: Cost
1: 0.84123672
1000: 0.42119166
2000: 0.42077951
3000: 0.42062294
4000: 0.42055778
```

In [ ]:
```
plot_loss(loss_final)
```



In [ ]:
```
# accuracy on X_train
calculate_accuracy(X_train, y_train, final_theta)
```

Out[ ]: 0.8234662897822446

In [ ]:
```
# accuracy on split X_val
calculate_accuracy(X_val, y_val, final_theta)
```

Out[ ]: 0.8138609715242882

In [ ]:
```
# accuracy on split X_test
calculate_accuracy(X_test, y_test, final_theta)
```

Out[ ]: 0.8138609715242882

In [ ]:
```
mat_prob_test = test_data.dot(final_theta.T)
mat_prob_test
```

Out[ ]:

	0	1	2
47760	4.556023	-7.667289	3.111266
47761	4.602816	-7.756571	3.153755
47762	4.688239	-7.892798	3.204560
47763	4.742801	-7.966166	3.223365
47764	4.720127	-7.910267	3.190139
...	...	...	...
55075	1.889559	0.603445	-2.493003
55076	1.822941	0.651805	-2.474746
55077	1.709969	0.691091	-2.401060
55078	1.600662	0.744508	-2.345170
55079	1.574041	0.721485	-2.295525

7320 rows × 3 columns

In [ ]:
```
mat_prob_test_array = mat_prob_test.to_numpy()
mat_prob_test_array
```

Out[ ]:
```
array([[ 4.55602338, -7.66728914,  3.11126576],
       [ 4.60281606, -7.75657117,  3.15375511],
       [ 4.68823894, -7.89279848,  3.20455953],
       ...,
       [ 1.70996931,  0.69109068, -2.40105999],
       [ 1.60066155,  0.74450838, -2.34516993],
       [ 1.57404053,  0.72148489, -2.29552542]])
```

In [ ]:
```
pred_test = np.argmax(mat_prob_test_array, axis=1)
pred_test
```

Out[ ]:
```
array([0, 0, 0, ..., 0, 0, 0])
```

In [ ]:
```
label_percentages(pred_test)
```

Out[ ]:
```
((6408, 209, 703),
 (0.8754098360655738, 0.028551912568306012, 0.09603825136612022),
 7320)
```

In [ ]:
```
submission = pd.read_csv('/content/drive/My Drive/Competition/sample_submission.csv')
submission
```

Out[ ]:

	S.No	LABELS
0	0	1
1	1	1
2	2	1
3	3	1
4	4	1
...	...	...
7315	7315	1
7316	7316	1
7317	7317	1
7318	7318	1
7319	7319	1

7320 rows × 2 columns

In [ ]:
```
submission.iloc[:,1] = pred_test
submission
```

Out[ ]:

	S.No	LABELS
0	0	0
1	1	0
2	2	0
3	3	0
4	4	0
...	...	...
7315	7315	0
7316	7316	0
7317	7317	0
7318	7318	0
7319	7319	0

7320 rows × 2 columns

In [ ]:
```
from google.colab import files
submission.to_csv('submission_pred.csv', index=False)
files.download('submission_pred.csv')
```

Other models:
(1) Neural Nets

(2) Soft Voting

(3) Stacking

Data Preparation

In [78]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call dri
ve.mount("/content/drive", force_remount=True).
```

In [79]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
warnings.filterwarnings('ignore')
from datetime import datetime
```

In [80]:
```
import sklearn
from sklearn.model_selection import cross_val_score, GridSearchCV, KFold
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, \
                             GradientBoostingClassifier, \
                             ExtraTreesClassifier, \
                             VotingClassifier, \
                             AdaBoostClassifier, \
                             StackingClassifier
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from lightgbm import LGBMClassifier
from xgboost import XGBClassifier
from sklearn.model_selection import RandomizedSearchCV, StratifiedKFold
```

In [81]:
```
train = pd.read_csv('/content/drive/My Drive/Competition/train.csv')
test = pd.read_csv('/content/drive/My Drive/Competition/test.csv')
sub = pd.read_csv('/content/drive/My Drive/Competition/sample_submission.csv')
```

Feature Engineering

In [82]:
```
def month(times):
  month = []
  for each_time in times:
    time_str = str(each_time)
    each_month = time_str[4:6]
    month.append(int(each_month))
  return month
```

In [83]:
```
def add_seasons(data, train):
  new_data = data.copy()
  month_list = month(train['time'])
  new_data['winter'] = [1 if (x == 1 or x == 2 or x == 12) else 0 for x in month_l
ist]
  new_data['summer'] = [1 if (x == 6 or x == 7 or x == 8) else 0 for x in month_li
st]
  new_data['automn'] = [1 if (x == 9 or x == 10 or x == 11) else 0 for x in month_
list]
  new_data['spring'] = [1 if (x == 3 or x == 4 or x == 5) else 0 for x in month_li
st]
  return new_data
```

Preprocessing

In [84]:
```
def preprocessing(features, removed_features):
    X = features.copy()
    X = X.drop(columns=removed_features)

    X_means = np.mean(X)
    X_std = np.std(X)
    X_scale = (X - X_means) / X_std

    return X_scale
```

In [85]:
```
def separate_train_test(train, data):
    train_data = data.iloc[0:train.shape[0], :]
    train_data_y = train.iloc[:, -1]
    test_data = data.iloc[train.shape[0]:data.shape[0], :]
    return train_data, train_data_y, test_data
```

In [86]:
```
def label_percentages(labels):
  n0 = 0
  n1 = 0
  n2 = 0
  total = labels.shape[0]
  for label in labels:
    if label == 0:
      n0 += 1
    elif label == 1:
      n1 += 1
    elif label == 2:
      n2 += 1

  return (n0, n1, n2), (n0/total, n1/total, n2/total), total
```

In [87]:
```
all_features = pd.concat([train.iloc[:, :-1], test]).reset_index(drop=True)
```

In [88]:
```
X_train = train.iloc[:, 0:train.shape[1]-1]
```

In [89]:
```
y_train = train.iloc[:, -1:]
```

In [90]:
```
clean_data = preprocessing(all_features, ['S.No', 'time', 'PS', 'PRECT', 'T200'])
clean_data_fea_imp = preprocessing(all_features, ['S.No', 'time', 'PS'])
```

In [91]:
```
clean_data_add_season = add_seasons(clean_data, all_features)
data_feature_importance = add_seasons(clean_data_fea_imp, all_features)
```

In [92]:
```
fea_eng_train, train_y, fea_eng_test = separate_train_test(train, clean_data_add_season)
```

In [93]:
```
data_for_feature_importance, test1, test2 = separate_train_test(train, data_feature_importance)
```

In [94]:
```
data_for_feature_importance
```

Out[94]:

	lat	lon	TMQ	U850	V850	UBOT	VBOT	QREFHT	PSL	T
0	-0.952799	-0.659826	-1.456660	-0.544970	1.082394	-0.689690	1.750971	-1.019028	1.385133	1.999
1	1.167947	0.208099	0.980482	1.330311	2.126323	1.095283	2.212530	1.028018	-1.364069	1.457
2	1.167947	0.185259	-1.798887	-0.270104	-1.403102	-0.220842	-1.095445	-1.715713	0.589421	-0.453
3	0.717160	-0.393358	1.393384	0.109542	-0.322441	0.959048	-0.313340	1.097092	-1.325220	0.640
4	-0.942553	-0.682666	-0.894244	-0.988140	-1.196643	-1.202690	-0.838749	-0.673120	1.796060	-1.153
...	...	...	...	...	...	...	...	...	...	
47755	-1.208927	2.050537	-1.559338	0.542162	0.960851	0.282271	0.865639	-1.501326	0.993904	-1.375
47756	-0.963044	-0.659826	-0.419844	1.410663	-0.103645	1.506244	-1.000442	-0.183821	0.316672	-1.348
47757	-1.208927	2.050537	-1.075436	1.198620	-1.466387	0.950071	-1.395281	-0.956686	0.436830	-0.253
47758	1.157701	0.177646	0.818513	-0.347110	0.046155	-0.779050	0.281748	1.554312	0.148711	0.020
47759	1.178192	0.185259	1.114539	-0.284359	-0.212958	-0.678358	-0.274555	0.785492	-0.333471	0.191

47760 rows × 21 columns

In [95]:
```
fea_eng_train
```

Out[95]:

	lat	lon	TMQ	U850	V850	UBOT	VBOT	QREFHT	PSL	T
0	-0.952799	-0.659826	-1.456660	-0.544970	1.082394	-0.689690	1.750971	-1.019028	1.385133	-0.706
1	1.167947	0.208099	0.980482	1.330311	2.126323	1.095283	2.212530	1.028018	-1.364069	1.322
2	1.167947	0.185259	-1.798887	-0.270104	-1.403102	-0.220842	-1.095445	-1.715713	0.589421	-1.322
3	0.717160	-0.393358	1.393384	0.109542	-0.322441	0.959048	-0.313340	1.097092	-1.325220	0.839
4	-0.942553	-0.682666	-0.894244	-0.988140	-1.196643	-1.202690	-0.838749	-0.673120	1.796060	-0.218
...	...	...	...	...	...	...	...	...	...	
47755	-1.208927	2.050537	-1.559338	0.542162	0.960851	0.282271	0.865639	-1.501326	0.993904	-1.257
47756	-0.963044	-0.659826	-0.419844	1.410663	-0.103645	1.506244	-1.000442	-0.183821	0.316672	-0.101
47757	-1.208927	2.050537	-1.075436	1.198620	-1.466387	0.950071	-1.395281	-0.956686	0.436830	-0.046
47758	1.157701	0.177646	0.818513	-0.347110	0.046155	-0.779050	0.281748	1.554312	0.148711	0.395
47759	1.178192	0.185259	1.114539	-0.284359	-0.212958	-0.678358	-0.274555	0.785492	-0.333471	0.271

47760 rows × 19 columns

In [96]:
```
train_y
```

Out[96]:
```
0         0
1         1
2         0
3         0
4         0
         ..
47755     2
47756     2
47757     0
47758     0
47759     0
Name: LABELS, Length: 47760, dtype: int64
```

In [97]:
```
fea_eng_test
```

Out[97]:

	lat	lon	TMQ	U850	V850	UBOT	VBOT	QREFHT	PSL	T5
47760	-1.229417	2.042923	-0.969527	-0.199640	-0.019019	-0.908190	-0.249263	-1.533322	2.257725	-1.7338
47761	-1.229417	2.050537	-1.033650	-0.208858	0.073115	-1.022537	-0.056414	-1.611895	2.270233	-1.7141
47762	-1.229417	2.058150	-1.115755	-0.218882	0.146943	-1.088501	0.171496	-1.659356	2.290553	-1.7040
47763	-1.229417	2.065764	-1.153955	-0.218557	0.240390	-1.077335	0.400787	-1.703007	2.295390	-1.6781
47764	-1.229417	2.073377	-1.182211	-0.208022	0.401727	-1.028380	0.603821	-1.759042	2.283319	-1.6284
...	...	...	...	...	...	...	...	...	...	...
55075	1.178192	0.177646	1.257555	-0.070115	-1.906459	0.752188	-2.054562	1.622598	-1.471042	1.2591
55076	1.178192	0.185259	1.331331	0.049520	-2.030020	0.963159	-2.102031	1.736878	-1.543805	1.3161
55077	1.178192	0.192872	1.504772	0.199574	-2.240993	1.175167	-2.110102	1.825097	-1.597136	1.3776
55078	1.178192	0.200486	1.618416	0.349145	-2.518317	1.395306	-2.125182	1.848681	-1.645236	1.4641
55079	1.178192	0.208099	1.677245	0.493976	-2.823162	1.595011	-2.079873	1.947685	-1.692151	1.5543

7320 rows × 19 columns

Neural Nets

In [98]:
```
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
from keras.utils import np_utils
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.preprocessing import LabelEncoder
from sklearn.pipeline import Pipeline
from sklearn.metrics import accuracy_score
```

In [99]:
```
def onehot_labels(labels):
  # encode class values as integers
  encoder = LabelEncoder()
  encoder.fit(labels)
  encoded_Y = encoder.transform(labels)
  # convert integers to dummy variables (i.e. one hot encoded)
  dummy_y = np_utils.to_categorical(encoded_Y)
  return dummy_y
```

In [100]:
```
y_onehot = onehot_labels(train_y)
y_onehot.shape
```

Out[100]:
```
(47760, 3)
```

In [101]:
```
from sklearn.model_selection import train_test_split
X_train_nn, X_val_nn, y_train_nn, y_val_nn = train_test_split(fea_eng_train,
                                                              y_onehot,
                                                              test_size=0.3,
                                                              random_state=42)

X_train_nn.shape, X_val_nn.shape, y_train_nn.shape, y_val_nn.shape
```

Out[101]:
```
((33432, 19), (14328, 19), (33432, 3), (14328, 3))
```

In [102]:
```
from tensorflow.keras import regularizers
from keras.layers import Dropout
# create model
model = Sequential()
model.add(Dense(14, input_dim=19, activation='relu'))
model.add(Dense(3, activation='softmax', kernel_regularizer='l2'))
# Compile model
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accurac
y'])
```

In [103]:
```
training_result = model.fit(X_train_nn,
                            y_train_nn,
                            batch_size=10,
                            epochs=50,
                            validation_data=(X_val_nn, y_val_nn))
```

```
Epoch 1/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.5206 - accurac
y: 0.7976 - val_loss: 0.4302 - val_accuracy: 0.8234
Epoch 2/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.4223 - accurac
y: 0.8278 - val_loss: 0.4076 - val_accuracy: 0.8347
Epoch 3/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.4087 - accurac
y: 0.8303 - val_loss: 0.4010 - val_accuracy: 0.8368
Epoch 4/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.4008 - accurac
y: 0.8325 - val_loss: 0.3943 - val_accuracy: 0.8381
Epoch 5/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3959 - accurac
y: 0.8340 - val_loss: 0.3865 - val_accuracy: 0.8412
Epoch 6/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3917 - accurac
y: 0.8351 - val_loss: 0.3869 - val_accuracy: 0.8412
Epoch 7/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3886 - accurac
y: 0.8355 - val_loss: 0.3851 - val_accuracy: 0.8386
Epoch 8/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3856 - accurac
y: 0.8368 - val_loss: 0.3790 - val_accuracy: 0.8414
Epoch 9/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3838 - accurac
y: 0.8369 - val_loss: 0.3772 - val_accuracy: 0.8428
Epoch 10/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3817 - accurac
y: 0.8381 - val_loss: 0.3816 - val_accuracy: 0.8425
Epoch 11/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3804 - accurac
y: 0.8381 - val_loss: 0.3767 - val_accuracy: 0.8418
Epoch 12/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3789 - accurac
y: 0.8392 - val_loss: 0.3764 - val_accuracy: 0.8437
Epoch 13/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3775 - accurac
y: 0.8388 - val_loss: 0.3733 - val_accuracy: 0.8419
Epoch 14/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3766 - accurac
y: 0.8402 - val_loss: 0.3705 - val_accuracy: 0.8450
Epoch 15/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3758 - accurac
y: 0.8402 - val_loss: 0.3726 - val_accuracy: 0.8432
Epoch 16/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3746 - accurac
y: 0.8398 - val_loss: 0.3693 - val_accuracy: 0.8428
Epoch 17/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3740 - accurac
y: 0.8402 - val_loss: 0.3693 - val_accuracy: 0.8426
Epoch 18/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3729 - accurac
y: 0.8414 - val_loss: 0.3700 - val_accuracy: 0.8467
Epoch 19/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3721 - accurac
y: 0.8409 - val_loss: 0.3723 - val_accuracy: 0.8445
Epoch 20/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3710 - accurac
y: 0.8412 - val_loss: 0.3658 - val_accuracy: 0.8475
Epoch 21/50
3344/3344 [==============================] - 5s 2ms/step - loss: 0.3703 - accurac
y: 0.8432 - val_loss: 0.3689 - val_accuracy: 0.8464
```

```
Epoch 22/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3699 - accurac
y: 0.8425 - val_loss: 0.3681 - val_accuracy: 0.8446
Epoch 23/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3690 - accurac
y: 0.8414 - val_loss: 0.3677 - val_accuracy: 0.8449
Epoch 24/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3684 - accurac
y: 0.8426 - val_loss: 0.3644 - val_accuracy: 0.8448
Epoch 25/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3681 - accurac
y: 0.8433 - val_loss: 0.3638 - val_accuracy: 0.8485
Epoch 26/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3668 - accurac
y: 0.8437 - val_loss: 0.3666 - val_accuracy: 0.8444
Epoch 27/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3671 - accurac
y: 0.8440 - val_loss: 0.3610 - val_accuracy: 0.8476
Epoch 28/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3659 - accurac
y: 0.8438 - val_loss: 0.3634 - val_accuracy: 0.8463
Epoch 29/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3655 - accurac
y: 0.8448 - val_loss: 0.3614 - val_accuracy: 0.8497
Epoch 30/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3652 - accurac
y: 0.8439 - val_loss: 0.3607 - val_accuracy: 0.8489
Epoch 31/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3650 - accurac
y: 0.8439 - val_loss: 0.3593 - val_accuracy: 0.8499
Epoch 32/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3647 - accurac
y: 0.8456 - val_loss: 0.3580 - val_accuracy: 0.8474
Epoch 33/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3635 - accurac
y: 0.8460 - val_loss: 0.3584 - val_accuracy: 0.8518
Epoch 34/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3631 - accurac
y: 0.8472 - val_loss: 0.3641 - val_accuracy: 0.8465
Epoch 35/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3629 - accurac
y: 0.8464 - val_loss: 0.3590 - val_accuracy: 0.8469
Epoch 36/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3626 - accurac
y: 0.8473 - val_loss: 0.3626 - val_accuracy: 0.8484
Epoch 37/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3619 - accurac
y: 0.8465 - val_loss: 0.3568 - val_accuracy: 0.8520
Epoch 38/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3613 - accurac
y: 0.8486 - val_loss: 0.3561 - val_accuracy: 0.8502
Epoch 39/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3610 - accurac
y: 0.8478 - val_loss: 0.3563 - val_accuracy: 0.8506
Epoch 40/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3605 - accurac
y: 0.8482 - val_loss: 0.3561 - val_accuracy: 0.8510
Epoch 41/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3597 - accurac
y: 0.8485 - val_loss: 0.3578 - val_accuracy: 0.8497
Epoch 42/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3597 - accurac
y: 0.8480 - val_loss: 0.3533 - val_accuracy: 0.8509
Epoch 43/50
```

```
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3592 - accurac
y: 0.8480 - val_loss: 0.3540 - val_accuracy: 0.8509
Epoch 44/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3591 - accurac
y: 0.8476 - val_loss: 0.3538 - val_accuracy: 0.8520
Epoch 45/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3583 - accurac
y: 0.8495 - val_loss: 0.3544 - val_accuracy: 0.8534
Epoch 46/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3579 - accurac
y: 0.8475 - val_loss: 0.3597 - val_accuracy: 0.8474
Epoch 47/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3576 - accurac
y: 0.8473 - val_loss: 0.3544 - val_accuracy: 0.8536
Epoch 48/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3576 - accurac
y: 0.8490 - val_loss: 0.3517 - val_accuracy: 0.8527
Epoch 49/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3570 - accurac
y: 0.8483 - val_loss: 0.3547 - val_accuracy: 0.8492
Epoch 50/50
3344/3344 [==============================] - 6s 2ms/step - loss: 0.3567 - accurac
y: 0.8491 - val_loss: 0.3570 - val_accuracy: 0.8513
```

In [104]:
```
model.evaluate(X_train_nn, y_train_nn)
```

```
1045/1045 [==============================] - 1s 1ms/step - loss: 0.3590 - accurac
y: 0.8474
```

Out[104]:
```
[0.35898593068122864, 0.8473917245864868]
```

In [105]:
```
fea_eng_test.shape
```

Out[105]:
```
(7320, 19)
```

In [106]:
```
y_mat_prob = model.predict(fea_eng_test)
```

In [107]:
```
y_pred_nn = np.argmax(y_mat_prob, axis=1)
y_pred_nn.shape
```

Out[107]:
```
(7320,)
```

In [108]:
```
y_pred_nn
```

Out[108]:
```
array([0, 0, 0, ..., 1, 1, 1])
```

In [109]:
```
label_percentages(y_pred_nn)
```

Out[109]:
```
((5547, 449, 1324),
 (0.7577868852459017, 0.06133879781420765, 0.1808743169398907),
 7320)
```

Soft Voting

In [ ]:
```
kfold = KFold(n_splits=5, random_state=42, shuffle=True)
```

In [ ]:
```
def accuracy_percent(model, features, label, kf):
    accuracy = cross_val_score(model, features, label, scoring='accuracy', cv=kf)
    return accuracy
```

In [ ]:
```
from sklearn.pipeline import Pipeline, make_pipeline
```

In [ ]:
```
models = [LogisticRegression(C=0.1, solver='newton-cg'),
          SVC(C=0.1, random_state=42),
          DecisionTreeClassifier(),
          KNeighborsClassifier(n_neighbors=9),
          RandomForestClassifier(),
          GradientBoostingClassifier(),
          ExtraTreesClassifier(),
          LGBMClassifier(),
          XGBClassifier(objective='multi:softprob', eval_metric='merror'),
         ]
```

In [ ]:
```
scores = {}
```

In [ ]:
```
names = ["Logistic Regression",
         "SVM",
         "Decision Tree Classifier",
         "KNN",
         "Random Forest Classifier",
         "Gradient Boosting Classifier",
         "Extra Trees Classifier",
         "Light Gradient Boosting Machine",
         "eXtreme Gradient Boosting"
         ]
```

In [ ]:
```
for name, model in zip(names, models):
    score = accuracy_percent(model, fea_eng_train, train_y, kfold)
    print("{}: {:.6f}, {:.4f}".format(name, score.mean(), score.std()))
    scores[name] = (score.mean(), score.std())
```

```
Logistic Regression: 0.821001, 0.0034
SVM: 0.844116, 0.0033
Decision Tree Classifier: 0.855653, 0.0025
KNN: 0.889175, 0.0027
Random Forest Classifier: 0.859883, 0.0029
Gradient Boosting Classifier: 0.863714, 0.0027
Extra Trees Classifier: 0.861642, 0.0017
Light Gradient Boosting Machine: 0.889363, 0.0031
eXtreme Gradient Boosting: 0.852931, 0.0034
```

In [ ]:
```
scores
```

Out[ ]:
```
{'DecisionTreeClassifier': (0.8562395309882748, 0.0023931711997034174),
 'ExtraTreesClassifier': (0.8617043551088777, 0.0016566198027514623),
 'GradientBoostingClassifier': (0.863714405360134, 0.0027028709485938146),
 'KNN': (0.889175041876047, 0.002669577462396826),
 'LGBM': (0.8893634840871021, 0.003108009620394584),
 'LogisticRegression': (0.821000837520938, 0.0033990609539782967),
 'RandomForestClassifier': (0.8602805695142377, 0.00253133689442334),
 'SVM': (0.8441164154103852, 0.003281329652288071),
 'XGB': (0.852931323283082, 0.003350999665414375)}
```

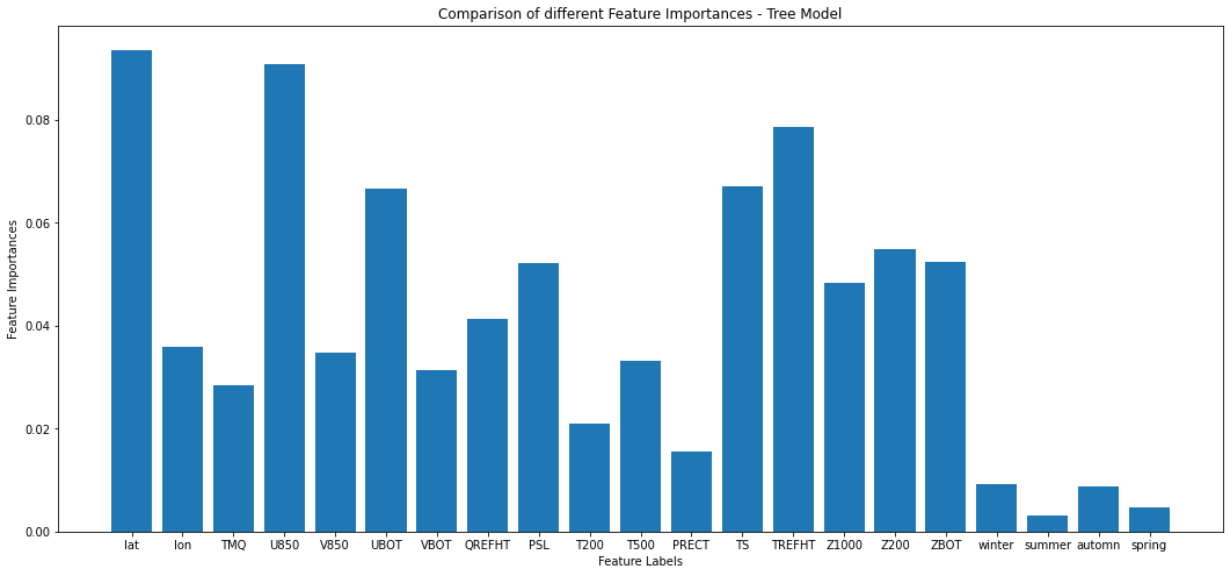
In [ ]:
```
final_models = [
          ('lr', LogisticRegression(solver='newton-cg')),
          ('dt', DecisionTreeClassifier()),
          ('rf', RandomForestClassifier()),
          ('gb', GradientBoostingClassifier()),
          ('etc', ExtraTreesClassifier()),
          ('svc', SVC(probability=True)),
          ('lgbm', LGBMClassifier()),
          ('xgb', XGBClassifier())
          ]
```

In [ ]:
```
eclfsoft = VotingClassifier(estimators=final_models, voting='soft')
eclfsoft = eclfsoft.fit(fea_eng_train, train_y)
```

In [ ]:
```
labels_soft = eclfsoft.predict(fea_eng_test)
```

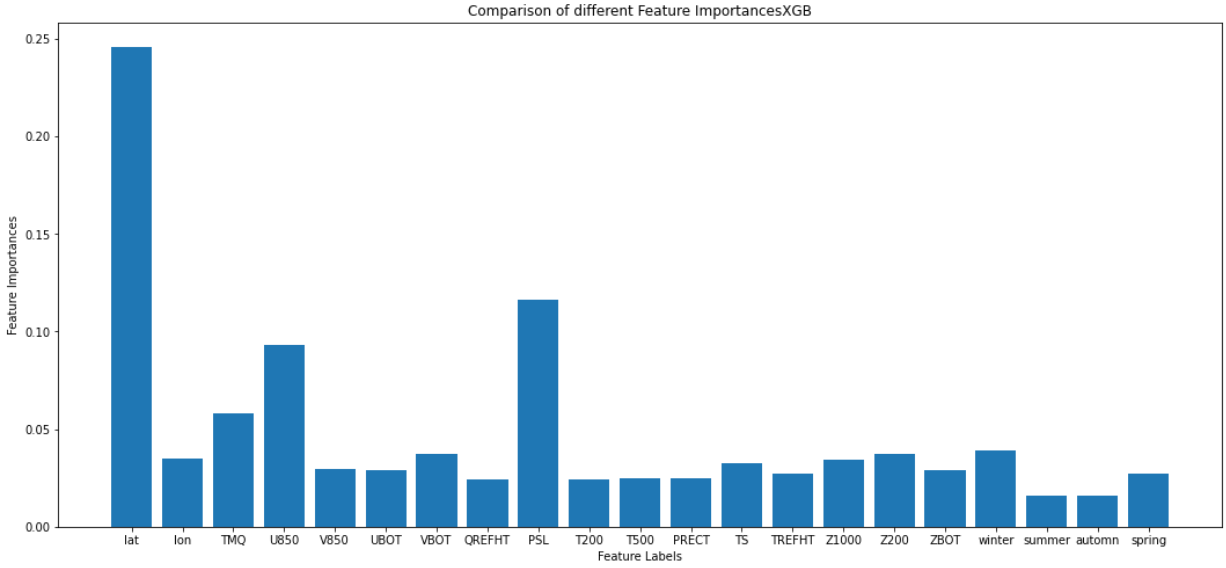
In [112]:
```
def plot_feature_importance_tree_model(X, y, model):
    # Training the model
    model.fit(X, y)

    # Computing the importance of each feature
    feature_importance = model.feature_importances_

    # Normalizing the individual importances
    feature_importance_normalized = np.std([tree.feature_importances_ for tree in
                                            model.estimators_],
                                            axis = 0)

    plt.figure(figsize=(18,8))
    plt.bar(X.columns, feature_importance_normalized)
    plt.xlabel('Feature Labels')
    plt.ylabel('Feature Importances')
    plt.title('Comparison of different Feature Importances - Tree Model')
    plt.show()
```

In [113]:
```
plot_feature_importance_tree_model(data_for_feature_importance, y_train, model=Ran
domForestClassifier(max_depth=7))
```

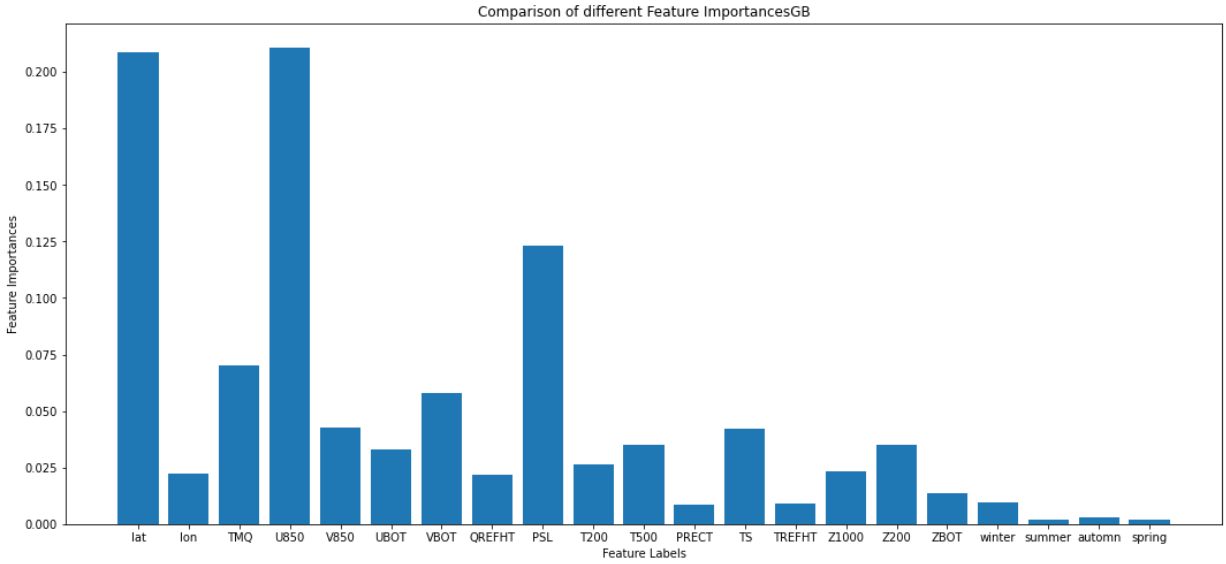


In [116]:
```
def plot_feature_importance_boost_model(X, y, model, algo_type):
  model.fit(X, y)
  plt.figure(figsize=(18,8))
  plt.bar(X.columns, model.feature_importances_)
  plt.xlabel('Feature Labels')
  plt.ylabel('Feature Importances')
  plt.title('Comparison of different Feature Importances' + algo_type)
  plt.show()
```

In [117]:
```
plot_feature_importance_boost_model(data_for_feature_importance, train_y, XGBClass
ifier(
    objective='multi:softmax',
    eval_metric='merror',
    max_depth=4), 'XGB')
```



In [118]:
```
plot_feature_importance_boost_model(data_for_feature_importance, train_y, Gradient
BoostingClassifier(), 'GB')
```



Stacking Classifier

Hyperparameter tuning

In [ ]:
```
c_list = [0.01, 0.05, 0.1, 0.2, 0.33, 0.5, 1]
grid={"C":c_list}
logreg=LogisticRegression()
logreg_cv=GridSearchCV(logreg, grid, cv=10)
logreg_cv.fit(fea_eng_train,train_y)

print("tuned hyperparameters :(best parameters) ", logreg_cv.best_params_)
print("accuracy :", logreg_cv.best_score_)
```

```
tuned hyperparameters :(best parameters)  {'C': 0.1}
accuracy : 0.8217336683417085
```

In [ ]:
```
max_depth_list = [4, 5, 6, 7, 8, 9, 10]
grid={"max_depth": max_depth_list}

rf_clf = RandomForestClassifier()
rf_cv = GridSearchCV(rf_clf, grid, cv=10)
rf_cv.fit(fea_eng_train, train_y)

print("tuned hyperparameters :(best parameters) ", rf_cv.best_params_)
print("accuracy :", rf_cv.best_score_)
```

```
tuned hyperparameters :(best parameters)  {'max_depth': 10}
accuracy : 0.8793760469011724
```

In [ ]:
```
# https://xgboost.readthedocs.io/en/latest/parameter.html
params_xgb = {
        # Increasing this value will make the model more complex and more likely t
o overfit
        'max_depth': [3, 4, 5],
        'learning_rate': [0.02, 0.05, 0.1, 0.3],
        'n_estimators' : [10, 50, 100]
        }

xgb_clf = XGBClassifier()
xgb_cv=GridSearchCV(xgb_clf, params_xgb, cv=3)
xgb_cv.fit(fea_eng_train, train_y)
print("tuned hyperparameters :(best parameters) ", xgb_cv.best_params_)
print("accuracy :", xgb_cv.best_score_)
```

```
tuned hyperparameters :(best parameters)  {'colsample_bytree': 0.5, 'max_depth':
5, 'n_estimators': 150}
accuracy : 0.88142797319933
```

In [ ]:
```
n_neighbors = [5, 7, 9, 11, 13, 15, 17, 19, 21]
p=[1,2]
knn_params = dict(n_neighbors=n_neighbors, p=p)

knn = KNeighborsClassifier()
knn_cv=GridSearchCV(knn, knn_params, cv=3)
knn_cv.fit(fea_eng_train,train_y)

print("tuned hyperparameters :(best parameters) ", knn_cv.best_params_)
print("accuracy :", knn_cv.best_score_)
```

```
tuned hyperparameters :(best parameters)  {'n_neighbors': 19, 'p': 2}
accuracy : 0.8971105527638191
```

In [119]:
```
models = [
          ('xgb',  XGBClassifier(
                    objective='multi:softprob',
                    eval_metric='merror',
                    colsample_bytree=1,
                    learning_rate=0.02,
                    max_depth=4,
                    n_estimators=10,
                    )
          ),
          ('gb',  GradientBoostingClassifier()),
          ('svc', SVC(C=0.1, random_state=42)),
          ('rf', RandomForestClassifier(max_depth=7))
          ]
```

In [120]:
```
clf_lr = StackingClassifier(
    estimators=models,
    final_estimator=LogisticRegression(C=0.1, multi_class='multinomial', solver='l
bfgs')
)
```

In [121]:
```
clf_lr.fit(fea_eng_train, train_y)
```

Out[121]:
```
StackingClassifier(cv=None,
                   estimators=[('xgb',
                                XGBClassifier(base_score=0.5, booster='gbtree',
                                              colsample_bylevel=1,
                                              colsample_bynode=1,
                                              colsample_bytree=1,
                                              eval_metric='merror', gamma=0,
                                              learning_rate=0.02,
                                              max_delta_step=0, max_depth=4,
                                              min_child_weight=1, missing=None,
                                              n_estimators=10, n_jobs=1,
                                              nthread=None,
                                              objective='multi:softprob',
                                              random_state=0, reg_alpha=0...
                                              random_state=None,
                                              verbose=0,
                                              warm_start=False))],
                   final_estimator=LogisticRegression(C=0.1, class_weight=None,
                                                      dual=False,
                                                      fit_intercept=True,
                                                      intercept_scaling=1,
                                                      l1_ratio=None,
                                                      max_iter=100,
                                                      multi_class='multinomial',
                                                      n_jobs=None, penalty='l2',
                                                      random_state=None,
                                                      solver='lbfgs',
                                                      tol=0.0001, verbose=0,
                                                      warm_start=False),
                   n_jobs=None, passthrough=False, stack_method='auto',
                   verbose=0)
```

In [122]:
```
labels_stacking_lr = clf_lr.predict(fea_eng_test)
```

In [123]:
```
label_percentages(labels_stacking_lr)
```

Out[123]:
```
((5765, 350, 1205),
 (0.787568306010929, 0.04781420765027322, 0.1646174863387978),
 7320)
```

Submission

In [124]:
```
from google.colab import files
sub.iloc[:,1] = labels_stacking_lr
sub
```

Out[124]:

	S.No	LABELS
0	0	0
1	1	0
2	2	0
3	3	0
4	4	0
...	...	...
7315	7315	0
7316	7316	1
7317	7317	1
7318	7318	1
7319	7319	1

7320 rows × 2 columns

In [125]:
```
sub.to_csv('submissionlabels_stacking_lr.csv', index=False)
files.download('submissionlabels_stacking_lr.csv')
```